

# QPC Keywords

This Keyword Reference Guide lists all the QPC keywords in alphabetical order: A brief explanation of the keywords function is given followed by loose definition of the syntax and examples of usage.

This guide is a combination of the Sinclair QL manuals Keyword section, the (Super)Gold card manual, the Toolkit 2 manual, the SMSQ/E manual and the QPC manual.

© 1984 SINCLAIR RESEARCH LIMITED

© MIRACLE SYSTEMS

© 1994-2002 TONY TEBBY

© MARCEL KILGUS





## ALCHP

### RECHP memory management

The function **ALCHP** will allocate the requested amount of memory from the 'common heap' and return the base address of the space.

**RECHP** will return space allocated by **ALCHP** to the 'common heap'

syntax: *number\_of\_bytes := numeric\_expression*

**ALCHP** (*number\_of\_bytes*)  
**RECHP** *base\_address*

example: i. **base = ALCHP (3000)** {allocate 3000 bytes from the heap}  
ii. **RECHP base** {return 3000 bytes allocated in i above}

## ALTKEY

The **ALTKEY** command assigns a string to an 'ALT' keystroke (hold the ALT key down and press another key). The string itself may contain newline characters, or, if more than one string is given, then there will be an implicit newline between the strings. Thus a null string may be put at the end to add a newline to the string.

**ALTKEY** with just character alone will cancel the string associated with that character.

**ALTKEY** alone will cancel all ALTKEY strings.

syntax: **ALTKEY** [*character, strings*]

example: i. **ALTKEY 'r', 'RJOB "SPL",'** {when ALT r is pressed, the command  
ii. **ALTKEY 'r', 'RJOB "SPL" & CHR\$(10)** 'RJOB "SPL"' will be executed}  
iii. **ALTKEY 'r'** {will cancel the ALTKEY string for 'r'}  
iv. **ALTKEY** {cancel all ALTKEY strings}

comment: **ALTKEY** is case dependent i.e. ALT r is not the same as ALT R.

## AJOB SMSQ/E

**AJOB** is used to re-activate jobs which have been suspended.

syntax: *job\_identifier :=* | *job\_number, tag\_number*  
| *job\_number + (tag\_number \* 65536)*  
*id := job\_identifier*

**AJOB** *id | name, priority*

example: i. **AJOB demon,1** {start the Job called 'demon' with a priority of 1}  
ii. **AJOB 2,1,80** {start the job, Job number 2, Tag number 1 with a priority of 80}

comment: If a name is given rather than a Job ID, then the procedure will search for the first Job it can find with the given name.

## ARC

### ARC\_R graphics

**ARC** will draw an arc of a circle between two specified points in the *window* attached to the default or specified channel. The end points of the arc are specified using the *graphics coordinate* system.

Multiple arcs can be drawn with a single **ARC** command.

The end points of the arc can be specified in absolute coordinates (relative to the *graphics origin* or in relative coordinates (relative to the *graphics cursor*). If the first point is omitted then the arc is drawn from the graphics cursor to the specified point through the specified angle.

**ARC** will always draw with absolute coordinates, while **ARC\_R** will always draw relative to the graphics cursor.

syntax: *x:= numeric\_expression*  
*y:= numeric\_expression*  
*angle:= numeric\_expression (in radians)*  
*point:= x,y*

*parameter\_2:= | TO point, angle (1)*  
*| ,point TO point,angle (2)*

*parameter\_1:= | point TO point,angle (1)*  
*| TO point,angle (2)*

**ARC** [*channel,*] *parameter\_1* \*[*parameter\_2*]\*  
**ARC\_R** [*channel,*] *parameter\_1* \*[*parameter\_2*]\*

where (1) will draw from the specified point to the next specified point turning through the specified angle

(2) will draw from the the last point plotted to the specified point turning through the specified angle

- example: i. **ARC 15,10 TO 40,40,PI/2**  
{draw an arc from 15,10 to 40,40 turning through PI/2 radians}
- ii. **ARC TO 50,50,PI/2**  
{draw an arc from the last point plotted to 50,50 turning through PI/2 radians}
- iii. **ARC\_R 10,10 TO 55,45,0.5**  
{draw an arc, starting 10,10 from the last point plotted to 55,45 from the start of the arc, turning through 0.5 radians}

## AT windows

**AT** allows the print position to be modified on an imaginary row/column grid based on the current character size. **AT** uses a modified form of the *pixel coordinate system* where (row 0, column 0) is in the top left hand corner of the window. **AT** affects the print position in the window attached to the specified or default channel.

syntax: *line:= numeric\_expression*  
*column:= numeric\_expression*

**AT** [*channel,*] *line* , *column*

example: **AT 10,20 : PRINT "This is at line 10 column 20"**

## AUTO

AUTO has been replaced by ED.

## BAUD communications

BAUD sets the baud rate for communication via the serial channels. The speed of the channels be set independently by supplying an optional port number.

If no port number is supplied, then the command will default to SER1.

syntax: *rate:= numeric\_expression*  
*port:= numeric\_expression*

**BAUD** [*port*,] *rate*

The value of the rate numeric expression must be one of the supported baud rates Supported by SMSQ/E on QPC:

300  
600  
1200  
2400  
4800  
9600  
19200  
38400  
57600  
115200

If the selected baud rate is not supported, then an error will be generated.

example: i. **BAUD 2,9600** {set SER2 to 9600 baud}  
ii. **BAUD print\_speed** {set SER1 to 'print\_speed' baud}

## **BEEP** sound

**BEEP** activates the inbuilt sound functions on the QL. **BEEP** can accept a variable number of parameters to give various levels of control over the sound produced. The minimum specification requires only a duration and pitch to be specified. **BEEP** used with no parameters will kill any sound being generated.

syntax:    *duration:= numeric\_expression*            {range -32768..32767}  
          *pitch:= numeric\_expression*                {range 0..255}  
          *grad\_x:= numeric\_expression*              {range -32768..32767}  
          *grad\_y:= numeric\_expression*              {range -8..7}  
          *wrap:= numeric\_expression*                {range 0..15}  
          *fuzzy:= numeric\_expression*                {range 0..15}  
          *random:= numeric\_expression*              {range 0..15}

**BEEP** [ *duration*, *pitch*  
          [,*pitch\_2*, *grad\_x*, *grad\_y*  
          [, *wrap*  
          [, *fuzzy*  
          [, *random* ]]]]

*duration* - specifies the duration of the sound in units of 72 microseconds. A duration of zero will run the sound until terminated by another BEEP command.

*pitch* - specifies the pitch of the sound. A pitch of 1 is high and 255 is low.

*Pitch\_2* - specifies an second pitch level between which the sound will 'bounce'

*grad\_x* - defines the time interval between pitch steps.

*grad\_y* - defines the size of each step, *grad\_x* and *grad\_y* control the rate at which the pitch bounces between levels.

*wrap* - will force the sound to wrap around the specified number of times. If wrap is equal to 15 the sound will wrap around forever:

*fuzzy* - defines the amount of fuzziness to be added to the sound.

*random* - defines the amount of randomness to be added to the sound.

## **BEEPING** sound

**BEEPING** is a function which will return zero (false) if QPC is currently not beeping and a value of one (true) if it is beeping.

syntax:    **BEEPING**

example: **100 DEFine PROCedure be\_ quiet**  
          **110 BEEP**  
          **120 END DEFine**  
          **130 IF BEEPING THEN be\_ quiet**

## BGCOLOUR\_QL

### BGCOLOUR\_24 graphics device 2

**BGCOLOUR\_QL** and **BGCOLOUR\_24** set the screens background colour, the colour behind any open window. To one of the QL compatible colours, or to a plain true colour.

syntax: *colour := numeric\_expression*

<b>BGCOLOUR_QL</b> <i>colour</i>	{range 0 ... 255}
<b>BGCOLOUR_24</b> <i>colour</i>	{range 0 ... 16,777,215}

example: i. **BGCOLOUR\_QL 255** {set background to black / white check}  
ii. **BGCOLOUR\_QL 0,7** {set background to black / white check}  
iii. **BGCOLOUR\_QL 0,7,3** {set background to black / white check}  
iv. **BGCOLOUR\_24 40** {set the background to deep blue}

comment: You can get stippled extended colours by cheating. Set two of the QL palette entries (see **PALETTE\_QL**) to the colours you require before calling **BGCOLOUR\_QL**.

## BGET, BPUT WGET, WPUT

### LGET, LPUT, UPUT byte input/output

**BGET** gets 0 or more bytes from the channel. **BPUT** puts 0 or more bytes into the channel.

For **BGET**, each item must be a floating point or integer variable; for each variable, a byte is fetched from the channel. **BGET** will accept a parameter that is a sub-string of a string array to get multiple bytes.

For **BPUT**, each item must evaluate to an integer between 0 and 255; for each item a byte is sent to the output channel. **BPUT** will accept string parameters to put multiple bytes.

**WGET**, **WPUT**, **LGET**, and **LPUT** work like **BGET** and **BPUT**, but they always read a word or long word instead of a byte.

**UPUT** works as **BPUT**, but will never translate the character. Very useful to send translated text to a channel which does use **TRA**, as well as sending printer control codes using **UPUT** to the same channel.

syntax: **BGET** *#channel* [*position*] , *items* {get bytes from a file}  
**BPUT** *#channel* [*position*] , *items* {put bytes onto a file}  
**WGET** *#channel* [*position*] , *items* {get words from a file}  
**WPUT** *#channel* [*position*] , *items* {put words onto a file}  
**LGET** *#channel* [*position*] , *items* {get long words from a file}  
**LPUT** *#channel* [*position*] , *items* {put long words onto a file}  
**UPUT** *#channel* [*position*] , *items* {put bytes onto a file}

example: i. **abcd=2.6 : zz%=243**  
**BPUT #3,abcd+1,zz%** {will put the byte values 4 and 243 after the current file position on the file open on #3}  
ii. **BPUT #3,27,'R1'** {put ESC R1 to channel #3}  
iii. **DIM a\$(10): a\$(10)='** '  
**BGET #3, a\$(1 to 6)** {get 6 bytes from #3 into a\$}

comment: Provided no attempt is made to set a file position, the direct I/O routines can be used to send unformatted data to devices which are not part of the file system. If, for example, a channel is opened to an Epson compatible printer (channel #3) then the printer may be put into condensed underline mode by either

**BPUT #3,15,27,45,1**  
or **PRINT #3,chr\$(15);chr\$(27);'-';chr\$(1);** {Which is easier?}

## **BGIMAGE** graphics device 2

**BGIMAGE** will load an image to be used as a background behind any open windows.

syntax: **BGIMAGE** *filename*

example: **BGIMAGE** win1\_wallpaper

comment: Background images must be in the form of a screen snapshot. It is relatively simple to create background images.

```
500 WINDOW SCR_XLIM, SCR_YLIM, 0, 0 : REMark whole screen window
510 ..... draw the wallpaper on the screen
520 SBYTES_0 win1_wallpaper, SCR_BASE, SCR_LLEN * SCR_SIZE
```

## **BIN**

### **BIN\$** conversion functions

**BIN** will convert the supplied binary string into a value. Any character in the string, whose ASCII value is even, is treated as 0, while any character, whose ASCII value is odd, is treated as 1. E.g. **BIN** (".#.#") returns the value 5. The 'digits' '0' to '9' 'A' to 'F' and 'a' to 'f' have their conventional meanings.

**BIN\$** will return a string of sufficient length to represent the value of the specified number of bits of the least significant end of the value.

syntax: *number\_of\_bits* := *numeric\_expression*

```
BIN (binary_string)
BIN$ (value, number_of_bits)
```

example: **PRINT BIN** ("1010")            {will output 10}  
**PRINT BIN\$** (9 , 8)            {will output "00001001"}

## **BLOCK** windows

**BLOCK** will fill a block of the specified size and shape, at the specified position relative to the origin of the *window* attached to the specified, or default *channel*.

**BLOCK** uses *the pixel coordinate system*.

syntax: *width*:= *numeric\_expression*  
*height*:= *numeric\_expression*  
*x*:= *numeric\_expression*  
*y*:= *numeric\_expression*

```
BLOCK [channel,] width, height, x, y, colour
```

example: i. **BLOCK 10,10,5,5,7**        {10x10 pixel white block at 5,5}  
ii. **100** REMark "bar chart"  
**110** CSIZE 3,1  
**120** PRINT "bar chart"  
**130** LET bottom =100 : size = 20 : left = 10  
**140** FOR bar =1 to 10  
**150** LET colour = RND(O TO 255)  
**160** LET height = RND(2 TO 20)  
**170** **BLOCK** size, height, Left+bar\*size, bottom-height,0  
**180** **BLOCK** size-2, height-2, left+bar\*size+1, bottom-height+1,colour  
**190** END FOR bar

## **BORDER windows**

**BORDER** will add a border to the window attached to the specified *channel*, or default channel.

For all subsequent operations except **BORDER** the window size is reduced to allow space for the **BORDER**. If another **BORDER** command is used then the full size of the original window is restored prior to the border being added; thus multiple **BORDER** commands have the effect of changing the size and colour of a single border. Multiple borders are not created unless specific action is taken.

If **BORDER** is used without specifying a colour then a transparent border of the specified width is created.

syntax: *width* = *numeric\_expression*

**BORDER** [*channel*,] *size* [, *colour*]

example: i. **BORDER 10,0,7** {black and white stipple border}  
ii. **100 REMark Lurid Borders**  
**110 FOR thickness = 50 to 2 STEP -2**  
**120 BORDER thickness, RND(0 TO 255)**  
**130 END FOR thickness**  
**140 BORDER 50**

## **CACHE\_OFF**

**CACHE\_ON** memory management

These commands have no effect on QPC.

## CALL SMSQ/E

Machine code can be accessed directly from SBASIC by using the **CALL** command. **CALL** can accept up to 13 long word parameters which will be placed into the 68010 data and address registers (D1 to D7, AO to A5) in sequence.

No data is returned from **CALL**.

syntax:    *address:= numeric\_expression*  
          *data:= numeric\_expression*

**CALL** *address*, \*[*data*]\*       {13 data parameters maximum}

example:  i. **CALL 262144,0,0,0**  
          ii. **CALL 262500,12,3,4,1212,6**

**warning:** Address register A6 should not be used in routines *called using* this command. To return to SBASIC use the instructions:

```
MOVEQ #0, D0  
RTS
```

## CD\_ALLTIME audio CD player

**CD\_ALLTIME** will return the totally elapsed time of the CD.

syntax:    **CD\_ALLTIME**

example:  **x=CD\_ALLTIME**

## CD\_CLOSE

### CD\_EJECT audio CD player

**CD\_CLOSE** will close the CD drive tray.

**CD\_EJECT** will open the CD drive tray.

syntax:    **CD\_CLOSE**  
          **CD\_EJECT**

## CD\_FIRSTTRACK

### CD\_LASTTRACK audio CD player

**CD\_FIRSTTRACK** will return the number of the first track.

**CD\_LASTTRACK** will return the number of the last track.

syntax:    **CD\_FIRSTTRACK**  
          **CD\_LASTTRACK**

example:  i. **x%=CD\_FIRSTTRACK**  
          ii. **x%=CD\_LASTTRACK**

## **CD\_HOUR**

### **CD\_MINUTE, CD\_SECOND** audio CD player

Returns the hour, minute or second of a Redbook address.

syntax:    **CD\_HOUR** *numeric\_expression*  
          **CD\_MINUTE** *numeric\_expression*  
          **CD\_SECOND** *numeric\_expression*

example: i. **h%=CD\_HOUR redbook**  
          ii. **m%=CD\_MINUTE redbook**  
          iii. **s%=CD\_SECOND redbook**

## **CD\_HSG2RED**

### **CD\_RED2HSG** audio CD player

**CD\_HSG2RED** will convert an HSG address to a Redbook address.

**CD\_RED2HSG** will convert a Redbook address to an HSG address.

syntax:    **CD\_HSG2RED** *numeric\_expression*  
          **CD\_RED2HSG** *numeric\_expression*

example i. **red=CD\_HSG2RED hsg**  
          ii. **hsg=CD\_RED2HSG red**

## **CD\_INIT** audio CD player

**CD\_INIT** must be used before anything else in order to initialise the CD drive for SMSQ. After the first call the command is ignored in all subsequent calls. The string parameter is only there for compatibility with QPC1, it is ignored by QPC2.

syntax:    *name* := *string\_expression*

**CD\_INIT** [*name*]

example: **CD\_INIT**

## **CD\_ISPLAYING, CD\_ISCLOSED**

### **CD\_ISINSERTED, CD\_ISPAUSED** audio CD player

These function return a binary value indicating the current status according to the keyword.

Please note that Windows cannot tell whether the tray is closed or not, therefore

**CD\_ISCLOSED** always returns the same result as **CD\_ISINSERTED** when used on QPC2. An empty tray is obviously something the Microsoft geniuses could not imagine.

syntax:    **CD\_ISPLAYING**  
          **CD\_ISCLOSED**  
          **CD\_ISINSERTED**  
          **CD\_ISPAUSED**

example: i. **x%=CD\_ISPLAYING**  
          ii. **PRINT CD\_ISCLOSED**  
          iii. **inserted%=CD\_ISINSERTED**  
          iv. **playing%=CD\_ISPAUSED**

## **CD\_LENGTH** audio CD player

**CD\_LENGTH** will return the total length of the CD.

syntax: **CD\_LENGTH**

example: **x=CD\_LENGTH**

## **CD\_PLAY** audio CD player

**CD\_PLAY** will begin playing the audio CD. Without parameters the whole CD is played. An optional start and end track can be given. The command returns immediately when the CD starts playing. The parameters are given in tracks (bit 31 clear) or in sector units (bit 31 set).

syntax: *start := numeric\_expression*  
*end := numeric\_expression*

**CD\_PLAY** [*start*[,*end*]]

example: i. **CD\_PLAY 3** {start playing from track 3}  
**CD\_PLAY CD\_TRACKSTART(3) + \$80000000** {same as above}

## **CD\_RESUME** audio CD player

**CD\_RESUME** will resume the playing of a paused audio CD.

syntax: **CD\_RESUME**

## **CD\_STOP** audio CD player

**CD\_STOP** will pause playing. If the driver was already in pause mode, a complete stop is performed (as if a new CD was inserted, restart from track 1 and so on)

syntax: **CD\_STOP**

## **CD\_TRACK** audio CD player

**CD\_TRACK** will return the number of the track which is currently being played.

syntax: **CD\_TRACK**

example: **track%=CD\_TRACK**

## **CD\_TRACKLENGTH** audio CD player

**CD\_TRACKLENGTH** will return the length of a track.

syntax: *track := numeric\_expression*

**CD\_TRACKLENGTH** *track*

example: **x=CD\_TRACKLENGTH track**

comment: This is the only function that returns an HSG-number.

## **CD\_TRACKTIME** audio CD player

**CD\_TRACKTIME** will return the number of the track which is currently being played.

syntax: **CD\_TRACKTIME**

example: **PRINT CD\_TRACKTIME**

## **CD\_TRACKSTART** audio CD player

**CD\_TRACKSTART** will return the beginning sector of a track.

syntax: *track := numeric\_expression*

**CD\_TRACKSTART** *track*

example: **x=CD\_TRACKSTART track**

## **CHAR\_DEF** windows

The QPC display driver has two character founts built in. The first provides patterns for the values 32 (space) to 127 (copyright), while the second provides patterns for the values 127 (undefined) to 191 (down arrow). For each character the display driver will use the appropriate pattern from the first fount, if there is one, failing that, it will use the appropriate pattern from the second fount, failing that, it will use the first defined pattern in the second fount.

The command **CHAR\_DEF** is used to set or reset one or both character founts.

Setting a fount address to zero will force the built in founts to be used.

All windows which are opened after using **CHAR\_DEF** now will use the new system fonts (except if they define their own fonts, of course).

Channels already open will not use the new fonts automatically for various reasons: the most obvious is, that if the font file did not contain any font data, you will not be able to correct this as all characters printed will look like complete rubbish.

To change the fonts on channels already open use the **CHAR\_USE** command.

syntax: **CHAR\_DEF** *font1, font2*

example: i. **CHAR\_DEF addr1, addr2** {use the substitute founts at, addr1 and addr2}  
ii. **CHAR\_DEF 0, addr2** {the built in first fount will be used, addr2 points to a substitute second fount}  
iii. **CHAR\_DEF 0,0** {reset both founts for window #1}

## CHAR\_INC windows

**CHAR\_INC** will set the character and line spacing for the specified or default window.

The QPC display driver assumes that all characters are 5 pixels wide by 9 pixels high. Other sizes are obtained by doubling the pixels or by adding blank pixels between characters. It is possible, to set any horizontal and vertical spacing. If the increment is set to less than the current character size (set by **CSIZE**) then extreme caution is required as it will be possible for the display driver to write characters (at the right hand side or bottom of the window) partly outside the window. The windows should not come closer to the bottom or right hand edges of the screen than the amount by which the increment specified is smaller than the character spacing set by **CSIZE**.

syntax: *x\_inc := numeric\_expression*  
*y\_inc := numeric\_expression*

**CHAR\_INC** [*#channel,* ] *x\_inc, y\_inc*

example: If there is a 3x6 character fount in a file called 'f3x6' (length 875 bytes), then a 127 column by 36 row screen can be set up:

```
10 WINDOW 512-2,256-3,0,0      :REMark clear of edges of screen
20 CSIZE 0,0                    :REMark spacing 6x10
30 CHAR_INC 4,7                 :REMark spacing 4x7
:
70 fount = ALCHP (875)         :REMark reserve space for fount
80 LBYTES f3x6, fount          :REMark load fount
90 CHAR_USE fount,0            :REMark single fount only
```

comment: The character increments specified are cancelled by a **CSIZE** command.

## CHAR\_USE windows

The QPC display driver has two character founts built in. The first provides patterns for the values 32 (space) to 127 (copyright), while the second provides patterns for the values 127 (undefined) to 191 (down arrow). For each character the display driver will use the appropriate pattern from the first fount, if there is one, failing that, it will use the appropriate pattern from the second fount, failing that, it will use the first defined pattern in the second fount.

The command **CHAR\_USE** is used to set or reset one or both character founts.

Setting a fount address to zero will force the built in founts to be used.

syntax: **CHAR\_USE** [*#channel,* ] *address1, address2*

example: i. **CHAR\_USE #3, addr1, addr2** {the window attached to channel 3, will use the substitute founts at, addr1 and addr2}  
ii. **CHAR\_USE #2, 0, addr2** {in window 2, the built in first fount will be used, addr2 points to a substitute second fount}  
iii. **CHAR\_USE 0,0** {reset both founts for window #1}

## CHK\_HEAP

This enables debugging routines in the SBASIC core for SBASIC development and should not be used by any users.

## CHR\$ BASIC

**CHR\$** is a function which will return the character whose value is specified as a parameter:  
**CHR\$** is the inverse of **CODE**.

syntax: **CHR\$(numeric\_expressen)**

example: i. **PRINT CHR\$(27)** {print ASCII escape character}  
ii. **PRINT CHR\$(65)** {print A}

## CIRCLE, CIRCLE\_R ELLIPSE, ELLIPSE\_R graphics

**CIRCLE** will draw a circle (or an ellipse at a specified angle) on the screen at a specified position and size. The circle will be drawn in the *window* attached to the specified or default channel.

**CIRCLE** uses the *graphics coordinate system* and can use absolute coordinates (i.e. relative to the *graphics origin*), and relative coordinates (i.e. relative to the *graphics cursor*). For relative coordinates use **CIRCLE\_R**.

Multiple circles or ellipses can be plotted with a single call to **CIRCLE**. Each set of parameters must be separated from each other with a semi colon (;)

The word **ELLIPSE** can be substituted for **CIRCLE** if required.

syntax: *x:= numeric\_expression*  
*y:= numeric\_expression*  
*radius:= numeric\_expression*  
*eccentricity:= numeric\_expression*  
*angle:= numeric\_expression* {range 0..2PI}

*parameters:=* | *x, y,* (1)  
| *radius, eccentricity, angle* (2)

where (1) will draw a circle  
(2) will draw an ellipse of specified eccentricity and angle

**CIRCLE** [*channel,*] *parameters\**[; *parameters*]\*

*x* - horizontal offset from the graphics origin or graphics cursor  
*y* - vertical offset from the graphics origin or graphics cursor  
*radius* - radius of the circle eccentricity the ratio between the major and minor axes of an ellipse.  
*Angle* - the orientation of the major axis of the ellipse relative to the screen vertical. The angle must be specified in radians.

example: i. **CIRCLE 50,50,20** {a circle at 50,50 radius 20}  
ii. **CIRCLE 50,50,20,0.5,0** {an ellipse at 50,50 major axis 20 eccentricity 0.5 and aligned with the vertical axis}

## CKEYOFF CKEYON pointer interface

**CKEYOFF** will disable the use of the cursor keys to move the pointer around the screen.

**CKEYON** will re-enable the use of the cursor keys to move the pointer around the screen.

syntax: **CKEYOFF**  
**CKEYON**

## **CLCHP** memory management

**CLCHP** will release all space in the 'common heap' which has been allocated with ALCHP.

syntax: **CLCHP**

comment: **CLEAR** and **NEW** will also release all space allocated in the common heap.

## **CLEAR**

**CLEAR** will clear out the SBASIC variable area for the current program and will release the space for SMSQ/E.

syntax: **CLEAR**

example: **CLEAR**

comment: **CLEAR** can be used to restore to a known state the SBASIC system. For example, if a program is broken into (or stops due to an error) while it is in a procedure then SBASIC is still in the procedure even after the program has stopped. **CLEAR** will reset the SBASIC. {See **CONTINUE**, **RETRY**.}

## **CLOCK** timekeeping

**CLOCK** is a procedure to set up a resident digital clock using the QPC's system clock. If no window is specified, then a default window is set up in the top RHS of the monitor mode default channel 0. This window is 60 by 20 pixels. The clock may be invoked to execute within a window set up by SBASIC. In this case the clock job will be removed when the window is closed.

syntax: **CLOCK** [#channel,] [string ]

The string is used to define the characters written to the clock window: any character may be written except \$ or %. If a dollar sign is found in the string then the next character is checked and

\$d or \$D will insert the three characters of the day of week,  
\$m or \$M will insert the three characters of the month.

If a percentage sign is found then

%y or %Y will insert the two digit year  
%d or %D will insert the two digit day of month  
%h or %H will insert the two digit hour  
%m or %M will insert the two digit minute  
%s or %S will insert the two digit second

The default string is '\$d %d \$m %h/%m/%s ' a newline should be forced by padding out a line with spaces until the right hand margin of the window is reached.

example: **10 OPEN #6,'scr\_156x10a32x16'**  
**20 INK #6,0: PAPER #6,4**  
**30 CLOCK #6,'QPC time %h:%m'**

## CLOSE devices

**CLOSE** will close all channel numbers #3 and above, or the specified *channels*. Any *window* associated with the channel will be deactivated.

It will not report an error if a channel is not open.

syntax: *channel:= numeric\_expression*

**CLOSE** [ \**channel*, \* ]

example: i. **CLOSE #4**  
ii. **CLOSE #input\_channel**  
iii. **CLOSE #3, #4, #7** {close channels #3, #4 and #7}

## CLS windows

Will clear the *window* attached to the specified or default *channel* to current **PAPER** colour, excluding the border if one has been specified. **CLS** will accept an optional parameter which specifies if only a part of the window must be cleared.

syntax: *part:= numeric\_expression*

**CLS** [*channel*,] [*part*]

where: *part* = 0 - whole screen (default if no parameter)  
*part* = 1 - top excluding the cursor line  
*part* = 2 - bottom excluding the cursor line  
*part* = 3 - whole of the cursor line  
*part* = 4 - right end of cursor line including the cursor position

example: i. **CLS** {the whole window}  
ii. **CLS 3** {clear the cursor line}  
iii. **CLS #2,2** {clear the bottom of the window on channel 2}

## CODE

**CODE** is a function which returns the internal code used to represent the specified character. If a string is specified then **CODE** will return the internal representation of the first character of the string.

**CODE** is the inverse of **CHR\$**.

syntax: **CODE** (*string\_expression*)

example: i. **PRINT CODE("A")** {prints 65}  
ii. **PRINT CODE ("SBASIC")** {prints 83}

## COLOUR\_NATIVE, COLOUR\_PAL

### COLOUR\_QL, COLOUR\_24 graphics device 2

COLOUR\_NATIVE, COLOUR\_PAL, COLOUR\_QL, and COLOUR\_24 will select the colour definition used by **INK, PAPER, STRIP, BORDER, BLOCK**.

**COLOUR\_QL** selects the standard QL colour definitions (the QL colours can be mapped to colours other than the standard black, blue, red, magenta, green, cyan, yellow and white). This is the default colour scheme for SBASIC and it's daughter jobs.

**COLOUR\_PAL** selects the 256 colour palette mapped definition.

**COLOUR\_24** selects the true colour (24 bit) definition.

**COLOUR\_NATIVE** selects the native colour definition - the significance of the colour numbers specified by **INK, PAPER**, etc. depends on the hardware.

syntax: **COLOUR\_QL**  
**COLOUR\_PAL**  
**COLOUR\_24**  
**COLOUR\_NATIVE**

example: **200 COLOUR\_24** {select true colour mode}  
**210 BORDER 2, 128\*65536 + 128\*256 +128** {grey border}  
**220 BORDER 2,\$808080** {grey border for hexadecimal hackers}

comment: The commands have no effect on any other programs executing. When an SBASIC program starts executing, it is set to QL colour definition.

## CONTINUE

### RETRY error handling

**CONTINUE** allows a program which has been halted to be continued. **RETRY** allows a program statement which has reported an error to be re-executed.

As the **RETRY** and **CONTINUE** exit from an error clause without resetting the **WHEN ERROR**, they can also be used to exit to a different part of the program via an optional line number.

syntax: *line\_number := numeric\_expression*  
**CONTINUE** [*line\_number*]  
**RETRY** [*line\_number*]

example: **CONTINUE**  
**RETRY 1040**

**warning:** A program can only continue if:

1. No new lines have been added to the program
2. No new variables have been added to the program
3. No lines have been changed

The value of variables may be set or changed.

## **COPY**

### **COPY\_N** devices

**COPY** will copy a file from an input device to an output device until an end of file marker is detected. **COPY\_N** will not copy the header (if it exists) associated with a file and will allow Disk files to be correctly copied to another type of device.

Headers are associated with directory-type devices and should be removed using **COPY\_N** when copying to non-directory devices, e.g. **flp1** is a directory device; **ser1** is a non-directory device.

syntax: **COPY** device **TO** device  
**COPY\_N** device **TO** device

It must be possible to input from the source device and it must be possible to output to the destination device.

example: i. **COPY flp1\_data\_file TO con\_** {copy to default window}  
ii. **COPY neti\_3 TO flp1\_data** {copy data from network station to flp\_data.}  
iii. **COPY\_N flp1\_test\_data TO ser1\_** {copy flp1\_test\_data to serial port 1 removing header information}

## **COPY\_O**

## **COPY\_H**

### **WCOPY** devices

Files in SMSQ/E have headers which provide useful information about the file that follows. It depends on the circumstances whether it is a good idea to copy the header of a file when the file is copied.

It is a good idea to copy the header when:

- a) copying an executable program file so that the additional file information is preserved,
- b) copying a file over a pure byte serial link so that the communications software will know in advance the length of the file.

It is a bad idea to copy the header when:

- c) copying a text file to a printer because the header will be likely to have control codes and spurious or unprintable characters.

The general rules used by the **COPY** procedures in SMSQ/E, are that the header is only copied if there is additional information in the header. This caters for cases (a) and (c) above. A **COPY\_N** command is included for compatibility with the standard QL **COPY\_N**: this never copies the header. A **COPY\_H** command is included to copy a file with the header to cater for case (b) above. (Note that the standard QL command **COPY** always copies the header.) Neither **COPY\_N** nor **COPY\_H** need ever be used for file to file copying.

A second general rule used by the **COPY** (as well as by the **WREN**) procedures is that if the destination file already exists, then the user will be asked to confirm that overwriting the old file is acceptable. The **COPY\_O** (copy overwrite) and the spooler procedures do not extend this courtesy to the user.

If the commands are given with two filenames then the data default directory is used for both files. If, however, only one filename (or, in the case of the wild card procedures, no name at all) is given then the destination will be derived from the destination default:

a) if the destination default is a directory (ending with '\_', set by **DEST\_USE**) then the destination file is the destination default followed by the name.

b) if the destination default is a device (not ending with '\_', set by **SPL\_USE**) then the destination is the destination default unmodified.

syntax:    **COPY** name **TO** name                    {copy a file}  
          **COPY\_O** name **TO** name                {copy a file (overwriting)}  
          **COPY\_N** name **TO** name                {copy a file (without header)}  
          **COPY\_H** name **TO** name                {copy a file (with header)}

These commands can be given with one or two names. The separator 'TO' is used for clarity, you may use a comma instead.

To illustrate the use of the copy command, assume that the data default is **FLP2\_** and the destination default is **FLP1\_**.

example: i. **COPY fred TO old\_fred**                {copies flp2\_fred to flp2\_old\_fred}  
          ii. **COPY fred, ser**                    {copies flp2\_fred to ser}  
          iii. **COPY fred**                        {copies flp2\_fred to flp1\_fred}  
          iv. **SPL\_USE ser**  
          ....  
          **COPY fred**                            {copies flp2\_fred to ser}

The interactive copying procedure **WCOPY** is used for copying all or selected parts of directories. The command may be given with both source and destination wild card names, with one wild card name or with no wild card names at all. Giving the command with no wild card names has the same effect as giving one null name:

**WCOPY**    and    **WCOPY "**    are the same.

If you get confused by the following rules about the derivation of the copy destination, just use **WCOPY** intuitively and look carefully at the prompts.

If the destination is not the destination default device, then the actual destination file name for each copy operation is made up from the actual source file name and the destination wild name. If a missing section of the source wild name is matched by a missing section of the destination wild name, then that part of the actual source file name will be used as the corresponding part of the actual destination name. Otherwise the actual destination file name is taken from the destination wild name. If there are more sections in the destination wild name than in the source wild name, then these extra sections will be inserted after the drive name, and vice versa.

syntax:    **WCOPY** [#channel,] name **TO** name

The separator **TO** is used for clarity, you may use a comma instead.

If the channel is not given (i.e. most of the time), then the requests for confirmation will be sent to the command channel #0. Otherwise confirmation will be sent to the chosen channel, and the user is requested to press one of:

Y (yes)    copy this file  
N (no)    do not copy this file  
A (all)    copy this and all the next matching files.  
Q (quit)   do not copy this or any other files

If the destination file already exists, the user is requested to press one of:

Y (yes)    copy this file, overwriting the old file  
N (no)    do not copy this file  
A (all)    overwrite the old file, and overwrite any other files requested to be copied.  
Q (quit)   do not copy this or any other files

example: If the default data directory is flp2\_, and the default destination is flp1\_

- i. **WCOPY** {would copy all files on flp2\_ to flp1\_}
- ii. **WCOPY flp1\_,flp2\_** {would copy all files on flp1\_ to flp2\_}
- iii. **WCOPY fred** {would copy flp2\_fred to flp1\_fred  
flp2\_freda\_list to flp1\_freda\_list}
- iv. **WCOPY fred,mog** {would copy flp2\_fred to flp2\_mog  
flp2\_freda\_list to flp2\_moga\_list}
- v. **WCOPY \_fred,\_mog** {would copy flp2\_fred to flp2\_mog  
flp2\_freda\_list to flp2\_moga\_list  
flp2\_old\_fred to flp2\_old\_mog  
flp2\_old\_freda\_list to flp2\_old\_moga\_list}
- vi. **WCOPY \_list,old\_\_** {would copy flp2\_jo\_list to flp2\_old\_jo\_list  
flp2\_freda\_list to flp2\_old\_freda\_list}
- vii. **WCOPY old\_\_list,flp1\_\_** {would copy flp2\_old\_jo\_list to flp1\_jo\_list  
flp2\_old\_freda\_list to flp1\_freda\_list}

### **COS math functions**

**COS** will compute the cosine of the specified argument.

syntax: *angle:= numeric\_expression* {range -10000..10000 in radians}

**COS** (*angle*)

- example: i. **PRINT COS(theta)**  
ii. **PRINT COS(3.141592654/2)**

### **COT maths functions**

**COT** will compute the cotangent of the specified argument.

syntax: *angle:= numeric\_expression* {range -30000..30000 in radians}

**COT** (*angle*)

- example: i. **PRINT COT(3)**  
ii. **PRINT COT(3.141592654/2)**

## **CSIZE** window

Sets a new character size for the *window* attached to the specified or default *channel*.

The standard size in 512 x 256 QL colour mode is, 0,0 in 512 mode and 2,0 in 256 mode.

In other screen resolutions the standard size 0,0.

Width defines the horizontal size of the character space. Height defines the vertical size of the character space. The character size is adjusted to fill the space available.

---

<b>width</b>	<b>size</b>	<b>height</b>	<b>size</b>
0	6 pixels	0	10 pixels
1	8 pixels	1	20 pixels
2	12 pixels		
3	16 pixels		

---

syntax: *width*:= *numeric\_expression* {range 0..3}  
*height*:= *numeric\_expression* {range 0..1}

**CSIZE** [*channel*,] *width*, *height*

example: i. **CSIZE 3,0**  
ii. **CSIZE 3,1**

## **CURSEN**

### **CURDIS** windows

The function **INKEY\$** is designed so that keystrokes may be read from the keyboard without enabling the cursor. Two procedures are supplied to enable and disable the cursor. When the cursor is enabled, it will usually appear solid (inactive). The cursor will start to flash (active) when the keyboard queue has been switched to the window with the cursor (e.g. by an **INKEY\$**).

syntax: **CURSEN** #*channel* {enable the cursor}  
**CURDIS** #*channel* {disable the cursor}

example: **10 CURSEN** {enable the cursor in window #1}  
**20 in\$=INKEY\$ (#1,250)** }wait for up to 5 seconds for a character  
from the keyboard. If nothing is typed within  
the 5 seconds, then in\$ will be set to a null  
string ("")}

**30 CURDIS**

comment: Note that while **CURSEN** and **CURDIS** default to channel #1, like most I/O commands, **INKEY\$** defaults to channel #0.

## **CURSOR windows**

**CURSOR** allows the screen cursor to be positioned anywhere in the window attached to the specified or default *channel*.

**CURSOR** uses the *pixel coordinate system* relative to the window origin and defines the position for the top left hand corner of the cursor. The size of the cursor is dependent on the character size in use.

If **CURSOR** is used with four parameters then the first pair is interpreted as graphics coordinates (using the graphics coordinate system) and the second pair as the position of the cursor (in the pixel coordinate system) relative to the first point.

This allows diagrams to be annotated relatively easily.

syntax: *x:= numeric\_expression*  
*y:= numeric\_expression*

**CURSOR** [*channel*,] *x, y* [,*x, y*]

example: i. **CURSOR 0,0**  
ii. **CURSOR 20,30**  
iii. **CURSOR 50,50,10,10**

## **DATA**

### **READ**

### **RESTORE BASIC**

**READ**, **DATA** and **RESTORE** allow embedded data, contained in a SBASIC program, to be assigned to variables at run time.

**DATA** is used to mark and define the data, **READ** accesses the data and assigns it to variables and **RESTORE** allows specific data to be selected.

**DATA** allows data to be defined within a program. The data can be read by a **READ** statement and the data assigned to variables. A **DATA** statement is ignored by SBASIC when it is encountered during normal processing.

syntax: **DATA** \*[*expression*,]\*

**READ** reads data contained in **DATA** statements and assigns it to a list of variables. Initially the data pointer is set to the first **DATA** statement in the program and is incremented after each **READ**. Re-running the program will not reset the data pointer and so in general a program should contain an explicit **RESTORE**.

An error is reported if a **READ** is attempted for which there is no **DATA**.

syntax: **READ** \*[*identifier*,]\*

**RESTORE** restores the data pointer, i.e. the position from which subsequent **READs** will read their data. If **RESTORE** is followed by a line number then the data pointer is set to that line. If no parameter is specified then the data pointer is reset to the start of the program.

syntax: **RESTORE** [*line\_number*]

- example: i. **100 REMark Data statement example**  
**110 DIM weekdays\$(7,4)**  
**120 RESTORE**  
**130 FOR count= 1 TO 7 : READ weekdays\$(count)**  
**140 PRINT weekday\$**  
**150 DATA "MON","TUE","WED","THUR","FRI"**  
**160 DATA "SAT","SUN"**
- ii. **100 DIM month\$(12,9)**  
**110 RESTORE**  
**120 REMark Data statement example**  
**130 FOR count=1 TO 12 : READ month\$(count)**  
**140 PRINT month\$**  
**150 DATA "January", "February", "March"**  
**160 DATA "April", "May", "June"**  
**170 DATA "July", "August", "September"**  
**180 DATA "October", "November", "December"**

**warning:** An implicit **RESTORE** is not performed before running a program. This allows a single program to run with different sets of data. Either include a **RESTORE** in the program or perform an explicit **RESTORE** or **CLEAR** before running the program.

## **DATA\$** **PROG\$** **DESTD\$** defaults functions

DATA\$, PROG\$, and DESTD\$ are functions to find the current data, program, and destination defaults.

syntax: **DATAD\$** {find the data default}  
**PROGD\$** {find the program default}  
**DESTD\$** {find the destination default}

comment: The functions to find the individual defaults should be used without any parameters.

- example: i. **IF DATAD\$<>PROGD\$: PRINT 'Separate directories'**
- ii. **DEST\$=DESTD\$**  
**IF DEST\$ (LEN (DEST\$))=' ': PRINT 'Destination!' DEST\$**

## **DATA\_USE** data default

**DATA\_USE** is used to set a default, which is added to most of the filing system commands. If you do not supply a complete SMSQ/E filename in the command, the **DATA\_USE** default will be added to the beginning of the supplied filename.

If the supplied filename is not found in the system, Then the **DATA\_USE** default will be added to the beginning of the supplied filename, and another attempt will be made to execute the command.

syntax: *directory\_name := device\*[subdirectory\_]\**

**DATA\_USE** *directory\_name*

- example: **100 DATA\_USE win1\_programs\_**  
**110 DIR** {Gives a directory of "win1\_programs\_"}  
**120 LOAD draw** {Loads the program "win1\_programs\_draw"}

comment: If the directory name supplied does not end with '\_', '\_' will be appended to the directory name.



## **DDOWN**

## **DUP**

## **DNEXT** directory navigation

These three commands are provided to move through a directory tree.

**DDOWN** moves down through the directory tree, **DUP** move up through the directory tree, and **DNEXT** moves up and then down a different branch of the tree.

It is not possible to move up beyond the drive name using the **DUP** command. At no time is the default name length allowed to exceed 32 characters.

These commands operate on the data default directory. By appending directories onto the end of, or stripping directories off of the end of the default. Under certain conditions they may operate on the other defaults as well:

If the program default is the same as the data default, then the two defaults are linked and these commands will operate on the **PROG\_USE** default as well.

If the destination default ends with '\_' (i.e. it is a default directory rather than a default device), then these commands will operate on the destination default.

syntax: **DDOWN** *name*  
**DUP**  
**DNEXT** *name*

example:

defaults	data	program	destination
initial values	flp2_	flp1_	ser
<b>DDOWN</b> john	flp2_john_	flp1_	ser
<b>DNEXT</b> fred	flp2_fred_	flp1_	ser
<b>PROG_USE</b> flp2_fred	flp2_fred_	flp2_fred_	ser
<b>DNEXT</b> john	flp2_john_	flp2_john_	ser
<b>DUP</b>	flp2_	flp2_	ser
<b>DEST_USE</b> flp1	flp2_	flp2_	flp1_
<b>DDOWN</b> john	flp2_john_	flp2_john_	flp1_john_
<b>SPL_USE</b> ser1c	flp2_john_	flp2_john_	ser1c

## DEFine FuNction

### END DEFine functions and procedures

**DEFine FuNction** defines a SBASIC function. The sequence of statements between the **DEFine** function and the **END DEFine** constitute the function. The function definition may also include a list of *formal parameters* which will supply data for the function. Both the formal and *actual parameters* must be enclosed in brackets. If the function requires no parameters then there is no need to specify an empty set of brackets.

*Formal parameters* take their type and characteristics from the corresponding *actual parameters*. The type of data returned by the function is indicated by the type appended to the function identifier. The type of the data returned in the **RETURN** statement must match.

An answer is returned from a function by appending an expression to a **RETURN** statement. The type of the returned data is the same as type of this expression.

A function is activated by including its name in a SBASIC expression.

Function calls in SBASIC can be recursive; that is, a function may call itself directly or indirectly via a sequence of other calls.

syntax: *formal\_parameters* = (*expression* \*[, *expression*]\*)  
*actual\_parameters* := (*expression* \*[, *expression*]\*)

*type* := | \$  
          | %  
          |

```
DEF FuNction identifier type {formal_parameters}  
  [LOCaL identifier *[, identifier]*]  
  statements  
  RETuRn expression  
END DEFine
```

**RETURN** can be at any position within the procedure body. **LOCaL** statements must precede the first executable statement in the function.

example: 10 **DEFine FuNction** mean(a, b, c)  
          20 **LOCaL** answer  
          30 **LET** answer = (a + b + c)/3  
          40 **RETuRn** answer  
          50 **END DEFine**  
          60 **PRINT** mean(1,2,3)

comment: To improve legibility of programs the name of the function can be appended to the **END DEFine** statement. However, the name will not be checked by SBASIC.

## DEFine PROCedure

### END DEFine functions and procedures

**DEFine PROCedure** defines a SBASIC procedure. The sequence of statements between the **DEFine PROCedure** statement and the **END DEFine** statement constitutes the procedure. The procedure definition may also include a list of *formal parameters* which will supply data for the procedure. The *formal parameters* must be enclosed in brackets for the procedure definition, but the brackets are not necessary when the procedure is called. If the procedure requires no parameters then there is no need to include an empty set of brackets in the procedure definition.

Formal parameters take their type and characteristics from the corresponding *actual parameters*.

Variables may be defined to be **LOCal** to a procedure. Local variables have no effect on similarly named variables outside the procedure. If required, local arrays should be dimensioned within the **LOCAl** statement.

The procedure is called by entering its name as the first item in a SBASIC statement together with a list of actual parameters. Procedure calls in SBASIC are recursive that is, a procedure may call itself directly or indirectly via a sequence of other calls.

It is possible to regard a procedure definition as a command definition in SBASIC; many of the system commands are themselves defined as procedures.

syntax: *formal\_parameter* := (*expression* \*[, *expression*]\*)  
*actual\_parameters* := *expression* \*[, *expression*]\*

```
DEFine PROCedure identifier [formal_parameters]  
  [LOCAl identifier *[, identifier]*]  
  statements  
  [RETurn]  
END DEFine
```

**RETURN** can appear at any position within the procedure body. If present the **LOCAl** statement must be before the first executable statement in the procedure. The **END DEFine** statement will act as an automatic return.

example: i. **100 DEFine PROCedure start\_screen**  
          **110 WINDOW 100,100,10,10**  
          **120 PAPER 7 : INK 0 : CLS**  
          **130 BORDER 4,255**  
          **140 PRINT "Hello Everybody"**  
          **150 END DEFine**  
          **160 start\_screen**

ii. **100 DEFine PROCedure slow\_scroll(scroll\_limit)**  
      **110 LOCAl count**  
      **120 FOR count =1 TO scroll**  
      **130 SCROLL 2**  
      **140 END FOR count**  
      **150 END DEFine**  
      **160 slow\_scroll 20**

comment: To improve legibility of programs the name of the procedure can be appended to the **END DEFine** statement. However, the name will not be checked by SBASIC.

## **DEG** maths functions

**DEG** is a function which will convert an angle expressed in radians to an angle expressed in degrees.

syntax: **DEG**(*numeric\_expression*)

example: **PRINT DEG(PI/2)** {will print 90}

## **DELETE**

### **WDEL** directory devices

**DELETE** will remove a file from the directory of the directory device specified.

**WDEL** will remove multiple files from the directory of the directory device specified, using wild card names.

No error is generated if the file is not found.

syntax: **DELETE** *name* {delete one file}  
**WDEL** [#*channel*,] *name* {delete files}

example: i. **DELETE flp1\_old\_data**  
ii. **DELETE win1\_letter\_file**

For **WDEL** both the channel and the name are optional.

iii. **WDEL** {delete files from current directory}  
iv. **WDEL \_list** {delete all \_list files from current directory}

comment: Unless a channel is specified, the wild card deletion procedures use the command window #0 to request confirmation of deletion. There are four possible replies:

<b>Y</b>	(yes)	delete this file
<b>N</b>	(no)	do not delete this file
<b>A</b>	(all)	delete this and all the next matching files
<b>Q</b>	(quit)	do not delete this or any of the next files

## **DEL\_DEFB** memory management

**DEL\_DEFB** will delete file definition blocks from common heap.

Making large allocations in the common heap and then accessing a drive for the first time can cause a terrible heap disease called 'large scale fragmentation' where the drive definition blocks become widely scattered in the heap leaving large holes that cease to be available except as heap entries (i.e. you cannot load programs into them). A simple but dangerous cure is to delete the drive definition blocks.

syntax: **DEL\_DEFB**

comment: Although there are precautions within the procedure **DEL\_DEFB** to minimise damage, care should be taken to avoid using this command while any directory device is active.

## **DEST\_USE destination default**

**DEST\_USE** is used to set a default, which is used to find the destination filename when the file copying and renaming commands (**SPL**, **COPY**, **RENAME** etc.) are used with only one filename.

If the supplied filename is not found in the system, Then the **DEST\_USE** default will be added to the beginning of the supplied filename, and another attempt will be made to execute the command.

syntax: *directory\_name := device\*[subdirectory\_]\**

**DEST\_USE** *directory\_name*

example: **100 DEST\_USE win1\_programs\_**  
**110 COPY flp1\_john TO fred** {Copies the file "flp1\_john" to the file "win1\_programs\_fred"}

comment: There is a special form of the **DEST\_USE** command which does not append '\_' to the name given. Notionally this provides the default destination device for the spooler. See **SPL\_USE**.

## **DEVTYPE devices**

**DEVTYPE** returns a value indicating whether the specified or default channel is open to a window, or to a file.

Only the most significant bit, and the two least significant bits should be tested. All other bits are unidentified. The value returned is negative if the channel is not open. Bit 0 indicates that the channel is open to a window, Bit 1 indicates that the channel is open to a file.

The values returned in the two least significant bits are –

- 0 - Purely serial device
- 1 - Window
- 2 - Direct access file

syntax: **DEVTYPE** [ (# *channel* ) ]

example: i. **PRINT DEVTYPE**  
ii. **PRINT DEVTYPE (#4)**  
iii. **PRINT 3 && DEVTYPE(#6)**  
iv. **IF DEVTYPE(#4) < 0 then PRINT "Channel is closed"**

## **DEV\_LIST**

### **DEV\_USE\$, DEV\_NEXT\$ devices**

**DEV\_LIST** is a command to list to the specified or default channel the DEV device allocations.

**DEV\_USE\$** returns the DEV device usage for the supplied DEV device number.

**DEV\_NEXT\$** returns the next DEV in the chain after the supplied device number.

syntax: *device := numeric\_expression*

**DEV\_LIST** [*#channel*]  
**DEV\_USE\$** (*device*)  
**DEV\_NEXT\$** (*device*)

example: i. **DEV\_LIST#3** {Lists current DEV's to #3}  
ii. **PRINT DEV\_USE\$(3)** {Prints the usage for DEV3\_}  
iii. **PRINT DEV\_NEXT\$(1)** {Prints the next DEV in the chain after dev1\_}

## **DEV\_NEXT** directory devices

**DEV\_NEXT** returns the next DEV after the specified DEV.

syntax: **DEV\_NEXT** ( *numeric\_expression* )

example: **PRINT DEV\_NEXT(1)** {prints the next DEV In the chain after DEV1}

## **DEV\_USEN** directory devices

**DEV\_USEN** allows renaming of the DEV device. Both **DEV\_USE** or **DEV\_USEN** with one parameter will rename the DEV device, **DEV\_USEN** without parameter will reset the name of DEV back to DEV.

syntax: **DEV\_USEN** [ *name* ]

example: i. **DEV\_USEN mdv** {DEV is now called MDV}  
ii. **DEV\_USEN** {and now its name is DEV again}

## **DEV\_USE** directory devices

**DEV\_USE** allows you to attach a DEV device to a real directory.

There is a variation on the **DEV\_USE** call which enables the setting up of default chains. If you put another number at the end of the **DEV\_USE** command it will be taken as the DEV to try if the open fails. This next DEV can also chain to another DEV. The DEV driver stops chaining when all DEV's in the chain have been tried.

syntax: **DEV\_USE** [*device\_number* , *real\_directory* [ ,*chain* ] | *device* ]

example: i. **DEV\_USE 1,ram1\_** {dev1\_ is equivalent to ram1\_}  
ii. **DEV\_USE 2,flp1\_letters\_** {dev2\_ is equivalent to flp1\_letters\_}  
iii. **DEV\_USE 3,win1\_work\_new\_** {dev3\_ is equivalent to win1\_work\_new\_}  
iv. **DEV\_USE 4, ram2\_,5** {dev4\_ is equivalent to ram2\_}  
v. **DEV\_USE 5,flp1\_latest\_,6** {dev5\_ is equivalent to flp1\_latest\_}  
vi. **DEV\_USE 6,win1\_work\_,4** {dev6\_ is equivalent to win1\_work\_}

comment: Unlike **PROG\_USE** and **DATA\_USE**, the underscore at the end is significant. Thus, entering the above commands.

<b>OPEN#3,dev1_f1</b>	Opens "ram1_f1"
<b>OPEN#3,dev2_bankmanager</b>	Opens "flp1_letters_bankmanager"
<b>OPEN#3,dev3_f1</b>	Opens "win1_work_newf1"
<b>DELETE dev3__junk</b>	Deletes "win1_work_new_junk"
<b>LOAD dev4_prog_bas</b>	Tries "ram2_prog_bas", then "flp1_latest_prog_bas", and then finally "win1_work_prog_bas"
<b>LOAD dev5_DiskCheck</b>	Tries "flp1_latest_DiskCheck", then "win1_work_DiskCheck", and finally "ram2_DiskCheck"

**DELETE** does not chain with DEV.

The DEV name can be changed by specifying a three letter name of string.

**DEV\_USE** without any parameters will reset the name to DEV.

<b>DEV_USE 1,flp2_myprogs_</b>	"dev1_" is "myprogs_ "on drive 2}
<b>DEV_USE 2,flp1_ex_,1</b>	"dev2_" is "flp1_ex_", or "flp2_myprogs_"
<b>DEV_USE flp</b>	"flp1_" is now really "flp2_myprogs_" and "flp2_" is "flp1_ex_"
<b>DEV_USE</b>	"flp1_" is now "flp1_" again

## **DIM arrays**

Defines an array to SBASIC. *String*, *integer* and *floating point* arrays can be defined. String arrays handle fixed length strings and the final *index* is taken to be the string length.

Array indices run from 0 up to the maximum index specified in the **DIM** statement; thus **DIM** will generate an array with one more element in each dimension than is actually specified.

When an array is specified it is initialised to zero for a numeric array and zero length strings for a string array.

syntax: *index:= numeric\_expression*  
*array:= identifier(index \*[, index]\*)*

**DIM array \*[, array] \***

example: i. **DIM string\_array\$(10,10,50)**  
ii. **DIM matrix(100,100)**

## **DIMN arrays**

**DIMN** is a function which will return the maximum size of a specified dimension of a specified array. If a dimension is not specified then the first dimension is assumed. If the specified dimension does not exist or the identifier is not an array then zero is returned.

syntax: *array:= identifier*  
*index:= numeric\_expression* {1 for dimension 1, etc.}

**DIMN(array [, dimension])**

example: consider the array defined by: DIM a(2,3,4)  
i. **PRINT DIMN(A,1)** {will print 2}  
ii. **PRINT DIMN(A,Z)** {will print 3}  
iii. **PRINT DIMN(A,3)** {will print 4}  
iv. **PRINT DIMN(A)** {will print 2}  
v. **PRINT DIMN(A,4)** {will print 0}

## **DIR** directory devices

**DIR** will obtain and display in the *window* attached to the specified or default *channel* the directory of the disk drive in the specified directory device.

syntax: **DIR** *device*

The device specification must be a valid directory device

The directory format output by **DIR** is as follows:

*format*:= disk format operating system QDOS or MSDOS  
*density*:= formatting density SD, DD, or HD  
*free\_sectors*:= the number of free sectors  
*available\_sectors*:= the maximum number of sectors on this disk drive  
*file\_name*:= a SBASIC file name

screen format: *Volume name format density*  
*free\_sectors* | *available\_sectors* **sectors**  
*file\_name*  
.....  
*file\_name*

example: i. **DIR** flp1\_  
ii. **DIR** "dev2\_"  
iii. **DIR** "win" & hard\_drive\_number\$ & "\_"

screen format: **BASIC QDOS HD**  
**183 / 221 sectors**  
**demo\_1**  
**demo\_1\_old**  
**demo\_2**

## **DISP\_BLANK**

**DISP\_BLANK** has no effect in QPC.

## **DISP\_COLOUR** graphics device 2

**DISP\_COLOUR** specifies the colour depth to be used

0 for QL  
1 for 4 bit  
2 for 8 bit  
3 for 16 bit  
4 for 24 bit.

QPC supports mode 0, 2 and 3, but changing the colour depth after boot is not recommended. It is possible to specify the display size immediately after the colour depth.

The parameters from frame rate onwards may be specified, but have no effect in QPC.

syntax: *colour\_depth* := *numeric\_expression*  
*xsize* := *numeric\_expression*  
*ysize* := *numeric\_expression*

**DISP\_COLOUR** *colour\_depth* [, *xsize* [, *ysize* ]]

example: **DISP\_COLOUR 3, 800, 600** {specifies an 800x600 16 bit display}

## **DISP\_INVERSE**

**DISP\_INVERSE** has no effect in QPC.

## **DISP\_RATE**

**DISP\_RATE** has no effect in QPC.

## **DISP\_SIZE** graphics device 2

**DISP\_SIZE** allows the screen resolution to be changed.

**DISP\_SIZE** is much less destructive than **DISP\_COLOUR**, but jobs with Windows outside of the new display area will be removed.

The parameters from frame rate onwards may be specified, but have no effect in QPC.

syntax:    *xsize* := *numeric\_expression*  
          *ysize* := *numeric\_expression*

**DISP\_SIZE** *xsize* [, *ysize* ]

## **DISP\_TYPE** graphics device 2

**DISP\_TYPE** will return a value indicating the type of display you are using.

On QPC the result can be 0 (QL 4 colour mode), 8 (QL 8 colour mode), 16 (8 bit/256 colour mode) or 32 (16 bit/65536 colour mode).

syntax:    **DISP\_TYPE**

example:   **PRINT DISP\_TYPE**

## **DIV** operator

**DIV** is an operator which will perform an integer divide.

syntax:    *numeric\_expression* **DIV** *numeric\_expression*

example:   i. **PRINT 5 DIV 2**                    {will output 2}  
          ii. **PRINT -5 DIV 2**                   {will output -3}

## **DLINE BASIC**

**DLINE** will delete a single line or a range of lines from a SBASIC program.

syntax:    *range*:=        | *line\_number* **TO** *line\_number*    (1)  
                          | *line\_number* **TO**                            (2)  
                          | **TO** *line\_number*                            (3)  
                          | *line\_number*                                (4)

**DLINE** *range\**[,*range*]\*

where    (1) will delete a range of lines  
          (2) will delete from the specified line to the end  
          (3) will delete from the start to the specified line  
          (4) will delete the specified line

example: i. **DLINE 10 TO 70, 80, 200 TO 400**  
          {will delete lines 10 to 70 inclusive, line 80 and lines 200 to 400 inclusive}

          ii. **DLINE**  
              {will delete nothing}

## **DLIST defaults functions**

**DLIST** will display in the default, or specified window the three defaults (data, program, and destination).

syntax:    **DLIST** [*channel*]  
          **DLIST** \name

**DMEDIUM\_NAME\$, DMEDIUM\_DRIVE\$  
 DMEDIUM\_RDONLY, DMEDIUM\_REMOVE  
 DMEDIUM\_DENSITY, DMEDIUM\_FORMAT  
 DMEDIUM\_TYPE, DMEDIUM\_TOTAL  
 DMEDIUM\_FREE directory devices**

The DMEDIUM\_XXX set of functions can be used to obtain information about a device driver or a medium which is currently driven by this driver, which could not be obtained easily in the past (or not at all).

<b>DMEDIUM_NAME\$</b>	Returns the medium name of the specified device.
<b>DMEDIUM_DRIVE\$</b>	Returns the real device name of the specified file or device. This is the only way to check if the access is done to the device it is intended to be done, as devices may be renamed using <b>RAM_USE</b> , <b>FLP_USE</b> , <b>WIN_USE</b> etc. This function also allows you to discover the "real" device which may be hidden behind "DEV".
<b>DMEDIUM_RDONLY</b>	Returns 1 if the medium is write-protected, otherwise 0. It checks the various possibilities of write protection, even the software write-protection which is possible for hard disks and removable hard disks.
<b>DMEDIUM_REMOVE</b>	Returns 1 if the specified device is a removable hard disk.
<b>DMEDIUM_DENSITY</b>	Returns the density: 1=DD, 2=HD etc. RAM-Disks return -1, as they have no density.
<b>DMEDIUM_FORMAT</b>	Returns the logical format of the medium or partition: 1=QDOS/SMSQ, 2=DOS/TOS.
<b>DMEDIUM_TYPE</b>	Returns information about the physical drive: 0=RAM-Disk, 1=Floppy Disk, 2=Harddisk, 3=CD-ROM.
<b>DMEDIUM_TOTAL</b>	Returns the total number of free sectors (in 512 bytes sectors).
<b>DMEDIUM_FREE</b>	Returns the number of free sectors (in 512 bytes sectors).

These functions should be used on directory devices (RAM, FLP, WIN etc.) only. The parameter passed to these functions can either be a channel number (#channel) or a \directory or \file.

syntax: **DMEDIUM\_xxx ( #channel | \directory | \file )**

- example:
- i. **10 OPEN #3,flp1\_boot**  
**20 PRINT DMEDIUM\_NAME\$(#3)** {what's the name of the disk in flp1\_}  
**30 CLOSE #3**  
**40 PRINT DMEDIUM\_NAME\$(\win1\_)** {returns the name of WIN 1\_}
  - ii. **10 DEV\_USE 1,win1\_** {DEV1\_ accesses WIN1\_}  
**20 OPEN\_NEW #3,dev1\_test** {let's open a new file}  
**30 PRINT DMEDIUM\_DRIVE\$(#3)** {really, it's on WIN1\_}  
**40 CLOSE #3**
  - iii. **PRINT DMEDIUM\_RDONLY(\flp1\_)**
  - iv. **PRINT DMEDIUM\_REMOVE(\win2\_)**
  - v. **PRINT DMEDIUM\_DENSITY(#4)**
  - vi. **PRINT DMEDIUM\_FORMAT(fl2\_)**
  - vii. **PRINT DMEDIUM\_TYPE(dev2\_)**
  - viii. **PRINT DMEDIUM\_TOTAL(#3)**
  - ix. **PRINT DEMDUIM\_FREE(#3)**

## **DO** program

**DO** will execute a series of SBASIC commands from file.

The commands should be 'direct': any lines with line numbers will be merged into the current SBASIC program. The file should not contain any of the following commands. **RUN, LRUN, MRUN, MERGE, SAVE, SAVE\_O, LOAD, STOP, NEW, CLEAR, CONTINUE, RETRY** or **GOTO**.

A **DO** file should be able to invoke SBASIC procedures without harmful effect.

syntax: **DO** *name*

comment: A **DO** file can contain in line clauses:

```
FOR i=1 to 20: PRINT 'This is a DO file'
```

If you try to **RUN** a BASIC program from a **DO** file, then the file will be left open. Likewise, if you put direct commands in a file that is **MERGED**, then the file will be left open.

## **DOS\_DRIVE** directory devices

Unless configured otherwise each DOS device is connected to the root directory of a Windows drive, **DOS1\_** corresponds to C:\, **DOS2\_** to D:\ etc. Using **DOS\_DRIVE** this connection can be changed at runtime. After that the root directory of the DOS device corresponds to the specified directory.

syntax: **DOS\_DRIVE** *device*, *path*\$

example: **DOS\_DRIVE** 1, 'c:\windows\' {connects **DOS1\_** to the Windows directory}

## **DOS\_DRIVE\$** directory devices

Returns the host directory currently connected to a DOS device.

syntax: *dir*\$ = **DOS\_DRIVE\$** (*device*)

## **DOS\_USE** directory devices

**DOS\_USE** allows renaming of the DOS device. **DOS\_USE** without a parameter will reset the name of DOS back to DOS.

syntax: **DOS\_USE** [ *name* ]

example: i. **DOS\_USE** win : **LOAD** win2\_prog {loads 'prog' from **DOS2\_** }  
ii. **DOS\_USE** {and now its name is DOS again}  
iii. **DOS\_USE** ram : **DIR** ram1\_ {displays directory of **DOS1\_**}

## ED EDIT

**ED** is a window based editor for editing SBASIC programs which are already loaded into QPC.

If no line number is given, the first part of the program is listed, otherwise the listing in the window will start at or after the given line number. If no channel number is given, the listing will appear in the normal SBASIC edit window #2. If a window is given, then it must be a **CON**sole window, otherwise a 'bad parameter' error will be returned. The editor will use the current ink and paper colours for normal listing, while using white ink on black paper (or vice versa if the paper is already black or blue) for 'highlighting'. Please avoid using window #0 for the ED.

The editor makes full use of its window. Within its window, it attempts to display complete lines. If these lines are too long to fit within the width of the window, they are 'wrapped around' to the next row in the window: these extra rows are indented to make this 'wrap around' clear. For ease of use, however, the widest possible window should be used.

The **ESC** key is used to return to the SBASIC command mode.

After **ED** is invoked, the cursor in the edit window may be moved using the arrow keys to select the line to be changed. In addition the up and down keys may be used with the **ALT** key (press the **ALT** key and while holding it down, press the up or down key) to scroll the window while keeping the cursor in the same place, and the up and down keys may be used with the **SHIFT** key to scroll through the program a 'page' at a time.

The editor has two modes of operation: insert and overwrite. To change between the two modes use '**CTRL F4**' (press **CTRL** and while holding it down press **F4**). There is no difference between the modes when adding characters to or deleting characters from the end of a line. Within a line, however, insert mode implies that the right hand end of a line will be moved to the right when a character is inserted, and to the left when a character is deleted. No part of the line is moved in overwrite mode. Trailing spaces at the end of a line are removed automatically.

If you press **F10** while the cursor is over a program line, then this line is put (without line number) into the HOTKEY Buffer. It can easily be retrieved by pressing **ALT SPACE** in any program where input is expected. In order to work, the HOTKEY System has to be going (use **HOT\_GO** to activate).

To insert a new line anywhere in the program, press **ENTER**. If there is no room between the line the cursor is on and the next line in the program (e.g. the cursor is on line 100 and the next line is 101) then the **ENTER** key will be ignored, otherwise a space is opened up below the current line, and a new line number is generated. If there is a difference of 20 or more between the current line number and the next line number, the new line number will be 10 on from the current line number, otherwise, the new line number will be half way between them.

If a change is made to a line, the line is highlighted: this indicates that the line has been extracted from the program. The editor will only replace the line in the program when **ENTER** is pressed, the cursor is moved away from the line, or the window is scrolled. If the line is acceptable to SBASIC, it is rewritten without highlighting. If, however, there are syntax errors, the message 'bad line' is sent to window #0, and the line remains highlighted.

While a line is highlighted, **ESC** may be used to restore the original copy of the line, ignoring all changes made to that line.

If a line number is changed, the old line remains and the new line is inserted in the correct place in the program. This can be used to copy single lines from one part of the program to another.

If all the visible characters in a line are deleted, or if all but the line number is deleted, then the line will be deleted from the program. An easier way to delete a line is to press **CTRL** and **ALT** and then the left arrow as well.

The length of lines is limited to about 32766 bytes. Any attempt to edit longer lines may cause undesirable side effects. If the length of a line is increased when it is changed, there may be a brief pause while SBASIC moves its working space.

syntax: *line\_number := numeric\_expression*

**ED** [*channel*,] [*line\_number*]

summary of Edit operations:

<b>TAB</b>	tab right (columns of 8)
<b>SHIFT TAB</b>	tab left (columns of 8)
<b>ENTER</b>	accept line and create a new line
<b>ESC</b>	escape - undo changes or return to SBASIC
up arrow	move cursor up a line
down arrow	move cursor down a line
<b>ALT</b> up arrow	scroll up a line (the screen moves down!)
<b>ALT</b> down arrow	scroll down a line (the screen moves up!)
<b>SHIFT</b> up arrow	scroll up one page
<b>SHIFT</b> down arrow	scroll down one page
left arrow	move cursor left one character
right arrow	move cursor right one character
<b>SHIFT</b> left arrow	move cursor left one word
<b>SHIFT</b> right arrow	move cursor right one word
<b>ALT</b> left arrow	move to start of line
<b>ALT</b> right arrow	move to end of line
<b>CTRL</b> left arrow	delete character to left of cursor
<b>CTRL</b> right arrow	delete character under cursor
<b>CTRL SHIFT</b> left arrow	delete word to left of cursor
<b>CTRL SHIFT</b> right arrow	delete word to right of cursor
<b>CTRL ALT</b> left arrow	delete line to left of cursor
<b>CTRL ALT</b> right arrow	delete line to right of cursor
<b>CTRL</b> down arrow	delete whole line
<b>F9</b> or <b>SHIFT F4</b>	change between overwrite and insert mode
<b>F10</b> or <b>SHIFT F5</b>	when the cursor is over a program line, then this line is put (without line number) into the HOTKEY Buffer. It can easily be retrieved by pressing ALT SPACE in any program where input is expected. In order to work, the HOTKEY System has to be going (use <b>HOT_GO</b> to activate)

comment: **ED** must not be called from within a SBASIC program.

## **EOF** devices

**EOF** is a function which will determine if an end of file condition has been reached on a specified channel. If **EOF** is used without a channel specification then **EOF** will determine if the end of a program's embedded data statements has been reached.

syntax: **EOF** [(*channel*)]

example: i. **IF EOF(#6) THEN STOP**  
ii. **IF EOF THEN PRINT "Out of data"**

## EOFW

This is basically the same as **EOF**, but it will wait if the specified device is not ready whereas **EOF** would return "not complete".

## EPROM\_LOAD

**EPROM\_LOAD** will load an image of a QL EPROM cartridge. Most EPROM cartridges are programmed so that the cartridge may be at any address.

Some require to be at exactly \$C000, the QL ROM port address. The first time the command is used after reset, the EPROM image will be loaded at address \$C000. Subsequent images may be loaded at any address. Fussy EPROM images must, therefore, be loaded first.

An EPROM image file must not be longer than 16 kilobytes.

syntax: **EPROM\_LOAD** *filename*

example: **EPROM\_LOAD flp1\_Qleprom**

comment: To make an EPROM image, put the EPROM cartridge into a QL and turn on. **SBYTES** the image to a suitable file with the magic numbers 49152 (\$C000) for the base address and 16384 (16 kilobytes) for the length. .

**SBYTES flp1\_eprom, 49152, 16384** {Save EPROM image}

In QPC copy the file to your boot diskette or disk and add the **EPROM\_LOAD** statement to your "boot" file.

**EPROM\_LOAD flp1\_eprom** {Load EPROM image}

## ERLIN

### ERLNUM error handling

**ERLIN** is a function that will return the line number where an error has occurred.

**ERLNUM** is a function that will return the error number.

**ERLIN** and **ERLNUM** should only be used as direct commands from the keyboard, or within a **WHEN ERROR** clause.

syntax: **ERLIN**  
**ERLNUM**

example: i. **PRINT ERLIN**  
ii. **last\_error = ERLNUM**

## ERT hotkey system

**ERT** will report the error and stop if its parameter value is negative. If it is not negative then **ERT** will report nothing and continue processing the next statement.

As well as the Hotkey functions. **ERT** can be used with any function, which returns an error code.

syntax: **ERT** *function*

example: i. **ERT HOT\_LOAD ('x', flp1\_program)** {report error if hotkey in use, or file not found}  
ii. **ERT -9** {gives "in use" error}

## EX, EXEC EW, EXEC\_W ET SMSQ/E

**EX** and **EW** will load a sequence of programs and execute them in parallel.

**EX** will return to the command processor after all processes have started execution, **EW** will wait until all the processes have terminated before returning.

**ET** sets up the programs, but returns to SBASIC so that a debugger can be called to trace the execution.

**EXEC** is the same as **EX**, and **EXEC\_W** is the same as **EW**.

syntax: *program* := *device*  
*parameters* := *string\_expression*  
*file* := *filename*, or *channel\_number*

**EX** *program* [ \*,*file* \* ] [;*parameters*]  
**EX** *program* [ \*,*file* \* ] [;*parameters*]

In this case the program in the file 'name' is loaded into the transient program area, the string is pushed onto its stack and execution is initiated.

Finally it is possible for **EX** to open input and output files for a program as well as (or instead of) passing it parameters. If preferred, a SBASIC channel number may be used instead of a filename. A channel used in this way must already be open.

example: The program UC converts a text file to upper case, the command:

```
EX uc, flp1_fred, #1      {load and initiate the program UC, with the file flp1_fred  
                           as its input file, and the output being sent to  
                           window #1.}
```

**EX** is designed to set up filters for processing streams of data.

Within QPC it is possible to have a chain of co-operating jobs engaged in processing the same data in a form of production line. When using a production line of this type, each job performs a well-defined part of the total process. The first job takes the original data and does its part of the process; the partially processed data is then passed on to the next job which carries out its own part of the process; and so the data gradually passes through all the processes. The data is passed from one Job to the next through a 'pipe'. The data itself is termed a 'stream' and the Jobs processing the data are termed 'filters'.

the complete form of the **EX** command is

```
prog_spec := program [ *,file * ] [;parameters  
EX [#channel TO] prog_spec [ * TO prog_spec * ] [TO #channel]
```

Each **TO** separator creates a pipe between Jobs.

All the program names and the parameter string may be names, strings or string expressions. The significance of the filenames is, to some extent, program dependent; but there are two general rules which should be used by all filters:

the primary input of a filter is the pipe from the previous Job in the chain (if it exists), or else the first data file,

the primary output of a filter is the pipe to the next job in the chain (if it exists) or else the last data file.

Many filters will have only two I/O channels: the primary input and the primary output.

If the parameters of **EX** start with '#channel **TO**', then the corresponding SBASIC channel will be closed (if it was already open) and a new channel opened as a pipe to the first program. Any data sent to this channel (e.g. by **PRINT**ing to it) will be processed by the chain of Jobs. When the channel is **CLOSE**d, the chain of Jobs will be removed from QPC.

If the parameters of **EX** end with '**TO** #channel', then the corresponding SBASIC channel will be closed (if it was already open) and a new channel opened as a pipe from the last program. Any data passing through the chain of Jobs will arrive in this channel and may be read (e.g. by **INPUT**ing from it). When all the data has passed, the Jobs will remove themselves and any further attempt to take input from this channel will get an 'end of file' error. The **EOF** function may be used to test for this.

#### Example of Filter Processing

As an example of filter processing, the programs UC to convert a file to upper case, LNO to line number a file, and PAGE to split a file onto pages with an optional heading are all chained to process a single file:

**EX uc, fred TO lno TO page,ser; 'File fred at '&date\$**

The filter UC takes the file 'fred' and after converting it to upper case, passes through a pipe to LNO. LNO adds line numbers to each line and passes the file down a pipe to PAGE. In its turn, PAGE splits the file onto pages with the heading (including in this case the date) at the top of each page, before sending the file to the SER port. Note that the file fred itself is not modified; the modified versions are purely transient.



## EXTRAS

**EXTRAS** will output to the specified or default channel, a list of commands and functions available to SBASIC. The original TK2 command did only output the additional commands (therefore the name), but in SMSQ/E all commands are shown.

syntax: **EXTRAS** [*#channel*]

example: i. **EXTRAS #3** {output list to #3}  
ii. **EXTRAS** {output list to default channel #1}

## FEXP\$ conversion functions

**FEXP\$** will convert a value to a string representing the value in exponent form.

The form has an optional sign and one digit before the decimal point, and 'ndp' digits after the decimal point. The exponent is in the form of 'E' followed by a sign followed by 2 digits. The field must be at least 7 greater than ndp.

syntax: *field* := *numeric\_expression* {length of returned string}  
*ndp* := *numeric\_expression* {number of decimal places}

**FEXP\$** (*value*, *field*, *ndp*)

example: **PRINT FEXP\$ (1234.56,12,4)** {will print ' 1.2346E+03'}

## FDEC\$

### IDEC\$, CDEC\$ conversion functions

These routines convert a value into a decimal number in a string. The number of decimal places represented is fixed, and the exponent form of floating point number is not used.

The three routines are very similar. **FDEC\$** converts the value as it is, whereas **IDEC\$** assumes that the value given is an integral representation in units of the least significant digit displayed. **CDEC\$** is the currency conversion which is similar to **IDEC\$**, except that there are commas every 3 digits.

syntax: *field* := *numeric\_expression* {length of returned string}  
*ndp* := *numeric\_expression* {number of decimal places}

**FDEC\$** (*value*, *field*, *ndp*)

**IDEC\$** (*value*, *field*, *ndp*)

**CDEC\$** (*value*, *field*, *ndp*)

example: i. **PRINT FDEC\$ (1234.56,9,2)** {will print ' 1234.56'}  
ii. **PRINT IDEC\$ (123456,9,2)** {will print ' 1234.56'}  
iii. **PRINT CDEC\$ (123456,9,2)** {will print ' 1,234.56'}

comment: If the number of characters is not large enough to hold the value, the string is filled with '\*'. The value should be between  $-2^{31}$  and  $2^{31}$  (-2,000,000,000 to +2,000,000,000) for **IDEC\$** and **CDEC\$**, whereas for **FDEC\$** the value multiplied by  $10^{ndp}$  should be in this range.

## FILL graphics

**FILL** will turn *graphics fill* on or off. **FILL** will fill any non-re-entrant shape drawn with the *graphics* or *turtle graphics* procedures as the shape is being drawn. Re-entrant shapes must be split into smaller non-re-entrant shapes.

When you have finished filling, **FILL 0** should be called.

syntax: `switch:= numeric_expression` {range 0..1}

**FILL** [*channel*,] *switch*

example: i. **FILL 1:LINE 10,10 TO 50,50 TO 30,90 TO 10,10:FILL 0**  
{will draw a filled triangle}  
ii. **FILL 1:CIRCLE 50,50,20:FILL 0**  
{will draw a filled circle}

## FILL\$ string arrays

**FILL\$** is a function which will return a string of a specified length filled with a repetition of one or two characters.

syntax: **FILL\$** (*string\_expression*, *numeric\_expression*)

The string expression supplied to **FILL\$** must be either one or two characters long.

example: i. **PRINT FILL\$("a",5)** {will print aaaaa}  
ii. **PRINT FILL\$("oO",7)** {will print oOoOoOo}  
iii. **LET a\$ = a\$ & FILL\$(" ",10)**

## FLASH windows

**FLASH** turns the flash state on and off. On a QL **FLASH** is only effective in *low resolution* mode, in QPC it has no effect whatsoever. **FLASH** will be effective in the window attached to the specified or default channel.

syntax: `switch:= numeric_expression` {range 0..1}

**FLASH** [*channel*,] *switch*

where: switch = 0 will turn the flash off  
switch = 1 will turn the flash on

example: **100 PRINT "A ";**  
**110 FLASH 1**  
**120 PRINT "flashing ";**  
**130 FLASH 0**  
**140 PRINT "word"**

**warning:** Writing over part of a flashing character can produce spurious results and should be avoided.

## **FLEN, FTYP, FDAT FXTRA, FNAME\$ FUPDT, FBKDT, FVERS** file information

There are six functions to extract information from the header of a file.

**FLEN** will return the length of the file.

**FTYP** will return the file type. The file type is, 0 for ordinary files, 1 for executable programs, and 2 for relocatable machine code.

**FDAT** will return the files data space. Only valid results will be obtained from executable programs.

**FXTRA** will return the file extra information.

**FNAME\$** will return the filename.

**FUPDT** will return the files update date

**FBKDT** will return the backup date from the file.

**FVERS** will return the files version number.

If a file is being extended, the file length can be found by using the **FPOS** function to find the current file position. (If necessary the file pointer can be set to the end of file by the command **GET \#n 999999**.)

syntax: **FLEN (#channel)**  
**FTYP (#channel)**  
**FDAT (#channel)**  
**FXTRA (#channel)**  
**FNAME\$ (#channel)**  
**FUPDT (#channel)**

example: **PRINT FLEN (#3)** {print the length of the file open on channel #3}

comment: The file information functions can also be used with implicit channels. E.g.

**PRINT FLEN (fred)** {print the length of file fred}

## **FLP\_DENSITY** directory devices

The SMSQ/E format routines will usually attempt to format a disk to the highest density possible for a medium. The **FLP\_DENSITY** command is used to specify a particular recording density during format. The density codes are "S" for single sided (double density), "D" for double density and "H" for high density.

syntax: **FLP\_DENSITY [ S | D | H ]**

example: i. **FLP\_DENSITY S** {Set the default format to single sided}  
ii. **FLP\_DENSITY H** {Set the default format to high density}  
iii. **FLP\_DENSITY** {Reset to automatic density selection}

comment: The same code letters may be added (after a \*) to the end of the medium name to force a particular density format. (For compatibility with older drivers, if the code letter is omitted after the \*, single sided format is assumed.

i. **FORMAT 'FLP1\_Disk23'** {Format at highest density or as specified by  
FLP\_DENSITY}  
ii. **FORMAT 'FLP1\_Disk24\*'** {Format single sided}  
iii. **FORMAT 'FLP1\_Disk25\*S'** {Format single sided}  
iv. **FORMAT 'FLP1\_Disk25\*D'** {Format double sided, double density}

## **FLP\_SEC**

### **FLP\_START, FLP\_STEP** directory devices

These commands are supplied for compatibility reasons. QPC has no influence over how the Windows disk driver works, therefore these commands are ignored.

### **FLP\_STEP** directory devices

This command is supplied for compatibility reasons. QPC has no influence over how the Windows disk driver works, therefore this command is ignored.

### **FLP\_TRACK** directory devices

FLP\_TRACK sets the number of tracks to be formatted on a floppy disk.

syntax: *tracks := numeric\_expression*

**FLP\_TRACK** *tracks*

example: **100 FLP\_TRACK 40** {set number of tracks to 40}  
**110 FORMAT flp1\_small** {only format 40 tracks of disk}

### **FLP\_USE** directory devices

FLP\_USE allows renaming of the FLP device. FLP\_USE without a parameter will reset the name of FLP back to FLP.

syntax: **FLP\_USE** [*name*]

example: i. **FLP\_USE dos : LOAD dos2\_prog** {loads 'prog' from FLP2\_}  
ii. **FLP\_USE** {and now its name is FLP again}  
iii. **FLP\_USE win : DIR win1\_** {displays directory of FLP1\_}

### **FLUSH** directory devices

SMSQ/E directory device drivers maintain as much of a file in RAM as possible. A power failure or other accident could result in a file being left in an incomplete state. The **FLUSH** command will ensure that a file is updated without closing it. Closing a file will always cause the file to be flushed.

syntax: **FLUSH** *#channel*

## FOPEN, FOP\_IN FOP\_NEW, FOP\_OVER FOP\_DIR devices

This is a set of functions for opening files. These functions differ from the **OPEN** procedures in two ways. Firstly, if a file system error occurs (e.g. 'not found' or 'already exists') these functions return the error code and continue. Secondly the functions may be used to find a vacant hole in the channel table: if successful they return the channel number.

When called with two parameters, these functions return the value zero for successful completion, or a negative error code.

The #channel parameter is optional: if it is not given, the functions will search the channel table for a vacant entry, and, if the open is successful, the channel number will be returned. Note that error codes are always negative, and channel numbers are positive.

syntax:    **FOPEN** ( [#channel,] name)            {open a file for read/write}  
           **FOP\_IN** ( [#channel,] name)            {open a file for input only}  
           **FOP\_NEW** ( [#channel,] name)        {open a new file}  
           **FOP\_OVER** ( [#channel,] name)      {open a new file, if it exists it is overwritten}  
           **FOP\_DIR** ( [#channel,] name)        {open a directory}

example: i. A file may be opened for read only with an optional extension using the following code:

```

ferr=FOP_IN (#3,name$&'_ASM')        :REMark try to open _ASM file
IF ferr=-7: ferr=FOP_IN (#3,name$)    :REMark ERR.NF, try no _ASM

ii.  outch = FOP_NEW (fred)            :REMark open fred
if outch < 0: REPORT outch: STOP      :REMark ... oops
PRINT #outch, 'This is file Fred'
CLOSE #outch

```

## FOR END FOR repetition

The **FOR** statement allows a group of SBASIC statements to be repeated a controlled number of times. The **FOR** statement can be used in both a long and a short form.

**NEXT** and **END FOR** can be used together within the same **FOR** loop to provide a *loop epilogue*, i.e. a group of SBASIC statements which will not be executed if a loop is exited via an **EXIT** statement but which will be executed if the **FOR** loop terminated normally.

```

define:        for_item:=            | numeric_expression
                                     | numeric_exp TO numeric_exp
                                     | numeric_exp TO numeric_exp STEP numeric_exp

              for_list. =            for_item *[, for_item] *

```

**SHORT:** The **FOR** statement is followed on the same logical line by a sequence of SBASIC statements. The sequence of statements is then repeatedly executed under the control of the **FOR** statement. When the **FOR** statement is exhausted, processing continues on the next line. The **FOR** statement does not require its terminating **NEXT** or **END FOR**. Single line **FOR** loops must not be nested.

syntax:        **FOR** variable = for\_list : statement \*[: statement]\*

example:      i. **FOR** i = 1, 2, 3, 4 TO 7 STEP 2 : PRINT i  
               ii. **FOR** element = first TO last : LET buffer (element) = 0

**LONG:** The **FOR** statement is the last statement on the line. Subsequent lines contain a series of SBASIC statements terminated by an **END FOR** statement. The statements enclosed between the **FOR** statement and the **END FOR** are processed under the control of the **FOR** statement.

syntax:     **FOR** *variable* = *for\_list*  
                  *statements*  
           **END FOR** *variable*

example:    **100 INPUT "data please" ! x**  
              **110 LET factorial = 1**  
              **120 FOR value = x TO 1 STEP -1**  
              **130 LET factorial = factorial \* value**  
              **140 PRINT x !!!! factorial**  
              **150 IF factorial > IE20 THEN**  
              **160 PRINT "Very Large number"**  
              **170 EXIT value**  
              **180 END IF**  
              **190 END FOR value**

### **FORMAT** directory devices

**FORMAT** will format and make ready for use the directory device contained in the specified drive.

syntax:     **FORMAT** [*channel*,] *device*

Device specifies the drive (physical or virtual) to be used for formatting and the identifier part of the specification is used as the medium or volume name for floppy disks, The number of sectors (512 bytes) for RAM disks, or the size in megabytes for WIN drives. **FORMAT** will write the number of good sectors and the total number of sectors available on the directory device on the default or on the specified channel.

A RAM disk may be removed by giving either a null name or zero sectors.

example:    i. **FORMAT flp1\_data\_disk**  
              ii. **FORMAT ram2\_20**                    {Format RAM2\_ to 10K bytes}  
              iii. **FORMAT win1\_40**                    {Format WIN1\_ to 40M bytes}  
              iv. **FORMAT ram1\_0**                     {Remove RAM1\_}

**FORMAT** can be used to reinitialise a used directory device. However all data contained on that device will be lost.

## **FPOS** devices

FPOS will return the current file position for the specified channel.

The file pointer can be set by the commands **BGET**, **BPUT**, **GET** or **PUT** with no items to be got or put. If an attempt is made to put the file pointer beyond the end of file, the file pointer will be set to the end of file and no error will be returned. Note that setting the file pointer does not mean that the required part of the file is actually in a buffer, but that the required part of the file is being fetched. In this way, it is possible for an application to control prefetch of parts of a file where the device driver is capable of prefetching.

syntax: **FPOS** (*#channel*)

example: **10 PUT #4\102,value1,value2**  
**20 ptr = FPOS (#4)** {set 'ptr' to 114 (=102+6+6)}

The file pointer can be set by the commands **BGET**, **BPUT**, **GET** or **PUT** with no items to be got or put. If an attempt is made to put the file pointer beyond the end of file, the file pointer will be set to the end of file and no error will be returned. Note that setting the file pointer does not mean that the required part of the file is actually in a buffer, but that the required part of the file is being fetched. In this way, it is possible for an application to control prefetch of parts of a file where the device driver is capable of prefetching.

## **FREE\_MEM** memory management

The function **FREE\_MEM** will return the amount of free memory available in the 'common heap'.

syntax: **FREE\_MEM**

example: **PRINT FREE\_MEM**

## **FTEST** devices

The function **FTEST** is used to determine the status of a file or device. It opens a file for input only and immediately closes it. If the file exists it will either return the value 0 or -9 (in use error code), if it does not exist, it will return -7 (not found error code). Other possible returns are -11 (bad name), -15 (bad parameter), -3 (out of memory) or -6 (no room in the channel table).

syntax: **FTEST** (*name*)

example: The function can be used to check that a file does not exist:

```
IF FTEST (file$) <> -7: PRINT 'File '; file$; ' exists'
```

## GET

### PUT unformatted I/O

It is possible to put or get values in their internal form. The **PRINT** and **INPUT** commands of SBASIC handle formatted I/O, whereas the direct I/O routines **GET** and **PUT** handle unformatted I/O. For example, if the value 1.5 is **PRINT**ed the byte values 49 ('1'), 46 ('.') and 53 ('5') are sent to the output channel. Internally, however, the number 1.5 is represented by 6 bytes (as are all other floating point numbers). These six bytes have the value 08 01 60 00 00 00 (in hexadecimal). If the value is **PUT**, these 6 bytes are sent to the output channel.

The internal form of an integer is 2 bytes (most significant byte first). The internal form of a floating point number is a 2 byte exponent to base 2 (offset by hex 81F), followed by a 4 byte mantissa, normalised so that the most significant bits (bits 31 and 30) are different. The internal form of a string is a 2 byte positive integer, holding the number of characters in the string, followed by the characters.

**GET** gets data in internal format from the specified or default channel. **PUT** puts data in internal format into the specified or default channel. For **GET**, each item must be an integer, floating point, or string variable. Each item should match the type of the next data item from the channel. For **PUT**, the type of data put into the channel, is the type of the item in the parameter list.

syntax:   **GET** #channel [position] , items                    {get internal format data from a file}  
          **PUT** #channel [position] , items                    {put internal format data onto a file}

example:  **10 fpoint=54**  
          **20 wally%=42: salary=78000: name\$='Smith'**  
          **30 PUT #3\3fpoint, wally%, salary, name\$**

position the file, open on #3, to the 54th byte, and put 2 bytes (integer 42), 6 bytes (floating point 78000), 2 bytes (integer 5) and the 5 characters 'Smith'. Fpoint will be set to 69 (54+2+6+2+5).

comment: For variables or array elements the type is self evident, while for expressions there are some tricks which can be used to force the type:

.... **+0**            will force floating point type;  
.... **&"**            will force string type;  
.... **||0**            will force integer type.

```
xyz$='ab258.z'  
...  
PUT #3\37,xyz$(3 to 5)||0
```

will position the file opened on channel #3 to the 37th byte and then will put the integer 258 on the file in the form of 2 bytes (value 1 and 2, i.e. 1\*256+2).

## GOSUB

For compatibility with other BASICs, SBASIC supports the **GOSUB** statement. **GOSUB** transfers processing to the specified line number; a **RETurn** statement will transfer processing back to the statement following **GOSUB**.

The line number specification can be an expression.

syntax:   **GOSUB** line\_number

example:  i. **GOSUB 100**  
          ii. **GOSUB 4\*select\_variable**

comment: The control structures available in SBASIC make the **GOSUB** statement redundant.

## GOTO

For compatibility with other BASICs, SBASIC supports the **GOTO** statement. **GOTO** will unconditionally transfer processing to the statement number specified. The statement number specification can be an expression.

syntax: **GOTO** *line\_number*

example: i. **GOTO program\_start**  
ii. **GOTO 9999**

comment: The control structures available in SBASIC make the **GOTO** statement redundant.

## HEX

### HEX\$ conversion functions

**HEX** will convert the supplied hexadecimal string into a value. The 'digits' '0' to '9' 'A' to 'F' and 'a' to 'f' have their conventional meanings. **HEX** will return an error if it encounters a non-recognised character.

**HEX\$** will return a string of sufficient length to represent the value of the specified number of bits of the least significant end of the value rounded up to the nearest multiple of 4.

syntax: *number\_of\_bits := numeric\_expression*

**HEX** (*hexadecimal\_string*)  
**HEX\$** (*value, number\_of\_bits*)

example: **PRINT HEX ("1AF6")** {will output 6902}  
**PRINT HEX\$ (32673, 16)** {will output "7FA1"}

## HGET

### HPUT formatted I/O

**HGET** and **HPUT** will read and write the first parts of a file header from the specified or default channel. Both commands accept up to 5 parameters, which are of the type floating point. The first parameter is the file length (long), followed by the access byte (byte), followed by the file type (byte), then comes the dataspace (long) and finally the extra-information (long).

syntax: *length := numeric\_expression*  
*access := numeric\_expression*  
*type := numeric\_expression*  
*dataspace := numeric\_expression*  
*extra := numeric\_expression*

**HGET** [#channel,] *length, access, type, dataspace, extra*  
**HPUT** [#channel,] *length, access, type, dataspace, extra*

example: **OPEN#3,flp1\_file**  
**HGET#3,length, access, type, space, extra**  
**HPUT#3,length, access,1,1024,extra**  
**CLOSE#3**

converts a file into an executable file with 1kByte dataspace.

## **HOT\_CHP, HOT\_CHP1 HOT\_RES, HOT\_RES1 hotkey system**

**HOT\_CHP** and **HOT\_RES** will load a program into either the common heap, or the resident procedure area, making it into an Executable Thing. This Thing can then be executed very quickly when the Hotkey is pressed.

For frequently used programs, these two functions set up an Executable Thing to be executed using a Hotkey. If you want to add a program temporarily that you may wish to remove later, **HOT\_CHP** should be used. Otherwise **HOT\_RES** should be used, as this will often give faster execution. If the resident procedure area is not available, then **HOT\_RES** will use the common heap instead.

**HOT\_CHP1** and **HOT\_RES1** are the same as **HOT\_CHP** and **HOT\_RES**, except that they set up a Wake Hotkey. When you press the Hotkey, if there is already a Job of the same name executing, then it will be Picked and Woken, otherwise a new copy will be executed.

Jobs may be identified by a name, which is normally the program name. This name is to be found in the base area of a standard program. It is possible, however, to specify a different name for a Job when you set up the Hotkey.

To persuade the HOTKEY system to execute a Job with Unlocked windows, you need to add the single parameter "U" to the function parameter list. To provide a "Guardian" window to preserve the whole area used by the Job, you need to add the single parameter "G" to the function parameter list. Optionally, you may follow this by the window area (size, position) of the Guardian window as four numbers. Any attempt by a program to open or redefine a window outside its Guardian will fail. To execute a Job so that it will be frozen when its windows are buried, you add the single parameter "F" to the parameter list. To prevent the program from taking too much memory, you add the parameter "P", optionally followed by the amount of memory (in kilo bytes) the program may take.

Note that "U", "G", "P" or "F" can be used after the "I" option for impure programs which modify their own code.

The functions will return one of the following error codes:

- 0 - No error
- 2 - No job (file is not executable)
- 3 - Out of memory
- 7 - Not found (file could not be found)
- 9 - In use (Hotkey is already being used for some other operation)
- 12- Bad name (bad file name)

syntax: *key* := *character\_string* {single character string in the range 32 to 191}  
*params* := *string* {list of parameters for individual programs}  
*options* := [ I, ] U  
          | G [ *width, height, xorg, yorg* ]  
          | P [ *memory* ]  
          | F

**HOT\_CHP** (*key, filename* [;*params*] [,*jobname*] [,*options*] )  
**HOT\_RES** (*key, filename* [;*params*] [,*jobname*] [,*options*] )  
**HOT\_CHP1** (*key, filename* [;*params*] [,*jobname* | !*wakename*] [,*options*] )  
**HOT\_RES1** (*key, filename* [;*params*] [,*jobname* | !*wakename*] [,*options*] )

- example:
- i. **ERT HOT\_RES ('t', qtyp)** {set up Qtyp using default drive}
  - ii. **ERT HOT\_RES1('t', flp1\_qtyp)** {just one copy on the specified drive}
  - iii. **ERT HOT\_RES ('t', ' flp1\_qtyp')** {or all between apostrophes}
  - iv. **ERT HOT\_CHP ('t', qtyp)** {or so we can **HOT\_REMV** it}
  - v. **ERT HOT\_RES ('=', qtyp\_e, 'Editor Qtyp')** {specifying a job name}
  - vi. **ERT HOT\_RES (c, capslock, u)** {set up unlocked "capslock" on **ALT C**}
  - vii. **ERT HOT\_RES (x, terminal, g)** {set up Terminal on ALT X with Guardian window covering the whole Screen}
  - viii **ERT HOT\_RES (r, rubbish, i, g, 124, 22, 388, 0)** {setup " rubbish", an impure program which requires a Guardian of 124x22 pixels with its origin at 388x0}

comment: Alternatively we can set up Qtyp in a loop checking the error return for a not found:

```

10 REPEAT lqtyp
20 herr = HOT_RES ('t', ' qtyp')           {try loading Qtyp}
30 IF NOT herr; EXIT lqtyp                 {..OK}
40 IF herr =-7                             {not found?}
50 INPUT #0, 'Put Qtyp disk in drive 1 and press ENTER'
60 NEXT lqtyp                               {try again}
70 END IF
80 PRINT #0, 'Loading Qtyp';: ERT herr     {give up}
90 END REPEAT lqtyp

```

## HOT\_CMD hotkey system

**HOT\_CMD** allows one or more commands to be sent directly to the command console of SBASIC. This is similar to **HOT\_KEY**, but when the Hotkey is pressed, SBASIC is picked to the top, and each command is sent to the command console, followed by a newline (ENTER).

This can be used to load and run SBASIC programs, or to execute simple command sequences.

The function will return one of the following error codes:

```

0 - No error
-9 - In use          (Hotkey is already being used for some other operation)

```

syntax: *key := character\_string* {single character string in the range 32 to 191}

**HOT\_CMD (key, string \*[, string]\*)**

- example:
- i. **ERT HOT\_CMD (m, 'LRUN flp1\_mandel')** {LRUN a BASIC program}
  - ii. **ERT HOT\_CMD (d, wdir)** {directory listing}
  - iii. **ERT HOT\_CMD (r, 'INPUT "Run> ";prg\$', 'LRUN prg\$')**  
 {prompt for name of, and LRUN a program, note the use of quotes within the string delimited by apostrophes}

## HOT\_DO hotkey system

**HOT\_DO** allows a previously defined Hotkey to be activated from SBASIC. The Hotkey system interprets the **HOT\_DO** command as if the Hotkey had been pressed.

syntax: *key := character\_string* {single character string in the range 32 to 191}

**HOT\_DO key | name**

- example:
- 10 ERT HOP\_CHP (q, Quill, p)** {set Quill on ALT-Q}
  - 20 HOT\_DO 'Quill'** {start Quill, without pressing ALT-Q}

## **HOT\_GETSTUFF\$ hotkey system**

**HOT\_STUFF\$** returns the current or previous contents of the stuffer buffer, e.g. set by **HOT\_STUFF**.

If no parameter is given or index is 0, the current contents of the stuffer buffer will be returned. If -1 is passed the previous contents will be returned, another call with -1 will return the contents before that etc.

Syntax: **HOT\_GETSTUFF\$** [ index ]

## **HOT\_GO**

### **HOT\_STOP hotkey system**

**HOT\_GO** and **HOT\_STOP** will start and stop the Hotkey system.

The Hotkey system is designed to remain dormant until all resident extensions have been loaded. It is then activated by the **HOT\_GO** command.

If, at any time, you wish to add more resident extensions to QPC, you can remove the HOTKEY Job using the **RJOB** command or the **HOT\_STOP** command.

Neither **HOT\_GO** nor **HOT\_STOP** have any parameters.

syntax: **HOT\_GO** {start HOTKEY Job}  
**HOT\_STOP** {stop HOTKEY Job}

## **HOT\_KEY hotkey system**

The **HOT\_KEY** function is used to set up Hotkeys to copy strings of keystrokes into the current keyboard queue.

When the appropriate Hotkey is pressed, each of the strings is sent to the keyboard queue, separated by a new line (Enter) character.

You can specify as many lines as you like. If you want one or more new lines after the last **HOT\_KEY** string, you should put one or more empty (null) strings at the end of the list.

The function will return one of the following error codes:

0 - No error  
-9 - In use (Hotkey is already being used for some other operation)

syntax: *key := character\_string* {single character string in the range 32 to 191}

**HOT\_KEY** (*key, string* \*[, *string* ]\*)

example: i. **ERT HOT\_KEY ("s", "Dear Sir", "", "")** {two new lines at end}  
ii. **ERT HOT\_KEY ("e", "Yours sincerely", "", "", " Joe Bloggs" )**  
iii. **ERT HOT\_KEY ("p", CHR\$(232) & "PD", "NP" )** {print from abacus}

comment: **HOT\_KEY** is functionally identical to the **ALTKEY** command.

## **HOT\_LIST hotkey system**

**HOT\_LIST** will send to the specified or default channel , the current list of Hotkey assignments.

syntax: **HOT\_LIST** [ #channel ]  
**HOT\_LIST** filename

example: i. **HOT\_LIST** {list Hotkeys to #1}  
ii. **HOT\_LIST ram1\_keys** {list to file "ram1\_keys"}

## HOT\_LOAD

### HOT\_LOAD1 hotkey system

**HOT\_LOAD** will set up a Hotkey to load and execute a program from disk, that is not required frequently enough to justify making it resident. This is similar to the **HOT\_RES** and **HOT\_CHP**, but the program is not loaded until required. It follows, of course, that the disk with the program file must be available at the time you press the Hotkey.

**HOT\_LOAD1** is the same as **HOT\_LOAD**, except that it sets up a Wake Hotkey. When you press the Hotkey, if there is already a Job of the same name executing, then it will be Picked and Woken, otherwise a new copy will be executed.

Jobs may be identified by a name, which is normally the program name. This name is to be found in the base area of a standard program. It is possible, however, to specify a different name for a Job when you set up the Hotkey.

To persuade the HOTKEY system to execute a Job with Unlocked windows, you need to add the single parameter "U" to the function parameter list. To provide a "Guardian" window to preserve the whole area used by the Job, you need to add the single parameter "G" to the function parameter list. Optionally, you may follow this by the window area (size, position) of the Guardian window as four numbers. Any attempt by a program to open or redefine a window outside its Guardian will fail. To execute a Job so that it will be frozen when its windows are buried, you add the single parameter "F" to the parameter list. To prevent the program from taking too much memory, you add the parameter "P", optionally followed by the amount of memory (in kilo bytes) the program may take. Note that "U", "G", "P" or "F" can be used after the "I" option for impure programs which modify their own code.

The function will return one of the following error codes:

0 - No error  
-9 - In use (Hotkey is already being used for some other operation)

syntax: *key* := *character\_string* {single character string in the range 32 to 191}  
*params* := *string* {list of parameters for individual programs}  
*options* := [ I, ] U  
          | G [ *width, height, xorg, yorg* ]  
          | P [ *memory* ]  
          | F

**HOT\_LOAD** (*key*, *filename* [;*params*] [,*jobname*] [,*options*] )

**HOT\_LOAD** (*key*, *filename* [;*params*] [,*jobname* | !*wakename*] [,*options*] )

example: **ERT HOT\_LOAD (f, qtyp\_file)** {Load and execute Qtyp\_File on ALT F}

### HOT\_NAME\$ hotkey system

The **HOT\_NAME\$** function will return the name associated with the supplied Hotkey.

The function will return a null (empty) string if the Hotkey is not defined.

syntax: *key* := *character\_string* {single character string in the range 32 to 191}

**HOT\_NAME\$** ( *key* )

example: **PRINT HOT\_NAME\$ ('a')** {display the name associated with the key **ALT-a**}

## HOT\_OFF

### HOT\_SET hotkey system

HOT\_OFF and HOT\_SET will turn off and on, or change individual Hotkey operations.

The functions will return one of the following error codes:

- 0 - No error
- 7 - Not found (Old key or name cannot be found)
- 9 - In use (New key is already in use, **HOT\_SET** only)

syntax: *key* := *character\_string* {single character string in the range 32 to 191}  
*newkey* := *key*  
*oldkey* := *key*

**HOT\_OFF** ( *key* | *name* )  
**HOT\_SET** ( *key* | *name* )  
**HOT\_SET** ( *newkey*, *oldkey* | *name* )

example: i. **ERT HOT\_OFF ('c')** {switch off **ALT-c**}  
ii. **ERT HOT\_SET ('h','r')** {**ALT-h** now does what **ALT-r** used to}

comment: The name is the program or Thing name for execute and Pick type Hotkeys, or the string or command for **HOT\_KEY** and **HOT\_CMD** Hotkeys.

## HOT\_PICK hotkey system

The **HOT\_PICK** function sets up a Hotkey to Pick a Job of a particular name, so that you may work with it.

The Job name is usually embedded at the start of the program file. For pure programs set up by **HOT\_RES** and **HOT\_CHP**, this name is replaced if you specify a Job name. For Psion programs, which do not have a name at the start, **HOT\_CHP**, etc, will set the Job name to be the same as the program file name.

You do not need to specify the complete Job name, just the first word in the name. This is useful for programs which add extra information after the program name (e.g. the Files menu of QPAC 2, which adds a directory name after the Job name). If there is more than one Job with a matching name, each Job will be Picked in turn.

The function will return one of the following error codes:

- 0 - No error
- 9 - In use (Hotkey is already being used for some other operation)

syntax: *key* := *character\_string* {single character string in the range 32 to 191}

**HOT\_PICK** ( *key*, *jobname* )

example: i. **ERT HOT\_PICK ( '1' , Quill )** {pick Quill on ALT 1}  
ii. **ERT HOT\_PICK ( '2' , Abacus )** {pick Abacus on ALT 2}

## **HOT\_REMV hotkey system**

The **HOT\_REMV** function will turn the Hotkey off, and remove the definition as well.

If the Hotkey was set up using **HOT\_CHP**, the Executable Thing and any Jobs using it are removed.

**HOT\_REMV** will usually need to be used to remove a Hotkey definition before re-using the particular Hotkey. Unless **HOT\_KEY** or **HOT\_CMD** are being used to re-define a string or command respectively.

syntax: *key := character\_string* {single character string in the range 32 to 191}

**HOT\_REMV ( key | name )**

example: 10 ERT **HOT\_CHP** (q, Quill, p) {Quill on ALT Q}  
20 ERT **HOT\_OFF** (q) {ALT Q turned off}  
30 ERT **HOT\_SET** (q) {ALT Q back on}  
40 ERT **HOT\_SET** (z,q) {Quill now on ALT Z}  
50 ERT **HOT\_REMV** (Quill) {Quill gone completely}

## **HOT\_STUFF hotkey system**

**HOT\_STUFF** will place the supplied strings into the Stuffer Buffer. The first string is put in the buffer first, immediately followed by the second string (if present).

The next time you press **ALT SPACE** the strings will be copied into the current keyboard queue as if you had just typed them.

syntax: **HOT\_STUFF** *string1* [ , *string2* ]

example: i. **HOT\_STUFF** DATE\$ {place time and date into Stuffer Buffer}  
ii. **HOT\_STUFF** "Dear Sir", CHR\$(13)&CHR\$(13)  
{place 'Dear Sir' and the Enter key twice}

## **HOT\_THING**

### **HOT\_THING1 hotkey system**

**HOT\_THING** will set up a Hotkey to execute an Executable Thing. The Thing need not have been created at the time the Hotkey is set up. QPAC 2 is implemented as a collection of (mostly) Executable Things. The **HOT\_RES** and **HOT\_CHP** functions create an Executable Thing for each program set up on a Hotkey.

The HOTKEY system 2 is a non-executable Thing.

**HOT\_THING1** is the same as **HOT\_THING**, except that it sets up a Wake Hotkey. When you press the Hotkey, if there is already a Job of the same name executing, then it will be Picked and Woken, otherwise a new copy will be executed.

Jobs may be identified by a name, which is normally the program name. This name is to be found in the base area of a standard program. It is possible, however, to specify a different name for a Job when you set up the Hotkey.

The function will return one of the following error codes:

0 - No error  
-9 - In use (Hotkey is already being used for some other operation)

syntax: *key* := *character\_string* {single character string in the range 32 to 191}  
*params* := *string* {list of parameters for individual programs}

**HOT\_THING** (*key*, *thingname* [;*params*] [,*jobname*] )  
**HOT\_THING1** (*key*, *thingname* [;*params*] [,*jobname* | !*wakename*] )

example: **ERT HOT\_THING (' f , Files )** {Execute QPAC 2 Files Menu on ALT F}

## **HOT\_TYPE** hotkey system

The **HOT\_TYPE** function will return the type of action associated with the supplied Hotkey.

The types returned by **HOT\_TYPE** are

-8	last line recall
-6	stuff keyboard queue with previous stuffer string
-4	stuff keyboard queue with current stuffer string
-2	stuff keyboard queue with defined string
0	pick SBASIC and stuff command
2	do code
4/5	execute Thing
6	execute file
8	pick Job
10/11	wake or execute Thing
12	wake / execute file
-7	not defined

syntax: *key* := *character\_string* {single character string in the range 32 to 191}

**HOT\_TYPE** ( *key* )

example: **PRINT HOT\_TYPE ('c' )** {display the Hotkey type of the key **ALT-c**}

## **HOT\_WAKE hotkey system**

**HOT\_WAKE** is a variation of **HOT\_PICK** which will set up a Hotkey to Wake a Job when Picking it. Hotkeys set up by **HOT\_WAKE** go a little further than this: if there is no Job of the required name executing at the time you press the Hotkey, then, if there is an Executable Thing of the same name, this will be Executed.

Even if a program does not recognize a Wake Event, this Hotkey can still be used to Pick or Execute the program.

This is most useful for accessing Executable Things that you will only ever want one copy executing at a time. It is, for example, pointless having more than one copy of the QPAC 2 EXEC menu. If you set up a **HOT\_WAKE** Hotkey for EXEC, the first time you use it you will Execute the EXEC Thing. Until you remove the EXEC Job, every time you use this Hotkey, the EXEC menu will be Picked and Woken.

The function will return one of the following error codes:

0 - No error  
-9 - In use (Hotkey is already being used for some other operation)

syntax: *key* := *character\_string* {single character string in the range 32 to 191}  
*params* := *string* {list of parameters for individual programs}

**HOT\_WAKE** (*key*, *thingname* [;*params*] [,*jobname* | ! *wakename* ] )

example: **ERT HOT\_WAKE ('x', 'Exec')**

comment: For normal programs, the best way of using this function is to create an Executable Thing using one of the **HOT\_RES** or **HOT\_CHP** functions, and then define a second Hotkey to Wake the Thing. Quite a neat way of doing this is to use a lower case Hotkey to Wake the program, and the corresponding upper case Hotkey to create a new copy.

**ERT HOT\_RES (' D', ' QD')** {Set up QD to Execute on **ALT D**}  
**ERT HOT\_WAKE (' d', ' QD')** {Set up to Wake or Execute on **ALT d**}

## **IF THEN ELSE END IF**

The **IF** statement allows conditions to be tested and the outcome of that test to control subsequent program flow.

The **IF** statement can be used in both a long and a short form:

**SHORT:** The **THEN** keyword is followed on the same logical line by a sequence of SBASIC keyword. This sequence of SBASIC statements may contain an **ELSE** keyword. If the expression in the **IF** statement is true (evaluates to be non-zero), then the statements between the **THEN** and the **ELSE** keywords are processed. If the condition is false (evaluates to be zero) then the statements between the **ELSE** and the end of the line are processed.

If the sequence of SBASIC statements does not contain an **ELSE** keyword and if the expression in the **IF** statement is true, then the statements between the **THEN** keyword and the end of the line are processed. If the expression is false then processing continues at the next line.

syntax:        *statements:= statement \*[: statement]\**  
  
              **IF** *expression* **THEN** *statements* [**ELSE** *statements*]

example:    i. **IF** *a=32* **THEN** **PRINT** "Limit" : **ELSE** **PRINT** "OK"  
              ii. **IF** *test >maximum* **THEN** **LET** *maximum = test*  
              iii. **IF** "1"+1=2 **THEN** **PRINT** "coercion OK"

**LONG 1:** The **THEN** keyword is the last entry on the logical line. A sequence of SBASIC statements is written following the **IF** statements. The sequence is terminated by the **END IF** statement. The sequence of SBASIC statements is executed if the Expression contained in the **IF** statement evaluates to be non zero. The **ELSE** keyword and second sequence of SBASIC statements are optional.

**LONG 2:** The **THEN** keyword is the last entry on the logical line. A Sequence of SBASIC statements follows on subsequent lines, terminated by the **ELSE** keyword. If the expression contained in the **IF** statement evaluates to be non zero then this first sequence of SBASIC statements is processed. After the **ELSE** keyword a second sequence of SBASIC statements is entered, terminated by the **END IF** keyword. If the expression evaluated by the **IF** statement is zero then this second sequence of SBASIC statements is processed.

syntax:        **IF** *expression* **THEN**  
                  *statements*  
                  [**ELSE**  
                  *statements*]  
                  **END IF**

example:    100 **LET** *Limit =10*  
              110 **INPUT** "Type in a number" ! *number*  
              120 **IF** *number > limit* **THEN**  
              130 **PRINT** "Range error"  
              140 **ELSE**  
              150 **PRINT** "Inside Limit"  
              160 **END IF**

**comment:** In all three forms of the **IF** statement the **THEN** is optional. In the short form it must be replaced by a colon to distinguish the end of the **IF** and the start of the next statement. In the long form it can be removed completely.

**nesting:** **IF** statements may be nested as deeply as the user requires (subject to available memory). However, confusion may arise as to which **ELSE**, **END IF** etc matches which **IF**. SBASIC will match nested **ELSE** statements etc to the closest **IF** statement, for example:

```
100 IF a = b THEN  
110 IF c = d THEN  
120 PRINT "error"  
130 ELSE  
140 PRINT "no error"  
150 END IF  
160 ELSE  
170 PRINT "not checked"  
180 END IF
```

The **ELSE** at line 130 is matched to the second **IF**. The **ELSE** at line 160 is matched with the first **IF** (at line 100).

## INK windows

This sets the current ink colour, i.e. the colour in which the output is written. **INK** will be effective for the window attached to the specified or default *channel*.

syntax: **INK** [*channel*,] *colour*

- example:
- i. **INK 5**
  - ii. **INK 6,2**
  - iii. **INK #2,255**

## INKEY\$

**INKEY\$** is a function which returns a single character input from either the specified or default *channel*.

An optional timeout can be specified which can wait for a specified time before returning, can return immediately or can wait forever. If no parameter is specified then **INKEY\$** will return immediately.

syntax: **INKEY\$** [(*channel*)  
          [(*channel*, *time*)  
          [(*time*)]

where: *time* = 1..32767      {wait for specified number of frames.  
                                  In the UK 50 Frames = 1 Second  
                                  In the US 60 Frames = 1 Second}  
*time* = -1                    {wait forever}  
*time* = 0                    {return immediately}

- example:
- i. **PRINT INKEY\$**                    {input from the default channel}
  - ii. **PRINT INKEY\$(#4)**                {input from channel 4}
  - iii. **PRINT INKEY\$(50)**                {wait for 50 frames then return anyway}
  - iv. **PRINT INKEY\$(0)**                {return immediately (poll the keyboard)}
  - v. **PRINT INKEY\$(#3,100)**            {wait for 100 frames for an input from channel 3 then return anyway}

comment: If no character was available when **INKEY\$** times out, then a Null (**CHR\$(0)**) will be returned.

## INPUT

**INPUT** allows data to be entered into a SBASIC program directly from the PC's keyboard by the user. SBASIC halts the program until the specified amount of data has been input; the program will then continue. Each item of data must be terminated by the **ENTER** key.

**INPUT** will input data from either the specified or the default *channel*.

If input is required from a particular console channel the cursor for the window connected to that channel will appear and start to flash.

```
syntax:  separator:= |!  
          |,  
          |\  
          |;  
          | TO
```

*prompt:= [channel,] expression separator*

**INPUT** [*prompt*] [*channel*] *variable* \*[,*variable*]\*

example: i. **INPUT ("Last guess "& guess & "New guess?") ! guess**  
ii. **INPUT "What is your guess?"; guess**  
iii. **100 INPUT "array size?" ! Limit**  
**110 DIM array(limit-1)**  
**120 FOR element = 0 to Limit-1**  
**130 INPUT ("data for element" & element) array(element)**  
**140 END FOR element**  
**150 PRINT array**

## IO\_PRIORITY

**IO\_PRIORITY** sets the priority of the IO retry operations. In effect, this sets a limit on the time spent by the scheduler retrying IO operations.

A priority of one sets the IO retry scheduling policy to the same as QDOS, thus giving a similar level of response but with a higher crude performance.

syntax: *level := numeric expression*

**IO\_PRIORITY** *level*

example: i. **IO\_PRIORITY 1** {QDOS levels of response, higher crude performance}  
ii. **IO\_PRIORITY 2** {QDOS levels of performance, better response under load}  
iii. **IO\_PRIORITY 10** {Much better response under load, degraded performance}  
iv. **IO\_PRIORITY 1000** {Maximum response, the performance depends on the number of jobs waiting for input.}

## INSTR operator

**INSTR** is an operator which will determine if a given substring is contained within a specified string. If the string is found then the substring's position is returned. If the string is not found then **INSTR** returns zero.

Zero can be interpreted as false, i.e. the substring was not contained in the given string. A non zero value, the substrings position, can be interpreted as true, i.e. the substring was contained in the specified string.

syntax: *string\_expression* **INSTR** *string expression*

example: i. **PRINT "a" INSTR "cat"** {will print 2}  
ii. **PRINT "CAT" INSTR "concatenate"** {will print 4}  
iii. **PRINT "x" INSTR "eggs"** {will print 0}

## INSTR\_CASE

**INSTR\_CASE** allows the type of string comparison to be used by **INSTR** to be set as either case independent (default), or case dependent.

syntax: **INSTR\_CASE 0 | 1**

example: i. **INSTR\_CASE 0** {INSTR is now case independent. (SuperBASIC compatible)}  
ii. **INSTR\_CASE 1** {INSTR now does direct byte by byte comparisons}

comment: The **internal INSTR\_CASE** flag is cleared on **NEW**, **LOAD**, **MERGE** and **RUN**.

## INT maths functions

**INT** will return the integer part of the specified floating point expression.

syntax: **INT** (*numeric\_expression*)

example: i. **PRINT INT(X)**  
ii. **PRINT INT(3.141592654/2)**

## JOBS SMSQ/E

**JOBS** is a command to list to the window attached to the specified or default channel, all the Jobs running in QPC at the time. If there are more Jobs in the machine than can be listed in the output window, the procedure will freeze the screen (CTRL F5) when it is full. The procedure may fail if Jobs are removed from the QL while the procedure is listing them.

syntax: **JOBS** [#*channel*] {list current Jobs}  
**JOBS** \ *device* {list Jobs to 'device'}

The following information is given for each Job

- The Job number
- The Job tag
- The Job's owner Job number
- A flag 'S' if the Job is suspended
- The Job priority
- The Job (or program) name.

## **JOB\$, NXJOB**

### **OJOB, PJOB SMSQ/E**

**JOB\$, NXJOB, OJOB, and PJOB** are Job status functions provided to enable an SBASIC program to scan the Job tree and carry out complex Job control procedures.

**JOB\$** will return as a string the name of the Job.

**NXJOB** is a rather complex function. The first parameter is the id of the Job currently being examined, the second is the id of the Job at the top of the tree. If the first id passed to **NXJOB** is the last Job owned, directly or indirectly, by the 'top Job', then **NXJOB** will return the value 0, otherwise it will return the id of the next Job in the tree.

**OJOB** will return Job identifier of the owner of the Job.

**PJOB** will return priority of the job.

syntax: *job\_identifier* := | *job\_number* , *tag\_number*  
| *job\_number* + (*tag\_number* \* 65536)  
*id* := *job\_identifier*

**JOB\$** (*id* | *name*)  
**NXJOB** (*id* | *name*)  
**OJOB** (*id* | *name*)  
**PJOB** (*id* | *name* , *top\_job\_id*)

example: i. **PRINT JOB\$ (3,8)** {will output name of Job}  
ii. **PRINT OJOB (demon)** {will output the id of the owner of Job 'demon'}  
iii. **PRINT PJOB (2,1)** {will output the priority of the Job}

comment: Job 0 always exists and owns directly or indirectly all other Jobs in QPC. Thus a scan starting with *id* = 0 and top Job *id* = 0 will scan all Jobs in QPC.

It is possible that, during a scan of the tree, a Job may terminate. As a precaution against this happening, the Job status functions return the following values if called with an invalid Job *id*:

**PJOB=0 OJOB=0 JOB\$="" NXJOB=-1**

## **JOB\_NAME SMSQ/E**

**JOB\_NAME** can be used to give a name to an SBASIC Job. It may appear anywhere within a program and may be used to reset the name whenever required. This command has no effect on compiled BASIC programs or Job 0.

syntax: **JOB\_NAME** *string\_expression*

example: i. **JOB\_NAME Killer** {sets the Job name to "Killer"}  
ii. **JOB\_NAME "My little Job"** {sets the Job name to "My little Job"}

## KBD\_TABLE

**KBD\_TABLE** will set the keyboard layout to be used.

syntax: *lang := language\_code | registration*

**KBD\_TABLE lang**

example: i. **KBD\_TABLE GB** {keyboard table set to English}  
ii. **KBD\_TABLE 33** {keyboard table set to French}

comment: Private keyboard tables may also be loaded.

**i= RESPR (512): LBYTES "kt",i: KBD\_TABLE i** {keyboard table set to table in "kt"}

For compatibility with older drivers, a "private" keyboard table loaded in this way should not be prefaced by flag word.

## KEYROW

**KEYROW** is a function which looks at the instantaneous state of a row of keys (the table below shows how the keys are mapped onto a matrix of 8 rows by 8 columns). **KEYROW** takes one parameter, which must be an integer in the range 0 to 7: this number selects which row is to be looked at. The value returned by **KEYROW** is an integer between 0 and 255 which gives a binary representation indicating which keys have been depressed in the selected row.

Since **KEYROW** is used as an alternative to the normal keyboard input mechanism using **INKEY\$** or **INPUT**, any character in the keyboard type-ahead buffer are cleared by **KEYROW**: thus key depressions which have been made before a call to **KEYROW** will not be read by a subsequent **INKEY\$** or **INPUT**.

Note that multiple key depressions can cause surprising results. In particular, if three keys at the corner of a rectangle in the matrix are depressed simultaneously, it will appear as if the key at the fourth corner has also been depressed. The three special keys **CTRL**, **SHIFT** and **ALT** are an exception to this rule, and do not interact with other keys in this way.

syntax: *row:= numeric\_expression* {range 0..7}

**KEYROW (row)**

example: **100 REMark run this program and press a few keys**  
**110 REPeat loop**  
**120 CURSOR 0,0**  
**130 FOR row = 0 to 7**  
**140 PRINT row !!! KEYROW(row) ;" "**  
**150 END FOR row**  
**160 END REPeat loop**

## KEYBOARD MATRIX

	COLUMN							
ROW	1	2	4	8	16	32	64	128
7	SHIFT	CTRL	ALT	X	V	/	N	,
6	8	2	6	Q	E	0	T	U
5	9	W	I	TAB	R	-	Y	O
4	L	3	H	1	A	P	D	J
3	[	CAPS	K	S	F	=	G	;
2	]	Z	.	C	B	`	M	'
1	C/R	left	up	ESC	right		SPC	down
0	F4	F1	5	F2	F3	F5	4	7

## LANGUAGE

### LANGUAGE\$

**LANGUAGE** and **LANGUAGE\$** will return the currently set language, or to find the language that would be used if a particular language were requested. They can also be used to convert the language (dialling code ) into a car registration and vice versa.

Language Code	Car Registration	Language and Country
33	F	French (in France)
44	GB	English (in England)
49	D	German (in Germany)
1	USA	USA (in USA)

**LANGUAGE** will return the language code, and **LANGUAGE\$** will return the car registration.

syntax: *lang := language\_code | registration*

**LANGUAGE** [ (*lang*) ]  
**LANGUAGE\$** [ (*lang*) ]

- example:
- i. **PRINT LANGUAGE** {returns the current language}
  - ii. **PRINT LANGUAGE\$** {the car registration of the current language}
  - iii. **PRINT LANGUAGE (F)** {the language corresponding to F}
  - iv. **PRINT LANGUAGE\$ (45)** {the car registration corresponding to 4}
  - v. **PRINT LANGUAGE (977)** {the language that would be used for Nepal}

## LANG\_USE

**LANG\_USE** will set the language used by the system messages. This sets the Operating System language word, and then scans the language dependent module list selecting modules and filling in the message table.

A language may be specified either by an international dialling code or an international car registration code. These codes may be modified by the addition of a digit where a country has more than one language.

Language Code	Car Registration	Language and Country
33	F	French (in France)
44	GB	English (in England)
49	D	German (in Germany)
1	USA	USA (in USA)

syntax: *lang := language\_code | registration*

**LANG\_USE** *lang*

- example:
- i. **LANG\_USE 33** {set language to French}
  - ii. **LANG\_USE D** {set language to German}
  - iii. **LANG\_USE 'g'&'b'** {set language to English}

**warning:** if you assign a value to a variable, then you will not be able to use that variable name to specify the car registration letters.

**D=33: LANG\_USE D** {set language to French (dialling code 33)  
rather than German (car registration D)}

## **LBYTES** devices, directory devices

**LBYTES** will load a data file into memory at the specified start address.

If a channel number of an open channel is supplied in place of a filename, then **LBYTES** will attempt to load the file from the channel.

syntax: *start\_address := numeric\_expression*  
*device := filename | channel*

**LBYTES** *device, start\_address*

example: i. **LBYTES flp1\_screen, SCR\_BASE**  
          {load a screen image}  
          ii. **LBYTES win1\_program, start\_address**  
              {load a program at a specified address}  
          iii. **10 OPEN#5,flp1\_data**                    {open a channel}  
               **20 address = ALCHP(FLEN(#5))**        {get file length and allocate space}  
               **30 LBYTES#5,address**                {load the file}  
               **40 CLOSE#5**                            {close the channel}

## **LEN** string arrays

**LEN** is a function which will return the length of the specified string *expression*.

syntax: **LEN**(*string\_expression*)

example: i. **PRINT LEN("LEN will find the length of this string")**  
          ii. **PRINT LEN(output\_string\$)**

## **LET**

**LET** starts a SBASIC assignment statement. The use of the **LET** keyword is optional. The assignment may be used for both string and numeric assignments. SBASIC will automatically convert unsuitable data types to a suitable form wherever possible.

syntax: [**LET**] *variable = expression*

example: i. **LET a = 1 + 2**  
          ii. **LET a\$ = "12345"**  
          iii. **LET a\$ = 6789**  
          iv. **b\$ = test\_data**

## LINE

### LINE\_R

**LINE** allows a straight line to be drawn between two points in the *window* attached to the default or specified channel. The ends of the line are specified using the *graphics coordinate system*.

Multiple lines can be drawn with a single **LINE** command.

The normal specification requires specifying the two end points for a line. These end points can be specified either in absolute coordinates (relative to the *graphics origin*) or in relative coordinates (relative to the *graphics cursor*). If the first point is omitted then a line is drawn from the graphics cursor to the specified point. If the second point is omitted then the graphics cursor is moved but no line is drawn.

**LINE** will always draw with absolute coordinates, i.e. relative to the *graphics origin*, while **LINE\_R** will always draw relative to the graphics cursor.

syntax: *x:= numeric\_expression*  
*y:= numeric\_expression*  
*point:= x,y*

*parameter\_2:=* | **TO point** (1)  
| *,point TO point* (2)

*parameter\_1:=* | **TO point, angle** (1)  
| **TO point** (2)  
| *point* (3)

**LINE** [*channel*,] *parameter\_1* \*[, *parameter\_2*]\*

**LINE\_R** [*channel*,] *parameter\_1* \*[,*parameter\_2*]\*

Where (1) will draw from the specified point to the next specified point  
(2) will draw from the the last point plotted to the specified point  
(3) will move to the specified point - no line will be drawn

example: i. **LINE 0,0 TO 0, 50 TO 50,0 TO 50,0 TO 0,0** {a square}  
ii. **LINE TO 0.75, 0.5** {a line}  
iii. **LINE 25,25** {move the graphics cursor}

## LIST

**LIST** allows a SBASIC line or group of lines to be listed on a specific or default *channel*.

syntax: *line:=* | *line\_number TO line\_number* (1)  
| *line\_number TO* (2)  
| *TO line\_number* (3)  
| *line\_number* (4)  
| (5)

**LIST** [*channel*,] *line*\*[,*line*]\*

where (1) will list from the specified line to the specified line  
(2) will list from the specified line to the end  
(3) will list from the start to the specified line  
(4) will list the specified line  
(5) will list the whole program

example: i. **LIST** {list all lines}  
ii. **LIST 10 TO 300** {list lines 10 to 300}  
iii. **LIST 12,20,50** {list lines 12,20 and 50 only}

If **LIST** output is directed to a channel opened as a printer channel then **LIST** will provide hard copy.

## LOAD

### QLOAD devices, directory devices

**LOAD** will load a SBASIC program from any QPC device. **LOAD** automatically performs a **NEW** before loading another program, and so any previously loaded program will be cleared by **LOAD**.

**QLOAD** will load an SBASIC program which has been saved by **QSAVE** or **QSAVE\_O** and has a **\_SAV** at the end of the filename.

If a line input during a load has incorrect SBASIC syntax, the word **MISTAKE** is inserted between the line number and the body of the line. Upon execution, a line of this sort will generate an error

syntax: **LOAD** *device*  
**QLOAD** *device*

example: i. **LOAD "flp2\_test\_program"**  
ii. **LOAD ram1\_guess**  
iii. **QLOAD flp1\_program**  
iv. **LOAD ser1\_e**  
v. **QLOAD dev1\_program\_sav**

## LN

### LOG10 maths functions

**LN** will return the natural logarithm of the specified argument. **LOG10** will return the common logarithm. There is no upper limit on the parameter other than the maximum number the computer can store.

syntax: **LOG10** (*numeric\_expression*) {range greater than zero}  
**LN** (*numeric\_expression*) {range greater than zero}

example: i. **PRINT LOG10(20)**  
ii. **PRINT LN(3.141592654)**

## LOCAL functions and procedures

**LOCAL** allows *identifiers* to be defined to be **LOCAL** to a *function or procedure*. Local identifiers only exist within the function or procedure in which they are defined, or in procedures and functions called from the function or procedure in which they are defined.

They are lost when the function or procedure terminates. Local identifiers are independent of similarly named identifiers outside the defining function or procedure. *Arrays* can be defined to be local by dimensioning them within the **LOCAL** statement.

The **LOCAL** statement must precede the first executable statement in the function or procedure in which it is used.

syntax: **LOCAL** *identifier* \*[, *identifier*]\*

example: i. **LOCAL a,b,c(10,10)**  
ii. **LOCAL temp\_data**

comment: Defining variables to be **LOCAL** allows variable names to be used within functions and procedures without corrupting meaningful variables of the same name outside the function or procedure.

## **LRESPR SMSQ/E**

**LRESPR** opens the load file and finds the length of the file, then reserves space for the file in the resident procedure area before loading the file. Finally a **CALL** is made to the start of the file.

syntax: **LRESPR** *name*

example: **LRESPR win1\_basic\_ext** {load and call the SBASIC extensions  
Win1\_basic\_ext}

## **LRUN**

### **QLRUN devices, directory devices**

**LRUN** will load and run a SBASIC *program* from a specified *device*. **LRUN** will perform **NEW** before loading another program and so any previously stored SBASIC program will be cleared by **LRUN**.

**QLRUN** will load an SBASIC program which has been saved by **QSAVE** or **QSAVE\_O** and has a **\_SAV** at the end of the filename.

If a line input during a loading has incorrect SBASIC syntax, the word **MISTAKE** is inserted between the line number and the body of the line. Upon execution, a line of this sort will generate an error.

syntax: **LRUN** *device*  
**QLRUN** *device*

example: i. **LRUN flp2\_TEST**  
ii. **LRUN ram1\_game**  
iii. **QLRUN win1\_applications\_editor**

## **MACHINE SMSQ/E**

**MACHINE** will return the machine type that SMSQ/E is running on

syntax: **MACHINE**

example: **PRINT MACHINE**

comment: **MACHINE** will return 30 for QPC.

## MAKE\_DIR

### FMAKE\_DIR directory devices

The command **MAKE\_DIR** is used to create a new subdirectory. It takes one parameter: the subdirectory filename.

**FMAKE\_DIR** is a function to perform the same operation as **MAKE\_DIR**. But will return a value of zero for no error, or a negative number if an error occurs.

Error code	<b>-7</b>	<b>not found</b>	Medium or drive is not available
	<b>-8</b>	<b>already exists</b>	Already directory/file of that name
	<b>-9</b>	<b>in use</b>	Already directory/file of that name
	<b>-15</b>	<b>bad parameter</b>	Device cannot handle subdirectories

syntax: **MAKE\_DIR** *filename*  
*ferr* = **FMAKE\_DIR** (*filename*)

example: i. **MAKE\_DIR** flp2\_letters\_  
ii. **error\_code** = **FMAKE\_DIR** ("dev1\_files\_")

comment: If there are any files which, by virtue of their names, would belong in the directory being made, then these files will be transferred to the new directory, even if they are open.

To remove a subdirectory, firstly delete its contents then delete the subdirectory itself. **COPY** and **WCOPY** deal only with files at the specified directory level. Subdirectories can also be applied to RAM disks.

## MERGE

### QMERGE devices, directory devices

**MERGE** will load a *file* from the specified *device* and interpret it as a SBASIC program. If the new file contains a line number which doesn't appear in the program already in QPC then the line will be added. If the new file contains a replacement line for one that already exists then the line will be replaced. All other old program lines are left undisturbed.

**QMERGE** will load an SBASIC program which has been saved by **QSAVE** or **QSAVE\_O** and has a **\_SAV** at the end of the filename.

If a line input during a **MERGE** has incorrect SBASIC syntax, the word **MISTAKE** is inserted between the line number and the body of the line. Upon execution, a line of this sort will generate an error.

syntax: **MERGE** *device*  
**QMERGE** *device*

example: i. **MERGE** win1\_overlay\_program  
ii. **QMERGE** flp1\_new\_data

## MOD operators

**MOD** is an operator which gives the modulus, or remainder; when one integer is divided by another.

syntax: *numeric\_expression* **MOD** *numeric\_expression*

example: i. **PRINT 5 MOD 2** {will print 1}  
ii. **PRINT 5 MOD 3** {will print 2}

## MODE windows

**MODE** sets the resolution of the screen and the number of solid colours which it can display. **MODE** will clear all *windows* currently on the screen, but will preserve their position and shape. Changing to low resolution mode (8 colour) will set the minimum character size to 2,0.

**MODE** only has any effect in QL colour mode.

syntax: **MODE** *numeric\_expression*

where: 8 or 256 will select low resolution mode  
4 or 512 will select high resolution mode

example: i. **MODE 256**  
ii. **MODE 4**

## MOUSE\_SPEED

**MOUSE\_SPEED** adjusts the mouse acceleration and wake up factor for the specified or default channel. From QPC2 version 2 on the acceleration is of no more use as the mouse position is adapted from Windows. The wakeup factor however is still valid and ranges from 1 to 9 with 1 being the most sensitive one.

syntax: *acceleration := numeric\_expression*  
*wakeup := numeric\_expression*

**MOUSE\_SPEED** [#channel,] *acceleration, wakeup*

## MOUSE\_STUFF

**MOUSE\_STUFF** adjusts the string that is stuffed into the keyboard queue of the specified or default if the middle mouse button is pressed. The string cannot be longer than 2 characters, but this is enough to trigger any hotkey, which can in turn do almost everything.

syntax: **MOUSE\_STUFF** [#channel,] *string*

example: i. **MOUSE\_STUFF '.'** {Generates a dot if middle mouse button is pressed}  
ii. **MOUSE\_STUFF CHR\$(255)&'.'** {Generates hotkey Alt +}

## MOVE turtle graphics

**MOVE** will move the graphics turtle in the *window* attached to the default or specified *channel* a specified distance in the current direction. The direction can be specified using the **TURN** and **TURNT0** commands. The graphics scale factor is used in determining how far the turtle actually moves. Specifying a negative distance will move the turtle backwards.

The turtle is moved in the window attached to the specified or default *channel*.

syntax: *distance:= numeric\_expression*

**MOVE** [*channel*,] *distance*

example: i. **MOVE #2,20** {move the turtle in channel 2 20 units forwards}  
ii. **MOVE -50** {move the turtle in the default channel 50 units backwards}

## **MRUN**

### **QMRUN** devices, directory devices

**MRUN** will interpret a *file* as a SBASIC program and merge it with the currently loaded program.

If used as *direct command* **MRUN** will run the new program from the start. If used as a program statement **MRUN** will continue processing on the line following **MRUN**.

**QMRUN** will load an SBASIC program which has been saved by **QSAVE** or **QSAVE\_O** and has a **\_SAV** at the end of the filename.

If a line input during a merge has incorrect SBASIC syntax, the word **MISTAKE** is inserted between the line number and the body of the line. Upon execution, a line of this sort will generate an error.

syntax:   **MRUN** *device*  
          **QMRUN** *device*

example:  i. **MRUN** flp1\_chain\_program  
          ii. **QMRUN** flp2\_new\_data

## **NET** network

**NET** originally allowed the network station number to be set. The **NET** device is not available in QPC. This keyword is provided for compatibility purposes only.

## **NEW**

**NEW** will clear out the old *program*, *variables* and *channels* other than 0,1 and 2.

syntax:   **NEW**

example:  **NEW**

## **NEXT** repetition

**NEXT** is used to terminate, or create a loop *epilogue* in, **REPeat** and **FOR** loops.

syntax: **NEXT** *identifier*

The identifier must match that of the loop which the **NEXT** is to control

- example:
- i. **10 REMark this loop must repeat forever**  
**11 REPeat infinite\_ loop**  
**12 PRINT "still looping"**  
**13 NEXT infinite\_ loop**
  - ii. **10 REMark this loop will repeat 20 times**  
**11 LET limit = 20**  
**12 FOR index=1 TO Limit**  
**13 PRINT index**  
**14 NEXT index**
  - iii. **10 REMark this Loop will tell you when a 30 is found**  
**11 REPeat Loop**  
**12 LET number = RND(1 TO 100)**  
**13 IF number = 30 THEN NEXT Loop**  
**14 PRINT number; " is 30"**  
**15 EXIT LOOP**  
**16 END REPeat loop**

**in REPeat:** If **NEXT** is used inside a **REPeat - END REPeat** construct it will force processing to continue at the statement following the matching **REPeat** statement.

**In FOR:** The **NEXT** statement can be used to repeat the **FOR** loop with the control variable set at its next value. If the **FOR** loop is exhausted then processing will continue at the statement following the **NEXT**; otherwise processing will continue at the statement after the **FOR**.

## **ON...GOTO** **ON...GOSUB**

To provide compatibility with other BASICs, SBASIC supports the **ON GOTO** and **ON GOSUB** statements. These statements allow a variable to select from a list of possible *line numbers* a line to process in a **GOTO** or **GOSUB** statement. If too few line numbers are specified in the list then an error is generated.

syntax: **ON** *variable* **GOTO** *expression* \*[, *expression*]\*  
**ON** *variable* **GOSUB** *expression* \*[, *expression*]\*

- example:
- i. **ON x GOTO 10, 20, 30, 40**
  - ii. **ON select\_variable GOSUB 1000,2000,3000,4000**

comment: **SELe**ct can be used to replace these two BASIC commands.

## **OPEN, OPEN\_IN OPEN\_OVER, OPEN\_DIR**

### **OPEN\_NEW** devices, directory devices

**OPEN** allows the user to link a logical *channel* to a physical QPC *device* for I/O purposes.

**OPEN\_OVER** will open a new directory device file overwriting the old file if it already exists.

**OPEN\_DIR** will open the directory of a directory device.

If the channel is to a directory device then the directory device file can be an existing file or a new file. In which case **OPEN\_IN** will open an already existing directory device file for input and **OPEN\_NEW** will create a new directory device file for output.

syntax: *channel:= # numeric\_expression*

**OPEN** *channel, device*  
**OPEN\_IN** *channel, device*  
**OPEN\_OVER** *channel, device*  
**OPEN\_DIR** *channel, device*  
**OPEN\_NEW** *channel, device*

example: i. **OPEN #5, f\_name\$**  
ii. **OPEN\_IN #9, "flp1\_filename"**  
    {open file mdvl\_file\_name}  
iii. **OPEN\_NEW #7, win1\_datafile**  
    {open file mdvl\_datafile}  
iv. **OPEN #6, con\_10x20a20x2032**  
    {Open channel 6 to the console device creating a window size 10x20 pixels at  
    position 20,20 with a 32 byte keyboard type ahead buffer.}  
v. **OPEN #8, dev1\_read\_write\_file.**

## **OUTLN windows**

**OUTLN** is used when writing SBASIC programs for the Pointer Interface, it signals that the window is managed. Only managed windows with managed primaries may be used for pointer input: SBASIC's primary window is usually #0.

The three optional parameters default to zero, but you can specify the move key, the shadow widths or both if you wish. The shadow will appear to the right or bottom if xshad or yshad are positive. The move key will discard the current window contents if it is zero, or move them to the new position if it is set to 1 (you must keep the x and y sizes the same for this to work).

If you set the outline of a secondary window, then the area underneath it will be saved, and restored when the outline is set again: this allows you to implement pull-down windows without having to do the saves and restores yourself.

If **OUTLN** is used without parameters, then it will declare the smallest area which outlines all windows currently opened for the job, to be the outline for that job, without changing the primary window.

syntax: *xsize* := *numeric\_expression*  
*ysize* := *numeric\_expression*  
*xorg* := *numeric\_expression*  
*yorg* := *numeric\_expression*  
*xshad* := *numeric\_expression*  
*yshad* := *numeric\_expression*  
*move* := *numeric\_expression*

**OUTLN** [*#channel*, ] *xsize*, *ysize*, *xorg*, *yorg* [ , *xshad*, *yshad* ] [ , *move* ]  
**OUTLN**

example: i. **OUTLN #4, 150,100,30,20,2,2** {set outline of #4 to a window 150 x 100, at 30, 20 with a 2 pixel shading}  
ii. **OUTLN 512,256** {set outline of #0 to 512 x 256}

## **OVER** windows

**OVER** selects the type of over printing required in the window attached to the specified or default channel. The selected type remains in effect until the next use of **OVER**.

syntax: *switch*:= *numeric\_expression* {range -1..1}

**OVER** [*channel*,] *switch*

where *switch* = 0 - print ink on *strip*  
*switch* = 1 - print in ink on transparent *strip*  
*switch* = -1 - XORs the data on the screen

example: i. **OVER 1** {set "overprinting"}  
ii. **10 REMark Shadow Writing**  
**11 PAPER 7 : INK 0 : OVER 1 : CLS**  
**12 CSIZE 3,1**  
**13 FOR i = 0 TO 10**  
**14 CURSOR i,i**  
**15 IF i=10 THEN INK 2**  
**16 PRINT "Shadow"**  
**17 END FOR i**

## PALETTE\_QL

### PALETTE\_8 graphics device 2

PALETTE\_QL allows you to change the displayed colours of the standard QL compatible colours 0 to 7.

PALETTE\_8 allows you to change the displayed colours of the 256 colour (8 bit) mode.

On systems like QPC that do not have a true palette map, palette map changes do not affect the information already drawn on screen.

syntax: *start* := *numeric\_expression*  
*true\_colour* := *numeric\_expression* {in the range 0 to 16,777,215}

**PALETTE\_QL** *start* \* , *true\_colour* \* {up to 8 true colours}  
**PALETTE\_8** *start* \* , *true\_colour* \* {up to 256 true colours}

example: i. **100 red = 255 \* 65536**  
**110 green = 255 \* 256**  
**120 blue = 255**  
**130 magenta = 255 \* 65536 + 255**  
**140 yellow = 255 \* 65536 + 255 \* 256**  
**150 cyan = 255 \* 256 + 255**  
**160 PALETTE\_QL 0,0,yellow,cyan,green,magenta,red,blue**

comment: There is a practical reason for changing the QL palette map entries. Many programs define some of the colours displayed as "white-colour" on a 4 colour QL display, white-red appears as green. White-red, however, is really cyan, not green. As a result, many QL mode 4 programs take on rainbow hues when displayed on a 256, 65536 or full colour display.

This can be "fixed" by redefining the colours so that colour 2 is a bright crimson and colour 4 is a bright sea green. This will ensure that colour 2 + colour 4 = colour 7. We also need to ensure that colour 0 = colour 1, colour 2 = colour 3, etc.

**600 crimson = 255 \* 65536 + 100 : REMark crimson is red + a bit of blue**  
**610 sea = 255 \* 256 + 155 : REMark: sea green is green + the rest of blue**  
**620 white = crimson + sea**  
**630 PALETTE\_QL 0, 0, 0, crimson, crimson, sea, sea, white, white**  
**: REMark set 8 colours**

## PAN windows

**PAN** the entire current window the specified number of pixels to the left or the right. **PAPER** is scrolled in to fill the clear area.

An optional second parameter can be specified which will allow only part of the screen to be panned.

syntax: *distance:= numeric\_expression*  
*part:= numeric\_expression*

**PAN** [*channel*,] *distance* [, *part*]

where        part = 0 - whole screen (or no parameter)  
              part = 3 - whole of the cursor line  
              part = 4 - right end of cursor line including the cursor position

If the expression evaluates to a positive value then the contents of the screen will be shifted to the right.

example: i. **PAN #2,50**        {pan left 50 pixels}  
          ii. **PAN -100**        {pan right 100 pixels}  
          iii. **PAN 50.3**        {pan the whole of the current cursor line 50 pixels to the right}

**warning:** If *stipples* are being used or the screen is in low resolution mode then, to maintain the stipple pattern, the screen must be panned in multiples of two pixels.

## PAPER windows

**PAPER** sets a new paper colour (ie. the colour which will be used by **CLS**, **PAN**, **SCROLL**, etc). The selected paper colour remains in effect until the next use of **PAPER**. **PAPER** will also set the **STRIP** colour

**PAPER** will change the paper colour in the *window* attached to the specified or default *channel*.

syntax: **PAPER** [*channel*,] *colour*

example: i. **PAPER #3,7**        {White paper on channel 3}  
          ii. **PAPER 7,2**        {White and red stipple}  
          iii. **PAPER 255**        {Black and white stipple}  
          iv. **10 REMark Show colours and stipples**  
              **11 FOR colour = 0 TO 7**  
              **12 FOR contrast = 0 TO 7**  
              **13 FOR stipple = 0 TO 3**  
              **14 PAPER colour, contrast, stipple**  
              **15 SCROLL 6**  
              **16 END FOR stipple**  
              **17 END FOR contrast**  
              **18 END FOR colour**

## **PARNAME\$** procedures

The function **PARNAME\$** when used in a procedure will return the name of the parameter number.

syntax: *parameter\_number := numeric\_expression*

**PARNAM\$** (*parameter\_number*)

example: **10 pname fred, joe, 'mary'**

```
****
70 DEF PROC pname (n1,n2,n3)
80 PRINT PARNAM$(1), PARNAM$(2), PARNAM$(3)
90 END DEF pname
```

would print 'fred joe ' (the expression has no name).

## **PARSTR\$** procedures

The function **PARSTR\$** when used in a procedure will if parameter 'name' is a string, return the value of the string, else find the name of the parameter number.

syntax: *parameter\_number := numeric\_expression*

**PARSTR\$** (*name, parameter\_number*)

example: **10 pstring fred, joe, 'mary'**

```
****
70 DEF PROC pstring (n1,n2,n3)
80 PRINT PARSTR$(n1,1), PARSTR$(n2,2), PARSTR$(n3,3)
90 END DEF pstring
```

would print 'fred joe mary'.

## **PARTYP**

### **PARUSE** procedures

The function **PARTYP** when used in a procedure will return the type of the named parameter.

The type returned is: 0 for null  
1 for string  
2 for floating point  
3 for integer

The function **PARUSE** when used in a procedure will return the usage of the named parameter.

The usage returned is: 0 for unset  
1 for variable  
2 for array

syntax: **PARTYP** (*name*)  
**PARUSE** (*name*)

## **PAR\_BUFF** devices

**PAR\_BUFF** specifies the output buffer size. The output buffer should be at least 5 bytes to avoid confusion with the port number. If the output buffer is specified as zero length, a dynamic buffer is used.

syntax: *port := numeric\_expression*  
*output\_buff := numeric\_expression*

**PAR\_BUFF** *port, output\_buff*

example: i. **PAR\_BUFF 1,200** {200 byte output buffer on PAR1}  
ii. **PAR\_BUFF 2,0** {dynamic output buffer on PAR2}

## **PAR\_CLEAR**

### **PAR\_ABORT** devices

**PAR\_CLEAR** and **PAR\_ABORT** clear the output buffers of any closed channels to the port. Channels still open are not affected. **PAR\_ABORT** also sends the "ABORTED" message to the port.

syntax: *port := numeric\_expression*

**PAR\_CLEAR** *port*  
**PAR\_ABORT** *port*

example: i. **PAR\_CLEAR 1** {clear output to PAR1}  
ii. **PAR\_ABORT 3** {abort output to PAR3}

## **PAR\_DEFAULTPRINTER\$** QPC

Returns the name of Windows' default printer.

syntax: *printer\$ = PAR\_DEFAULTPRINTER\$*

## **PAR\_GETPRINTER\$**

### **PAR\_SETPRINTER** QPC

Sets or gets the name of the printer connected to a PAR device. If the channel is connected to an LPT port the respective string is "LPT1".

syntax: *printer\$ = PAR\_GETPRINTER\$ (port)*  
**PAR\_SETPRINTER\$** *port, name\$*

example : **PRINT PAR\_GETPRINTER\$(1)**  
**PAR\_SETPRINTER 1, PAR\_DEFAULTPRINTER\$** {connect PAR1 with default Windows printer}

## **PAR\_GETFILTER**

### **PAR\_SETFILTER** QPC

Sets or gets the whether the specified PAR port uses a printer filter, like QPCPrint.

syntax: *status% = PAR\_GETFILTER (port)*  
**PAR\_SETFILTER** *port, status%*

example : **PRINT PAR\_GETFILTER(1)**  
**PAR\_SETFILTER 1, 1** {enable printer filter for PAR1}

## **PAR\_PRINTERCOUNT QPC**

Returns the number of printers installed on the Windows host system.

syntax: printers = **PAR\_PRINTERCOUNT**

## **PAR\_PRINTERNAME\$ QPC**

Returns the name of a specific printer installed on the Windows host system. The index can be from 1 to PAR\_PRINTERCOUNT.

syntax: name\$ = **PAR\_PRINTERNAME\$** (*index*)

## **PAR\_PULSE**

Not used in QPC. Sets the length of the strobe pulse of the parallel port.

## **PAR\_USE redirection**

The **PAR\_USE** command allows the parallel port to be used with software that only allows output to SER1 or SER2.

syntax: **PAR\_USE** *string\_expression*

example: **10 PAR\_USE "ser"**  
**20 COPY\_N "flp1\_myfile" TO "ser2"** {will send the file to PAR}  
**30 COPY\_N "flp1\_myfile" TO "ser1f"** {will print the file to PAR ending with  
a form feed}

comment: To print a file using the parallel port using free memory as a buffer enter the following:

```
10 PAR_USE "lpt"  
20 PRT_USE "par", "lpt"  
30 COPY_N "flp1_myfile" TO "par"
```

## **PAR\_WAIT**

Not used in QPC. The time the system waits at most between sending two bytes.

## PAUSE

**PAUSE** will cause a program to wait a specified period of time. Delays are specified in units of 20ms. If no delay is specified then the program will pause indefinitely. Keyboard input will terminate the **PAUSE** and restart program execution.

syntax: *delay:= numeric\_expression*

**PAUSE** [*delay*]

example: i. **PAUSE 50**            {wait 1 second}  
          ii. **PAUSE 500**        {wait 10 seconds}

## PEEK, PEEK\_W

### PEEK\_L BASIC

**PEEK** is a function which returns the contents of the specified memory location. **PEEK** has three forms which will access a byte (8 bits), a word (16 bits), or a long word (32 bits).

**PEEK** may be referenced from the system variables if the first parameter of **PEEK** is preceded by an exclamation mark, then the address of the peek is in the system variables or referenced via the system variables. There are two variations: direct and indirect references.

For direct references, the exclamation mark is followed by another exclamation mark and an offset within the system variables.

For indirect references, the exclamation mark is followed by the offset of a pointer within the system variables, another exclamation mark and an offset from that pointer.

**PEEK** may also be referenced from the SBASIC variables if the first parameter of **PEEK** is preceded by a backslash, then the address of the peek is in the SBASIC variables or referenced via the SBASIC variables. There are two variations: direct and indirect references.

For direct references, the backslash is followed by another backslash and an offset within the SBASIC variables.

For indirect references, the backslash is followed by the offset of a pointer within the SBASIC variables, another backslash and an offset from that pointer.

syntax: *address:= numeric\_expression*  
          | *!! numeric\_expression*  
          | *! numeric\_expression ! numeric\_expression*  
          | *\\ numeric\_expression*  
          | *\ numeric\_expression! \ numeric\_expression*

**PEEK**(*address*)        {byte access}  
**PEEK\_W**(*address*)    {word access}  
**PEEK\_L**(*address*)    {long word access}

example: i. **PRINT PEEK(12245)**        {byte contents of location 12245}  
          ii. **PRINT PEEK\_W(12)**        {word contents of locations 12 and 13}  
          iii. **PRINT PEEK\_L(1000)**    {long word contents of location 1000}  
          iv. **ramt = PEEK\_L(! ! \$20)**    {find the top of RAM \$20 bytes on from the base of the system variables}  
          v. **job1 = PEEK\_L(! \$68!4)**    {find the base address of Job 1 (4 bytes on from base of Job table)}  
          vi. **dal = PEEK\_W(\\ \$94)**        {find the current data line number}  
          vii. **n6 = PEEK\_W(\\ \$18!2+6\*8)**    {find the name pointer for the 6th name in the name table}  
          viii. **n!6 = PEEK(\\ \$20!n6)**    {...and the length of the name}  
          ix. **n6\$ = PEEK\$(\\ \$20!n6+1, n!6)**    {...and the name itself}

**warning:** For word and long word access the specified address must be an even address.

## PEEKs, PEEKs\_W

### PEEKs\_L BASIC

Supervisor mode access to I/O hardware in Atari emulator, not used in QPC.

### PEEK\$ BASIC

**PEEK\$** will return a string with the number of supplied bytes starting from the supplied address. The bytes need not, of course, be text.

syntax: *start\_address* := *numeric\_expression*  
*number\_of\_bytes* := *numeric\_expression*

**PEEK\$** (*start\_address*, *number\_of\_bytes*)

example: **PRINT PEEK\$(123456,20)** {will display the 20 bytes from address 123456}

### PEEKs\$ BASIC

Supervisor mode access to I/O hardware in Atari emulator, not used in QPC.

## PENUP

### PENDOWN turtle graphics

Operates the 'pen' in turtle graphics. If the pen is up then nothing will be drawn. If the pen is down then lines will be drawn as the turtle moves across the screen.

The line will be drawn in the *window* attached to the specified or default *channel*. The line will be drawn in the current ink colour for the channel to which the output is directed.

syntax: **PENUP** [*channel*]  
**PENDOWN** [*channel*]

example: i. **PENUP** {will raise the pen in the default channel}  
ii. **PENDOWN #2** {will lower the pen in the window attached to channel 2}

## PI maths function

**PI** is a function which returns the value of  $\pi$ .

syntax: **PI**

example: **PRINT PI**

## POINT

### POINT\_R graphics

**POINT** plots a point at the specified position in the *window* attached to the specified or default *channel*. The point is plotted using the *graphics coordinates system* relative to the *graphics origin*. If **POINT\_R** is used then all points are specified relative to the graphics cursor and are plotted relative to each other.

Multiple points can be plotted with a single call to **POINT**.

syntax: *x:=numeric\_expression*  
*y:=numeric\_expression*

*parameters:= x,y*

**POINT** [*channel*,] *parameters*\* [,*parameters*]\*

example: i. **POINT 256,128** {plot a point at (256,128)}  
ii. **POINT x,x\*x** {plot a point at (x,x\*x)}  
iii. **10 REPEAT example**  
**20 INK RND(255)**  
**30 POINT RND(100),RND(100)**  
**40 END REPEAT example**

## POKE, POKE\_W

### POKE\_L BASIC

**POKE** allows a memory location to be changed. For word and long word accesses the specified address must be an even address.

**POKE** has three forms which will access a byte (8 bits), a word (16 bits), a long word (32 bits).

**POKE** may be referenced from the system variables if the first parameter of **POKE** is preceded by an exclamation mark, then the address of the poke is in the system variables or referenced via the system variables. There are two variations: direct and indirect references.

For direct references, the exclamation mark is followed by another exclamation mark and an offset within the system variables.

For indirect references, the exclamation mark is followed by the offset of a pointer within the system variables, another exclamation mark and an offset from that pointer.

**POKE** may also be referenced from the SBASIC variables if the first parameter of **POKE** is preceded by a backslash, then the address of the poke is in the SBASIC variables or referenced via the SBASIC variables. There are two variations: direct and indirect references.

For direct references, the backslash is followed by another backslash and an offset within the SBASIC variables.

For indirect references, the backslash is followed by the offset of a pointer within the SBASIC variables, another backslash and an offset from that pointer.

**POKE** allows more than one value to be **POKEd** at a time. For **POKE\_W** and **POKE\_L**, the address may be followed by a number of values to poke in succession. For **POKE** the address may be followed by a number of values to poke in succession and the list of values may include strings. If a string is given, all the bytes in the string are **POKEd** in order. The length is not **POKEd**.

syntax: *address:= numeric\_expression*  
 | *!! numeric\_expression*  
 | *! numeric\_expression ! numeric\_expression*  
 | *\\ numeric\_expression*  
 | *\ numeric\_expression/ \ numeric\_expression*  
*data:= numeric\_expression*

**POKE** *address, data [ \* ,data | string \* ]* {byte access}  
**POKE\_W** *address, data [ \* ,data \* ]* {word access}  
**POKE\_L** *address, data [ \* ,data \* ]* {long word access}

example: i. **POKE 12235,0** {set byte at 12235 to 0}  
 ii. **POKE\_L 131072,12345** {set long word at 131072 to 12345}  
 iii. **POKE\_W ! ! \$8E,3** {set the auto-repeat speed to 3}  
 iv. **POKE ! \$B0!2, 'WIN'** {change the first three characters of DATA\_USE to WIN}

**warning:** Poking data into areas of memory used by SMSQ/E can cause the system to crash and data to be lost. Poking into such areas is not recommended.

## POKES, POKES\_W

### POKES\_L BASIC

Supervisor mode access to I/O hardware in Atari emulator, not used in QPC.

### POKE\$ BASIC

**POKE\$** will pokes the supplied string of bytes into memory, starting from the supplied address.

syntax: *start\_address := numeric\_expression*

**POKE\$** *start\_address, string*  
**POKE\$** *start\_address, strin)*

example: **POKE\$ 131072,"hello"** {will put the string "hello" into address 131072}

comment: **PEEK\$** and **POKE\$** can accept all the extended addressing facilities of **PEEK** and **POKE**. Indeed, **POKE\$** is identical to **POKE** which can now accept string parameters.

### POKES\$ BASIC

Supervisor mode access to I/O hardware in Atari emulator, not used in QPC.

## **PRINT** devices directory devices

Allows output to be sent to the specified or default channel. The normal use of **PRINT** is to send data to the QPC screen.

syntax:    *separator:=* | !  
                          | ,  
                          | \  
                          | ;  
                          | **TO** *numeric\_expression*

*item:=* | *expression*  
          | *channel*  
          | *separator*

**PRINT** \*[*item*]\*

Multiple print *separators* are allowed. At least one separator must separate *channel* specifications and *expressions*.

- example: i. **PRINT "Hello World"**  
          {will output Hello World on the default output device (channel 1)}
- ii. **PRINT #5,"data",1,2,3,4**  
      {will output the supplied data to channel 5 (which must have been previously opened)}
- iii. **PRINT TO 20; "This is in column 20"**

!        Normal action is to insert a space between items output on the screen. If the item will not fit on the current line a line feed will be generated. If the current print position is at the start of a line then a space will not be output. ! affects the next item to be printed and therefore must be placed in front of the print item being printed. Also a ; or a ! must be placed at the end of a print list if the spacing is to be continued over a series of **PRINT** statements.

,        Normal separator, SBASIC will tabulate output every 8 columns.

\        Will force a new line.

;        Will leave the print position immediately after the last item to be printed. Output will be printed in one continuous stream.

**TO**     Will perform a tabbing operation. **TO** followed by a *numeric\_expression* will advance the print position to the column specified by the *numeric\_expression*. If the requested column is meaningless or the current print position is beyond the specified position then no action will be taken.

## PRINT\_USING devices, directory devices

**PRINT\_USING** is a fixed format version of the **PRINT** command:

The 'format' is a string or string expression containing a template or 'image' of the required output. Within the format string the characters + - # \*, . ! \ ' " \$ and @ all have special meaning. When called, the procedure scans the format string, writing out the characters of the string, until a special character is found.

If the @ character is found, then the next character is written out, even if it is a special character.

If the character is a " or ' , then all the following characters are written out until the next " or ' .

If the \ character is found, then a newline is written out.

All the other special characters appear in format 'fields'. For each field an item is taken from the list, and formatted according to the form of the field and written out.

The field determines not only the format of the item, but also the width of the item (equal to the width of the field). The field widths in the examples below are arbitrary.

field	format
#####	if item is string, write string left justified or truncated otherwise write integer right justified
*****	write integer right justified empty part of field filled with * (e.g. ***12)
####.##	fixed point decimal (e.g. 12.67)
***.**	fixed point decimal, * filled (e.g. **12.67)
##,###.## **,**.*	fixed point decimal, thousands separated by commas (e.g. 1,234.56 or *1,234.56)
-#.#####!!!	exponent form (e.g. 2.9979E+08) optional sign
+#.#####!!!	exponent form always includes sign
###.>	fixed point decimal, scaled (i.e. if you calculate in pennies)

The exponent field must start with a sign, one #, and a decimal point (comma or full stop). It must end with four !s.

Any decimal field may be prefixed or postfixed with a + or -, or enclosed in parentheses. If a field is enclosed in parentheses, then negative values will be written out enclosed in parentheses. If a - is used then the sign is only written out if the value is negative; if a + is used, then the sign is always written out. If the sign is at the end of the field, then the sign will follow the value.

Numbers can be written out with either a comma or a full stop as the decimal point. If the field includes only one comma or full stop, then that is the character used as the decimal point. If there is more than one in the field, the last decimal point found (comma or full stop) will be used as the decimal point, the other is used as the thousands separator.

If the decimal point comes at the end of the field, then it will not be printed. This allows currencies to be printed with the thousands separated, but with no decimal point (e.g. 1,234).

Floating currency symbols are inserted into fields using the \$ character. The currency symbols are inserted between the \$ and the first # in the field (e.g. \$Dm#.###,## or +\$\$###,###.##). When the value is converted, the currency symbols are 'floated' to the right to meet the value.

syntax: **PRINT\_USING** #channel, format, \* items \*

example: **10** fmt\$='@\$ Charges \*\*\*\*\*.\*\* : (\$Kr###.###,##) : ##,###.##+'  
**20** PRINT\_USING fmt\$, 123.45, 123.45, 123.45  
**30** PRINT\_USING fmt\$, -12345.67, -12345.67, -12345.67  
**40** PRINT\_USING '-#.###!!!\!', 1234567

will print

\$ Charges ****123.45 :	SKr123,45 :	123.45+
\$ Charges *-12345.67 :	(SKr12.345,67) :	12,345.67-1.235E+06

## PROCESSOR SMSQ/E

PROCESSOR will return the Motorola MC680x0 family type.

syntax: **PROCESSOR**

example: **PRINT PROCESSOR**

comment: **PROCESSOR** will return 10 for QPC.

## PROG\_USE program default

The **PROG\_USE** default is used only for finding the program files for the **EX/EXEC** commands,

**PROG\_USE** is used to set a default, which is used only for finding the program files for the **EX/EXEC** commands, if you do not supply a complete SMSQ/E filename in the command, the **PROG\_USE** default will be added to the beginning of the supplied filename.

If the supplied filename is not found in the system, Then the **PROG\_USE** default will be added to the beginning of the supplied filename, and another attempt will be made to execute the command.

syntax: *directory\_name := device\*[subdirectory\_]\**

**PROG\_USE** *directory\_name*

example: **100** **PROG\_USE** win1\_programs\_  
**110** **EXEC** editor {Starts the executable program "win1\_programs\_editor}

comment: If the directory name supplied does not end with '\_', '\_' will be appended to the directory name.

## **PROT\_DATE** clock

**PROT\_DATE** is used to protect or unprotect the real time clock. If the real time clock is protected, setting the date affects only SMSQ's own clock, the real time will be restored then next time the computer is reset.

Where the system has a separate battery backed real time clock. The date is read from the clock when the system is reset. Thereafter, the clock is kept up to date by the SMSQ timer.

In general, the system real time clock is updated whenever you adjust or set the date. As some QL software writers could not resist the temptation of setting the date to their birthday (or other inconvenient date) this can play havoc with your file date stamps etc.

syntax:   **PROT\_DATE** *numeric\_expression*                    {0 or 1}

example:  i. **PROT\_DATE 0**                                    {date is not protected}  
          ii. **PROT\_DATE 1**                                    {date is protected}

## **PROT\_MEM**

Not used in QPC. Set the memory protection level in Atari emulators.

## **PRT\_BUFF** devices

**PRT\_BUFF** specifies the output buffer size. The output buffer should be at least 5 bytes to avoid confusion with the port number. If the output buffer is specified as zero length, a dynamic buffer is used.

syntax:   *port* := *numeric\_expression*  
          *output\_buff* := *numeric\_expression*

**PRT\_BUFF** *port, output\_buff*

example:  i. **PRT\_BUFF 1,200**                                {200 byte output buffer on PRT1}  
          ii. **PRT\_BUFF 2,0**                                 {dynamic output buffer on PRT2}

## **PRT\_CLEAR**

### **PRT\_ABORT** devices

**PRT\_CLEAR** and **PRT\_ABORT** clear the output buffers of any closed channels to the port. Channels still open are not affected. **PRT\_ABORT** also sends the "ABORTED" message to the port.

syntax:   *port* := *numeric\_expression*

**PRT\_CLEAR** *port*  
**PRT\_ABORT** *port*

example:  i. **PRT\_CLEAR 1**                                    {clear output to PRT1}  
          ii. **PRT\_ABORT 3**                                    {abort output to PRT3}

## **PRT\_USE** devices

**PRT\_USE** originally specified a name for the dynamic print buffer. However as all output ports now incorporate dynamic buffering, an "add-on" printer buffer is not required.

The SMSQ/E version of **PRT\_USE** is identical to that of the Atari ST drivers for QDOS. It merely specifies which port will be opened if you open the device PRT.

syntax: **PRT\_USE** [ *name* ]

example: i. **PRT\_USE PAR : COPY fred to PRT** {copy fred to PAR}  
ii. **PRT\_USE SER4XA : OPEN #5,PRT** {open a channel to SER4 with XON/XOFF and <CR><LF>}

## **PRT\_USE\$**

Returns the device set by **PRT\_USE**.

## **QPC\_CMDLINE\$** QPC

Returns the value specified on the QPC2 command line after the "-cmdline" parameter.

syntax: `cmdline$ = QPC_CMDLINE$`

example C:\QPC> **QPC2 -cmdline "This text will be returned by QPC\_CMDLINE\$"**

## **QPC\_EXEC** QPC

**QPC\_EXEC** will call an external DOS or Windows program. The name of the executable file is given in the first parameter. Optionally you can also supply the command line arguments with the second parameter.

Furthermore you can supply a data file as first parameter, in this case the default Windows viewer for this type of file is executed.

syntax: *program* := *string\_expression*  
*parameters* := *string\_expression*

**QPC\_EXEC** *program* [, *parameter* ]

example: i. **QPC\_EXEC 'notepad','c:\text.txt'** {Start notepad and load the c:\text file}  
ii. **QPC\_EXEC 'c:\text.txt'** {Start the default viewer for .TXT files}

## **QPC\_EXIT** QPC

**QPC\_EXIT** will quit QPC and returns to Windows.

syntax: **QPC\_EXIT**

## **QPC\_HOSTOS QPC**

**QPC\_HOSTOS** will return the host operating system under which QPC was started. Possible return codes are:

- 0 = DOS (QPC1)
- 1 = Win9x/ME (QPC2)
- 2 = WinNT/2000/XP (QPC2)

syntax: **QPC\_HOSTOS**

example: **system% = QPC\_HOSTOS**

## **QPC\_MAXIMIZE**

## **QPC\_MINIMIZE**

## **QPC\_RESTORE QPC**

**QPC\_MAXIMIZE**, **QPC\_MINIMIZE**, and **QPC\_RESTORE** will maximise, minimises or restore the QPC window.

syntax: **QPC\_MAXIMIZE**  
**QPC\_MINIMIZE**  
**QPC\_RESTORE**

## **QPC\_MSPEED QPC**

This command is supplied for compatibility reasons. It is used on QPC1 to change the mouse acceleration. It has no effect on QPC2.

## **QPC\_NETNAME\$ QPC**

**QPC\_NETNAME\$** will return the current network name of your PC (the one you supplied upon installation of Windows). This command can be used to distinguish between different PCs (e.g. in the BOOT program).

## **QPC\_QLSCREMU QPC**

**QPC\_QLSCREMU** will enable or disable the original QL screen emulation. When emulating the original screen, all memory write accesses to the area \$20000-\$207FFF are intercepted and translated into writes to the first 512x256 pixels of the big screen area. If the screen is in high colour mode, additional colour conversion is done.

Possible values are:

- 1: automatic mode
- 0: disabled (default)
- 4: force to 4 colour mode
- 8: force to 8 colour mode

When in QL colour mode the emulation just transfers the written bytes to the larger screen memory, i.e. when the big mode is in 4 colour mode, the original screen area is also treated as 4 colour mode. In high colour mode however the colour conversion can do both modes. In this case you can pre-select the emulated mode (4, 8 as parameter) or let the last issued **MODE** call decide (automatic mode). Please note that that the automatic mode does not work on a per-job basis, so any job which issues a **MODE** command changes the behaviour globally.

Please also note that this transition is one-way only, i.e. bytes written legally to the first 512x256 pixels are not transferred back to the original QL screen (in case of a high colours screen this would hardly be possible anyway). Unfortunately this also means that not all old programs run

perfectly with this type of emulation. If you experience problems, start the misbehaving application in 512x256 mode.

syntax: *value := numeric\_expression*

**QPC\_QLSCREMU** *value*

example: **QPC\_SCREMU 4** {force 4 colour mode}

## **QPC\_SYNCSCRAP** QPC

In order to quickly exchange text passages between Windows and SMSQ the syncscrap functionality was introduced. The equivalent of the Windows clipboard is the scrap extension of the menu extensions. After loading the menu extensions you can call this command which creates a job that periodically checks for changes in either the scrap or the Windows clipboard and synchronises their contents if necessary. Please note that only text contents is supported. The character conversion between the QL character set and the Windows ANSI set is done automatically. The line terminators (LF/CR, LF alone) are converted, too.

syntax: **QPC\_SYNCSCRAP**

## **QPC\_VER\$** QPC

**QPC\_VER\$** will return the current QPC version.

syntax: **QPC\_VER\$**

example: **v\$ = QPC\_VER\$**

comment: **QPC\_VER\$** will return 3.00 or higher.

## **QPC\_WINDOWTITLE** QPC

Amends the title of the QPC window within Windows. Can be used to distinguish between several running QPC instances.

syntax: **QPC\_WINDOWTITLE** *title\$*

example: **QPC\_WINDOWTITLE "Accounting"**

comment: The title will always be amended by the QPC version and the registration information.

## **QUIT** basic

**QUIT** will end any SBASIC daughter jobs whether it has been created by the **SBASIC** command, **EX** or any other means.

syntax: **QUIT**

comment: **QUIT** will not end the primary SBASIC job (job 0). To quit from this job, use **QPC\_QUIT**.

## **RAD** maths functions

**RAD** is a function which will convert an angle specified in degrees to an angle specified in radians.

syntax: **RAD** (*numeric\_expression*)

example: **PRINT RAD(180)** {will print 3.141593}

## **RAM\_USE** directory devices

**RAM\_USE** allows renaming of the RAM device. **RAM\_USE** without a parameter will reset the name of RAM back to RAM.

syntax: **RAM\_USE** [*name*]

example: i. **RAM\_USE flp : LOAD flp2\_prog** {loads 'prog' from RAM2\_ }  
ii. **RAM\_USE** {and now its name is RAM again}  
iii. **RAM\_USE win : DIR win1\_** {displays directory of RAM1\_ }

## **RANDOMISE** maths functions

**RANDOMISE** allows the random number generator to be reseeded. If a parameter is specified the parameter is taken to be the new seed. If no parameter is specified then the generator is reseeded from internal information.

syntax: **RANDOMISE** [*numeric\_expression*]

example: i. **RANDOMISE** {set seed to internal data}  
ii. **RANDOMISE 3.2235** {set seed to 3.2235}

## **RECOL** windows

**RECOL** will recolour individual pixels in the window attached to the specified or default *channel* according to some pre-set pattern. Each parameter is assumed to specify, in order, the colour in which each pixel is recoloured, i.e. the first parameter specifies the colour with which to recolour all black pixels, the second parameter blue pixels, etc.

The colour specification must be a solid colour, i.e. it must be in the range 0 to 7.

**RECOL** only works as specified in 4 or 8 colour QL mode. Using it in other screen modes has no effect.

syntax: *c0*:= new colour for black  
*c1*:= new colour for blue  
*c2*:= new colour for red  
*c3*:= new colour for magenta  
*c4*:= new colour for green  
*c5*:= new colour for cyan  
*c6*:= new colour for yellow  
*c7*:= new colour for white

**RECOL** [*channel* ,] *c0*, *c1*, *c2*, *c3*, *c4*, *c5*, *c6*, *c7*

example: **RECOL 2,3,4,5,6,7,1,0** {recolour blue to magenta, red to green, magenta to cyan etc.}

## REMark

**REMark** allows explanatory text to be inserted into a program. The remainder of the line is ignored by SBASIC.

syntax: **REMark** *text*

example: **REMark This is a comment in a program**

comment: **REMark** is used to add comments to a program to aid clarity.

## RENAME

### **WREN** directory devices

**RENAME** and **WREN** (wild card renaming) is a process similar to **COPY**ing a file, but the file itself is neither moved nor duplicated, only the directory name is changed. The commands, however, are exactly the same in use as the equivalent **COPY** commands.

syntax: **RENAME** *name* **TO** *name*  
**WREN** [#*channel*,] *name* **TO** *name*

## RENUM

**RENUM** allows a group or a series of groups of SBASIC line numbers to be changed. If no parameters are specified then **RENUM** will renumber the entire program. The new listing will begin at line 100 and proceed in steps of 10.

If a start line is specified then line numbers prior to the start line will be unchanged. If an end line is specified then line numbers following the end line will be unchanged.

If a start number and stop are specified then the lines to be renumbered will be numbered from the start number and proceed in steps of the specified size.

If a GOTO or GOSUB statement contains an expression starting with a number then this number is treated as a line number and is renumbered.

syntax: *startline*:= *numeric\_expression* {start renumber}  
*end\_line*:= *numeric\_expression* {stop renumber}  
*start\_number*:= *numeric\_expression* {base line number}  
*step*:= *numeric\_expression* {step}

**RENUM** [*start\_line* [**TO** *end\_line*];] [*startnumber*] [,*step*]

example: i. **RENUM** {renumber whole program from 100 by 10}  
ii. **RENUM 100 TO 200** {renumber from 100 to 200 by 10}

**warning:** No attempt must be made to use **RENUM** to renumber program lines out of sequence, ie to move lines about the program. **RENUM** should not be used in a program.

## REPeat

### END REPeat repetition

REPeat allows general repeat loops to be constructed. REPeat should be used with EXIT for maximum effect. REPeat can be used in both long and short forms:

**short:** The REPeat keyword and loop identifier are followed on the same logical line by a colon and a sequence of SBASIC *statements*. EXIT will resume normal processing at the next logical line.

syntax: REPeat *identifier* : *statements*

example: REPeat wait : IF INKEY\$ = "" THEN EXIT wait

**long:** The REPeat keyword and the loop identifier are the only statements on the logical line. Subsequent lines contain a series of SBASIC *statements* terminated by an END REPeat statement.

The statements between the REPeat and the END REPeat are repeatedly processed by SBASIC.

syntax: REPeat *identifier*  
*statements*  
END REPeat *identifier*

example: 10 LET number = RND(1 TO 50)  
11 REPeat guess  
12 INPUT "What is your guess?", guess  
13 IF guess = number THEN  
14 PRINT "You have guessed correctly"  
15 EXIT guess  
16 ELSE  
17 PRINT "You have guessed incorrectly"  
18 END IF  
19 END REPeat guess

comment: Normally at least one statement in a REPeat loop will be an EXIT statement.

## REPORT error handling

REPORT will report the description of the last error encountered to the specified or default channel. An optional error number may be supplied. If so, the error message for this number will be reported.

syntax: *error\_number* := *numeric\_expression*  
REPORT [#*channel*, ] [*error\_number*]

comment: The default channel is #0

## RESET

RESET will reset the computer. Using this command could result in loss of data (e.g. when you RESET while sectors are being written to your floppy disk or hard disk), therefore much care should be taken if this command is used without the control of the user.

syntax: RESET

## RESPR SMSQ/E

**RESPR** is a function which will reserve some of the resident procedure space. (For example to expand the SBASIC procedure list.)

syntax: *space* := *numeric\_expression*  
**RESPR** (*space*)

example: **PRINT RESPR(1024)** {will print the base address of a 1024 byte block}

## RETurn functions and procedures

**RETurn** is used to force a *function* or *procedure* to terminate and resume processing at the statement after the procedure or function call. When used within a function definition the **RETurn** statement is used to return the function's value.

syntax: **RETurn** [*expression*]

example: i. **100 PRINT ack (3,3)**  
**110 DEFine FuNction ack(m,n)**  
**120 IF m=0 THEN RETurn n+l**  
**130 IF n=0 THEN RETurn ack (m-l,l)**  
**140 RETern a c k (m-l ,a c k (m, n-l ) )**  
**150 END DEFine**

ii. **10 LET warning\_flag =1**  
**11 LET error\_number = RND(0 TO 10)**  
**12 warning error\_number**  
**13 DEFine PROCedure warning(n)**  
**14 IF warning\_flag THEN**  
**15 PRINT "WARNING:";**  
**16 SElect ON n**  
**17 ON n =1**  
**18 PRINT "Microdrive full"**  
**19 ON n = 2**  
**20 PRINT "Data space full"**  
**21 ON n = REMAINDER**  
**22 PRINT "Program error"**  
**23 END SElect**  
**24 ELSE**  
**25 RETurn**  
**26 END IF**  
**27 END DEFine**

comment: It is not compulsory to have a **RETurn** in a procedure. If processing reaches the **END DEFine** of a procedure then the procedure will return automatically.

**RETurn** by itself is used to return from a **GOSUB**.

## RJOB SMSQ/E

**RJOB** is a command to remove a job from SMSQ/E.

syntax: *job\_identifier* := | *job\_number* , *tag\_number*  
| *job\_number* + (*tag\_number* \* 65536)  
*id* := *job\_identifier*

**RJOB** *id* | *name* , *error\_code*

example: i. **RJOB 3,8,-1** {remove Job 3, tag 8 with error -1}  
ii. **RJOB 524291,-1** {Same as above}

comment: If a name is given rather than a Job ID, then the procedure will search for the first Job it can find with the given name.

## RND maths function

**RND** generates a random number. Up to two parameters may be specified for **RND**. If no parameters are specified then **RND** returns a pseudo random *floating point* number in the exclusive range 0 to 1. If a single parameter is specified then **RND** returns an integer in the inclusive range 0 to the specified parameter. If two parameters are specified then **RND** returns an integer in the inclusive range specified by the two parameters.

syntax: **RND**( [*numeric\_expression*] [**TO** *numeric\_expression*])

example: i. **PRINT RND** {floating point number between 0 and 1}  
ii. **PRINT RND(10 TO 20)** {integer between 10 and 20}  
iii. **PRINT RND(1 TO 6)** {integer between 1 and 6}  
iv. **PRINT RND(10)** {integer between 0 and 10}

## RUN program

**RUN** allows an SBASIC program to be started. If a line number is specified in the **RUN** command then the program will be started at that point, otherwise the program will start at the lowest line number.

syntax: **RUN** [*numeric\_expression*]

example: i. **RUN** {run from start}  
ii. **RUN 10** {run from line 10}  
iii. **RUN 2\*20** {run from line 40}

comment: Although **RUN** can be used within a program its normal use is to start program execution by typing it in as a direct command.

## SAVE, QSAVE

### SAVE\_O, QSAVE\_O devices, directory devices

**SAVE** will save a SBASIC program onto any QPC device.

**QSAVE** will save an SBASIC program, overwriting it if it already exists.

**QSAVE** and **QSAVE\_O** will save an SBASIC program in the quick load format with a **\_SAV** at the end of the filename.

syntax: *line:=* | *numeric\_expression TO numeric\_expression* (1)  
| *numeric\_expression TO* (2)  
| **TO** *numeric\_expression* (3)  
| *numeric\_expression* (4)  
| (5)

**SAVE** *device* \*[,*line*]\*  
**QSAVE** *device* \*[,*line*]\*  
**SAVE\_O** *device* \*[,*line*]\*  
**QSAVE\_O** *device* \*[,*line*]\*

where (1) will save from the specified line to the specified line  
(2) will save from the specified line to the end  
(3) will save from the start to the specified line  
(4) will save the specified line  
(5) will save the whole program



- example:
- i. **SAVE win1\_program,20 TO 70**  
     {save lines 20 to 70 on win1\_program}
  - ii. **QSAVE flp2\_test\_program,10,20,40**  
     {quick save lines 10,20,40 on flp1\_test\_program}
  - iii. **SAVE\_O dev1\_program**  
     {save the entire program to dev1\_program, overwriting if it exists}
  - iv. **SAVE ser1**  
     {save the entire program on serial channel }

## **SBASIC BASIC**

SBASIC will create a daughter SBASIC job.

Having a number of SBASIC jobs which completely cover each other may not be very useful. SBASIC daughter jobs may, therefore, either be created either with the full set of standard windows (in which case they all overlap) or they may be created with only one small window (#0).

The SBASIC command, has an optional parameter: the x and y positions of window #0 in a one or two digit number (or string).

If no parameters are given, the full set of standard windows will be opened. Otherwise, only window #0 will be opened: 6 rows high and 42 mode 4 characters wide within a 1 pixel wide border (total 62x256 pixels).

If only one digit is given, this is the SBASIC "row" number: row 0 is at the top, row 1 starts at screen line 64, row 4 is just below the standard window #0.

If two digits are given, this is the SBASIC "column, row" (x,y) position: column 0 is at the left, column 1 starts at 256 pixel in from the left.

syntax:    *row := numeric\_expression*  
           *columnrow := numeric\_expression*

**SBASIC [ row | columnrow ]**

- example:
- i. **SBASIC**            {create an SBASIC daughter with the 3 standard windows}
  - ii. **SBASIC 1**        {create an SBASIC daughter with just channel #0 in row 1}
  - iii. **SBASIC 24**      {create an SBASIC daughter to the right of and below the  
                           standard windows (an 800x600 display is required)}

comment: Because it is quite normal for an SBASIC job to have only #0 open, all the standard commands which default to window #1 (**PRINT**, **CLS** etc.) or window #2 (**ED**, **LIST** etc.) will default to window #0 if channel #1 or channel #2 is not open. This may not apply to extension commands.

## SBYTES

### SBYTES\_O devices, directory devices

SBYTES allows areas of QPC memory to be saved on a QPC device.

SBYTES\_O as SBYTES but overwrites the file if it exists.

If a channel number of an open channel is supplied in place of a filename, then SBYTES will attempt to save the file to the channel.

syntax: *start\_address:= numeric\_expression*  
*length:= numeric\_expression*  
*device := filename | channel*

**SBYTES** *device, start\_address, length*  
**SBYTES\_O** *device, start\_address, length*

example: i. **SBYTES flp1\_screendata, SCR\_BASE, SCR\_LLEN \* SCR\_YLIM**  
          {save screen image on flp1\_test\_program}  
ii. **SBYTES\_O ram1\_test\_program,50000,1000**  
      {save memory 50000 length 1000 bytes on ram1\_test\_program  
      overwriting if it already exists}  
iii. **SBYTES neto\_3,32768,32678**  
      {save memory 32768 length 32768 bytes on the network}  
iv. **SBYTES ser1,0,32768**  
      {save memory 0 length 32768 bytes on serial channel 1}  
v. **10 OPEN#5,ram1\_data** {open channel}  
   **20 SBYTES#5,50000,1000** {save 1000 bytes from address 50000}  
   **30 CLOSE#5** {close channel}

## SCALE graphics

SCALE allows the scale factor used by the graphics procedures to be altered. A scale of 'x' implies that a vertical line of length 'x' will fill the vertical axis of the *window* in which the figure is drawn. A scale of 100 is the default. SCALE also allows the origin of the coordinate system to be specified. This effectively allows the window being used for the graphics to be moved around a much larger graphics space.

syntax: *x:=numeric\_expression*  
*y:=numeric\_expression*  
*origin:= x,y*  
*scale:= numeric\_expression*

**SCALE** [*channel,*] *scale, origin*

example: i. **SCALE 0.5,0.1,0.1** {set scale to 0.5 with the origin at 0.1,0.1}  
ii. **SCALE 10,0,0** {set scale to 10 with the origin at 0,0}  
iii. **SCALE 100,50,50** {set scale to 100 with the origin at 50,50}

## **SCROLL windows**

**SCROLL** scrolls the window attached to the specified or default *channel* up or down by the given number of pixels. *Paper* is scrolled in at the top or the bottom to fill the clear space.

An optional third parameter can be specified to obtain a part screen scroll.

syntax: *part*:= *numeric\_expression*  
*distance*:= *numeric\_expression*

where *part* = 0 - whole screen (default is no parameter)  
*part* = 1 - top excluding the cursor line  
*part* = 2 - bottom excluding the cursor line

**SCROLL** [*channel*,] *distance* [, *part*]

If the distance is positive then the contents of the screen will be shifted down.

example: i. **SCROLL 10** {scroll down 10 pixels}  
ii. **SCROLL -70** {scroll up 70 pixels}  
iii. **SCROLL -10,2** {scroll the lower part of the window up 10 pixels}

## **SCR\_BASE**

### **SCR\_LLEN windows**

**SCR\_BASE** will return the base address of the screen attached to the specified or default channel.

**SCR\_LLEN** will return the line length in bytes of the screen attached to the specified or default channel.

syntax: **SCR\_BASE** [#*channel*]  
**SCR\_LLEN** [#*channel*]

example: i. **PRINT SCR\_BASE**  
ii. **PRINT SCR\_LLEN #1**

comment: In current versions, the values returned are the same for all screen channels.

## **SCR\_XLIM**

### **SCR\_YLIM windows**

**SCR\_XLIM** will return the maximum number of pixels across the screen (+1), available for the screen attached to the specified, or default channel.

**SCR\_YLIM** will return the maximum number of pixels down the screen (+1), available for the screen attached to the specified, or default channel.

syntax: **SCR\_XLIM** [#*channel*]  
**SCR\_YLIM** [#*channel*]

example: i. **PRINT SCR\_XLIM**  
ii. **PRINT SCR\_YLIM #1**

comment: The values returned are not the same as the current window size, but they defines the maximum size that a window can be. **SCR\_XLIM** and **SCR\_YLIM** should only be called for a primary window, usually #0 the default channel, for an SBASIC job.

## **SDATE** clock

The **SDATE** command allows QPC's and ultimately Windows's clock to be reset. If the user is not allowed to set Windows's clock, then the command has no effect.

syntax:    *year:=*        *numeric\_expression*  
          *month:=*     *numeric\_expression*  
          *day:=*        *numeric\_expression*  
          *hours:=*     *numeric\_expression*  
          *minutes:=*  *numeric\_expression*  
          *seconds:=* *numeric\_expression*

**SDATE** *year, month, day, hours, minutes, seconds*

example: i. **SDATE 1984,4,2,0,0,0**  
          ii. **SDATE 1984,1,12,9,30,0**  
          iii. **SDATE 1984,3,21,0,0,0**

## **SElect**

### **END SElect** conditions

**SElect** allows various courses of action to be taken depending on the value of a variable.

define:    *select\_variable:= numeric\_variable*  
          *select\_item:=*    | *expression*  
                          | *expression TO expression*  
*select\_list:=*        | *select\_item \*[, select\_item]\**

**long:**     Allows multiple actions to be selected depending on the value of a *select\_variable*. The select variable is the last item on the logical line. A series of SBASIC *statements* follows, which is terminated by the next **ON** statement or by the **END SElect** statement. If the select item is an expression then a check is made within approximately 1 part in  $10^{-7}$ , otherwise for expression **TO** expression the range is tested exactly and is inclusive. The **ON REMAINDER** statement allows a, "catch-all" which will respond if no other select conditions are satisfied.

syntax:    **SElect ON** *select\_variable*  
            \*[[**ON** *select\_variable*] = *select\_list*  
              *statements*] \*  
            [**ON** *selectvariable*] = **REMAINDER**  
              *statements*  
**END SElect**

example:    **100 LET error\_number = RND(1 TO 10)**  
            **110 SElect ON error\_number**  
            **120 ON error\_number = 1**  
            **130 PRINT "Divide by zero"**  
            **140 LET error\_number = 0**  
            **150 ON error\_number = 2**  
            **160 PRINT "File not found"**  
            **170 LET error\_number = 0**  
            **180 ON error\_number = 3 TO 5**  
            **190 PRINT "Microdrive file not found"**  
            **200 LET error\_number = 0**  
            **210 ON error\_number = REMAINDER**  
            **220 PRINT "Unknown error"**  
            **230 END SElect**

If the select variable is used in the body of the **SElect** statement then it must match the select variable given in the select header.

**Short:** The short form of the **SElect** statement allows simple single line selections to be made. A sequence of SBASIC statements follows on the same logical line as the **SElect** statement. If the condition defined in the select statement is satisfied then The sequence of SBASIC statements is processed.

syntax: **SElect ON** *select\_variable* = *select\_list* : *statement* \*[: *statement*] \*

example: i. **SElect ON test data =1 TO 10 :**  
**PRINT "Answer within range"**  
 ii **SElect ON answer = 0.00001 TO 0.00005 :**  
**PRINT "Accuracy OK"**  
 iii. **SElect ON a =1 TO 10 : PRINT a ! "in range"**

comment: The short form of the **SElect** statement allows ranges to be tested more easily than with an **IF** statement. Compare example ii. above with the corresponding **IF** statement.

## SEND\_EVENT

**SEND\_EVENT** event procedure is used to notify events to another job. The job ID can be the whole number, the job number and tag or the job name.

syntax: *jobID* := *numeric\_expression*  
 | *job\_number* , *job\_tag*  
 | *job\_name*  
*event* := *numeric\_expression* {in the range 1 to 256}

**SEND\_EVENT** *jobID*, *event*

example: i. **SEND\_EVENT 'fred',9** {Send events 1 and 8 (1 +8=9) to job fred}  
 ii. **SEND\_EVENT 20,4,8** {Send event 8 to job 20, tag 4}  
 iii. **SEND\_EVENT OJOB(-1),2** {Send event 2 to my owner}

## SER\_BUFF devices

**SER\_BUFF** specifies the output buffer size and, optionally, the input buffer size. The output buffer should be at least 5 bytes to avoid confusion with the port number. If the output buffer is specified as zero length, a dynamic buffer is used.

syntax: *port* := *numeric\_expression*  
*input\_buff* := *numeric\_expression*  
*output\_buff* := *numeric\_expression*

**SER\_BUFF** *port*, *output\_buff*, *input\_buff*

example: i. **SER\_BUFF 200** {200 byte output buffer on SER1}  
 ii. **SER\_BUFF 4,0,80** {dynamic output buffer, 80 byte input buffer on SER4}

## SER\_CDEOF devices

**SER\_CDEOF** specifies a timeout from the CD line being negated to the channel returning an end of file. The timeout should be at least 5 ticks to avoid confusion with the port number. If the timeout is zero, the CD line is ignored.

syntax: *port* := *numeric\_expression*  
*ticks* := *numeric\_expression*

**SER\_CDEOF** *port*, *ticks*

example: **SER\_CDEOF 2,100** {wait 100 ticks before timing out}

## SER\_CLEAR

### SER\_ABORT devices

**SER\_CLEAR** and **SER\_ABORT** clear the output buffers of any closed channels to the port. Channels still open are not affected. **SER\_ABORT** also sends the "ABORTED" message to the port.

syntax: *port := numeric\_expression*

**SER\_CLEAR** *port*

**SER\_ABORT** *port*

example: i. **SER\_CLEAR 1** {clear output to SER1}  
ii. **SER\_ABORT 3** {abort output to SER3}

## SER\_FLOW devices

**SER\_FLOW** specifies the flow control for the port: "Hardware", "XON/XOFF" or "Ignored". It usually takes effect immediately. If, however, the current flow is "Hardware" and handshake line CTS is negated and there is a byte waiting to be transmitted, the change will not take effect until either the handshake is asserted, or there is an output operation to that port

The default flow control is hardware unless the port does not have any handshake connections, in which case XON/XOFF is the default.

The flow control for a port is reset if a channel is opened to that port with a specific handshaking (H, X or I) option.

syntax: *port := numeric\_expression*  
*hand\_shake := H | X | I* {Hardware, XON/XOFF, or Ignore}

**SER\_FLOW** *port, hand\_shake*

example: i. **SER\_FLOW X** {XON/XOFF on SER1}  
ii. **SER=FLOW 2,H** {Hardware (default) handshaking on SER2}

## SER\_GETPORT\$

### SER\_SETPORT QPC

Sets or gets the COM port connected to a SER device.

syntax: *com\$ = SER\_GETPORT\$(port)*  
**SER\_SETPORT** *port, com\$*

example: **PRINT SER\_GETPORT\$(1)** {will per default print "COM1"}  
**SER\_SETPORT 2, "COM4"** {connect SER2 with COM4}

## SER\_PAUSE

Not used in QPC. Sets the length of the stop bits on the serial ports.

## SER\_ROOM devices

SER\_ROOM specifies the minimum level for the spare room in the input buffer. When the input buffer is filled beyond this level, the handshake (hardware or XOFF as specified by SER\_FLOW) is negated to stop the flow of data into the port. Some spare room is required to handle overruns (not all operating systems can respond as quickly as SMSQ). For hardware handshaking, a few spare bytes are all that is required. For connection to a dinosaur using XON/XOFF handshaking, up to 1000 spare bytes may be required.

syntax: *port* := *numeric\_expression*  
*room* := *numeric\_expression*

**SER\_ROOM** *port*, *room*

example: i. **SER\_FLOW 2,X : SER\_ROOM 2,1000** {connect SER2 to a UNIX system}  
ii. **SER\_FLOW 1,H : SER\_ROOM 1,4** {hardware handshaking on SER1}

comment: **SER\_ROOM** will not usually be required as **SER\_BUFF** also sets **SER\_ROOM** to one quarter of the buffer size. You will not succeed in setting **SER\_ROOM** to greater than **SER\_BUFF**, however, as **SER\_ROOM** will always ensure that the buffer is at least twice the size of the spare room.

## SER\_USE devices

**SER\_USE** specifies a name for the serial ports. The name can be SER or PAR. **SER\_USE** is provided for compatibility, its use is not recommended.

syntax: **SER\_USE** [*name*]

example: i. **SER\_USE PAR** {From now on, when you open PAR, you open a serial port}  
ii. **SER\_USE SER** {Sets you back to normal}  
iii. **SER\_USE** {..as does this}

## SET\_FUPDT

### SET\_FBKDT, SET\_FVERS directory devices

These three commands are used to set the update date, the backup date, and the version number of a file.

**SET\_FUPDT** will set the update date in the specified file, or the file connected to the specified or default channel, to the current or specified date and time.

**SET\_FBKDT** will set the backup date in the specified file, or the file connected to the specified or default channel, to the current or specified date and time.

**SET\_FVERS** will set the version number of the specified file, or the file connected to the specified or default channel, to the specified version number.

syntax: **SET\_FUPDT** [*\filename*, ] | [*channel*, ] [*date*]  
**SET\_FBKDT** [*\filename*, ] | [*channel*, ] [*date*]  
**SET\_FVERS** [*\filename*, ] | [*channel*, ] [*numeric\_expression*]

example: i. **SET\_FUPDT #5** {set update date to now}  
ii. **SET\_FUPDT \flp1\_fred,DATE-24\*60\*60** {set update of flp1\_fred to 24 hours ago}  
iii. **SET\_FBKDT \flp1\_fred** {set backup date of flp1\_fred to now}  
iv. **SET\_FBKDT #4,DATE(2002,7,10,13,32,15)** {set backup date to 10<sup>th</sup> July 2002 1:32 PM and 15 seconds}  
v. **SET\_FVERS #5** {do not increment version number}

- vi. **SET\_FVERS #5,1** {set version number to 1}
- vii. **SET\_FVERS \flp1\_fred,2** {set version number of flp1\_fred to 2}

comment: A date or a version number of 0 will have the same effect as omitting it. A date of a version number of -1 will have no effect on the file. If the update date has been set it will not be reset when the file is closed. If the version number has been set it will not be incremented when the file is closed.

## SEXEC

### SEXEC\_O SMSQ/E

Will save an area of memory in a form which is suitable for loading and executing with the **EXEC** command.

**SEXEC\_O** is the same as **SEXEC**, but will overwrite the file if it already exists.

The data saved should constitute a machine code program.

If a channel number of an open channel is supplied in place of a filename, then **SBYTES** will attempt to save the file to the channel.

syntax: *device := filename | channel*  
*start\_address:= numeric\_expression* {start of area}  
*length:= numeric\_expression* {length of area}  
*data\_space:= numeric\_expression* {length of data area which will be required by  
the program}  
**SEXEC** *device, start\_address, length, data\_space*  
**SEXEC\_O** *device, start\_address, length, data\_space*

example: i. **SEXEC flp1\_program,262144,3000,500**  
ii. **10 OPEN#5,flp1\_program** {open channel}  
**20 SEXEC\_O#5,50000,1000** {save 1000 bytes from address 50000}  
**30 CLOSE#5** {close channel}

The QDOS, SMSQ/E system documentation should be read before attempting to use this command.

## SIN maths function

**SIN** will compute the sine of the specified parameter.

syntax: *angle:= numeric\_expression* {range -10000..10000 in radians}

**SIN**(*angle*)

example: i. **PRINT SIN(3)**  
ii. **PRINT SIN(3.141592654/2)**

## SLUG

**SLUG** will delay all subsequent reads of the keyboard by a supplied amount in thousandths of a second (milliseconds). This is to allow some programs which are too fast in QPC to be slowed down.

syntax: **SLUG** *numeric\_expression*

example: **SLUG 15** {add a 15 thousandths of a second delay}



## SPJOB SMSQ/E

**SPJOB** is a command to set a jobs priority.

syntax: *job\_identifier* := | *job\_number* , *tag\_number*  
| *job\_number* + (*tag\_number* \* 65536)  
*id* := *job\_identifier*

**SPJOB** *id* | *name* , *priority*

example: i. **SPJOB demon,1** {set the priority of the Job called 'demon' to 1}  
ii. **SPJOB 2,1,80** {set the priority of the Job number 2, Tag number 1 to 80}

comment: If a name is given rather than a Job ID, then the procedure will search for the first Job it can find with the given name.

Setting a jobs priority to zero will suspend the job.

## SPL

### SPLF devices

**SPL** and **SPLF** will copy files in the background in the same way as **COPY\_O**, but is primarily intended for copying files to a printer. As an option, a form feed (ASCII <FF>) can be sent to the printer at the end of file.

syntax: **SPL** *name* **TO** *name* {spool a file}  
**SPLF** *name* **TO** *name* {spool a file, <FF> at end}

The separator TO is used for clarity, you may use a comma instead.

A variation on the **SPL** and **SPLF** commands is to use SBASIC channels in place of the filenames. These channels should be opened before the spooler is invoked:

syntax: **SPL** *#channel3* **TO** *#channel2*

Where channel3 must have been opened for input and channel2 must have been opened for output.

The normal use of this command is with one name only:

example: i. **SPL win1\_doc\_text TO par** {spool win1\_doc\_text to par}  
ii. **SPL\_USE ser** {set spooler default}  
.....  
**SPLF fred** {spool fred to ser, adding a form feed to the file}

comment: When used in this way, if the default device is in use, the Job will be suspended until the device is available. This means that many files can be spooled to a printer at once.

## SPL\_USE

**SPL\_USE** is used to set a default, which is used to find the destination filename or device for background spooling.

If the supplied device and filename is not found in the system, Then the **SPL\_USE** default will be added to the beginning of the supplied filename, and another attempt will be made to execute the command.

syntax: *directory\_name := device\*[subdirectory]\**

**SPL\_USE** *device\_name*

- example
- i. **DEST\_USE flp2\_old** {default is FLP2\_OLD\_}  
----  
**SPL fred**
  - ii. **SPL\_USE flp2\_old\_** {default is FLP2\_OLD\_}  
----  
**SPL fred**

Both of these examples will spool FRED to FLP2\_OLD\_FRED. Whereas if **SPL\_USE** is used with a name without a trailing '\_' (i.e. not a directory name) as follows

**SPL\_USE ser** {default is SER}  
----  
**SPL fred**

then FRED will be spooled to **SER** (not SER\_FRED).

Note that **SPL\_USE** overwrites the **DEST\_USE** default and vice versa

## SQRT maths function

will compute the square root of the specified argument. The argument must be greater than or equal to zero.

syntax: **SQRT** (*numeric\_expression*) {range >= 0}

- example:
- i. **PRINT SQRT(3)** {print square root of 3}
  - ii. **LET C = SQRT(a^2+b^2)** {let c become equal to the square root of a^2 + b^2}

## STAT directory devices

**STAT** will obtain and display in the window attached to the specified or default channel the directory device statistics for that drive.

syntax: **STAT** [*#channel*,] *name*  
**STAT** \i*name1*, *name2*

comment: Both the channel and the name are optional

## STOP BASIC

**STOP** will terminate execution of a program and will return SBASIC to the *command interpreter*.

syntax: **STOP**

example: i. **STOP**  
ii. **IF n =100 THEN STOP**

You may **CONTINUE** after **STOP**.

comment: The last executable line of a program will act as an automatic stop.

## STRIP windows

**STRIP** will set the current strip colour in the window attached to the specified or default *channel*. The strip colour is the background colour which is used when **OVER 1** is selected. Setting **PAPER** will automatically set the strip colour to the new **PAPER** colour.

syntax: **STRIP** [*channel*,] *colour*

example: i. **STRIP 7** {set a white strip}  
ii. **STRIP 0,4,2** {set a black and green stipple strip}

comment: The effect of **STRIP** is rather like using a highlighting pen.

## TAN maths functions

**TAN** will compute the tangent of the specified argument. The argument must be in the range -30000 to 30000 and must be specified in radians.

syntax: **TAN** (*numeric\_expression*) {range -30000..30000}

example: i. **TAN(3)** {print tan 3}  
ii. **TAN(3.141592654/2)** {print tan PI/2}

## TH\_FIX

No information available on this command.

## TK2\_EXT

If, for any reason, some of the SBASIC extensions have been re-defined, **TK2\_EXT** will reassert the common commands and functions .

syntax: **TK2\_EXT**

## TRA

**TRA** allows you to set up a translation table for a printer.

The SBASIC **TRA** command differs very slightly in use from the QL JS and MG **TRA**. The differences are quite deliberate and have been made to avoid the unfortunate interactions between functions of setting the Operating System message table and setting the printer translate tables. If you only wish to set the printer translate tables, the only difference is that **TRA 0** and **TRA 1** merely activate and deactivate the translate. They do not smash the pointer to the translate tables if you have previously set it with a **TRA** address command.

If you wish to change the system message tables, then the best way is to introduce a new language: this is done by **LRESPR**ing suitable message tables.

Language dependent printer translate tables are selected by the **TRA 1,lang** command. If no language code or car registration code is given, the currently defined language is used.

Language independent translate tables are set by the **TRA n** command where n is a small odd number.

Private translate tables are set by the **TRA addr** command where addr is the address of a table with the special language code \$4AFB.

syntax: *lang := language\_code | registration*  
*address := numeric\_expression*

**TRA [ lang | address ]**

example: i. **TRA 0** {translate off, table unchanged}  
ii. **TRA 0, 44** {translate off, table set to English}  
iii. **TRA 0, F** {translate off, table set to French}  
iv. **TRA 1** {translate on, table unchanged}  
v. **TRA 1, GB** {translate on, table set to English}  
vi. **TRA 1, 33** {translate on, table set to French}  
vii. **TRA 3** {translate on, table set to IBM graphics}  
viii. **TRA 5** {translate on, table set to GEM VDI}

**A = RESPR (512): LBYTES "tratab",A: TRA A** {translate on, table set to table in "tratab"}

comment: To use the language independent tables, your printer should be set to USA (to ensure that you have all the # \$ @ [ ] { } \ ^ ~ symbols which tend to go missing if you use one of the special country codes (thank you ANSI)), and select IBM graphics or GEM character codes as appropriate.

For the IBM tables, QDOS codes \$C0 to \$DF are passed through directly and QDOS codes \$E0 to \$EF are translated to \$B0 to \$BF to give you all the graphic characters in the range \$B0 to \$DF. QDOS codes \$F0 to \$FF are passed through directly to give access to the odd characters at the top of the IBM set. For the GEM tables, QDOS codes \$C0 to \$FF are passed through directly.

## TRUNCATE

**TRUNCATE** will delete the contents of the file connected to the specified or default channel, from the current or specified position to the end of the file.

syntax: **TRUNCATE #channel***position*

example: **TRUNCATE #dbchan** {truncate the file open on channel dbchan}

comment: If the position is not given, the file will be truncated to the current position

## TURN

### TURNTO turtle graphics

**TURN** allows the heading of the 'turtle' to be turned through a specified angle while **TURNTO** allows the turtle to be turned to a specific heading.

The turtle is turned in the *window* attached to the specified or default *channel*.

The angle is specified in degrees. A positive number of degrees will turn the turtle anti-clockwise and a negative number will turn it clockwise.

Initially the turtle is pointing at  $0^{\circ}$ , that is to the right hand side of the window.

syntax: *angle* := *numeric\_expression* {angle in degrees}

**TURN** [*channel*,] *angle*  
**TURNTO** [*channel*,] *angle*

example: i. **TURN 90**            {turn through  $90^{\circ}$  }  
          ii. **TURNTO 0**        {turn to heading  $0^{\circ}$  }

## UNDER windows

Turns underline either on or off for subsequent output lines. Underlining is in the current **INK** colour in the *window* attached to the specified or default *channel*.

syntax: *switch* := *numeric\_expression*            {range 0..1}

**UNDER** [*channel*,] *switch*

example: i. **UNDER 1**            {underlining on}  
          ii. **UNDER 0**        {underlining off}

## VER\$ SBASIC

**VER\$** will return system version information.

**VER\$** without parameters, or with a parameter of 0 will return the SBASIC version.

A parameter of 1 will return the SMSQ version number, a parameter of -1 will return the job ID, and a parameter of -2 will return the address of the system variables.

syntax: **VER\$** [ ( *numeric\_expression* ) ]

example: i. **PRINT ver\$**            {prints HBA (or later SBASIC version ID)}  
          ii. **PRINT ver\$(0)**        {also prints HBA (or later SBASIC version ID)}  
          iii. **PRINT ver\$(1)**        {prints 3.00 (or later SMSQ version number)}  
          iv. **PRINT ver\$(-1)**        {print the Job ID (0 for initial SBASIC)}  
          v. **PRINT ver\$(-2)**        {prints the address of the system variables (163840)}

## VIEW

**VIEW** allows a file to be examined in a window on the QPC display. The default window is #1.

**VIEW** truncates lines to fit the width of the window. When the window is full, **CTRL F5** is generated. If the output device (or file) is not a *console*, then lines are truncated to 80 characters.

syntax: **VIEW** [*channel*,] *device*  
**VIEW** \*device*,*device*

example: i. **VIEW** win1\_boot {View file 'win1\_boot' in window #1}  
ii. **VIEW** #3, flp1\_readme\_text {View file 'flp1\_readme\_text' in window #3}  
iii. **VIEW** \ser1,win1\_boot {Send file 'win1\_boot' to serial port 1}

## WAIT\_EVENT

The **WAIT\_EVENT** function is used to wait for one or more events. 8 events are defined; they are numbered 1, 2, 4, 8 ...256. The timeout is an optional 9th event .

The function returns the event or events that have occurred. The events that are returned are removed from the job's "event accumulator". Note that, if **WAIT\_EVENT** is called to wait for events 2 or 4 and events 2 and 8 have occurred, only event 2 is returned: event 8 remains pending and can be checked on another call.

If a timeout is specified, then, if no event of interest has occurred before the end of the timeout, the call will return the value 0 (no events). A timeout 0 can be used to check for events.

syntax: *event\_mask* := *numeric\_expression* {in range 1 to 256}  
*timeout* := *numeric\_expression*

**WAIT\_EVENT ( *event\_mask*, [ *timeout* ] )**

example: i. **evt = WAIT\_EVENT (6)** {Wait for event 2 or 4 (2++4=6)  
Events 2 and 8 are notified by another job so  
the wait is terminated and evt is set}  
ii. **PRINT evt** {Prints 2}  
iii. **PRINT WAIT\_EVENT (15)** {Wait for event 1,2,3,4, or 8, prints 8 as event 8  
is pending}  
iv. **PRINT WAIT\_EVENT (15)** {Wait for event 1,2,3,4, or8, wait as no events  
now pending}  
v. **evt = WAIT\_EVENT (6,50)** {Wait for event 2 or 4 (2++4=6) for no more  
than 1 second No events are notified by  
another job so the wait is terminated after one  
second and evt is set to 0}  
vi. **PRINT evt** {Prints 0}  
vii. **PRINT WAIT\_EVENT (3,0)** {Test for event 1 or2 without waiting}

## WDIR

### WSTAT directory devices

**WDIR** will obtain and display in the window attached to the specified or default channel the directory of the device using wild card names (Add **WDIR** to **DIR**)

**WSTAT** will obtain and display in the window attached to the specified or default channel the directory of the device together with file size and update date. Using wild card names

syntax:    **WDIR** [#channel,] name                    {list of files}  
          **WSTAT** [#channel,] name                 {list of files and their Statistics}

example: i. **WDIR**                                   list current directory to #1  
          ii. **WDIR** #channel                   list current directory to #channel  
          iii. **WDIR** \par                   list current directory to the parallel port  
          iv. **WDIR** win1\_data\_           list directory "win1\_data\_" to #1  
          v. **WSTAT** #4, flp2\_           list directory statistics of flp2\_ to channel 4  
          vi. **WDIR** \name1, name2       list directory 'name2' to 'name1'  
          vii. **WDIR** \ser, \_asm       list all \_asm files in current directory to SER  
          viii. **WSTAT** flp1\_       list all file statistics on FLP1\_ in window #1  
          ix. **WDIR** #3                   list all files in current directory to channel #3

## WHEN ERROR

### END WHEN error handling

Error handling is invoked by a **WHEN ERROR** clause. Unlike procedure and function definitions, these clauses are static. The error handling within a **WHEN ERROR** clause is set up when the clause is executed, but is only actioned **WHEN** an **ERROR** occurs. This means that a program may have more than one **WHEN ERROR** clause. As each one is executed, the error processing within that clause replaces the previously defined error processing.

The clause is opened with a **WHEN ERROR** statement, and closed with an **END WHEN** statement. Within the clause there may be any normal type of statement. (Although it might be better to avoid calling SBASIC functions or procedures!) A **WHEN ERROR** clause is exited by a **STOP**, **CONTINUE**, **RETRY**, **RUN**, **LOAD** or **LRUN** command. Furthermore **RUN**, **NEW**, **CLEAR**, **LOAD**, **LRUN**, **MERGE** and **MRUN** will reset the error processing.

syntax:    **WHEN ERROR**

There are some additional facilities intended for use within **WHEN ERROR** clauses.

#### ERROR functions

These functions correspond to each of the system error codes

<b>ERR_NC</b>	Not Complete,	<b>ERR_NJ</b>	Invalid Job,
<b>ERR_OM</b>	Out of Memory,	<b>ERR_OR</b>	Out of Range,
<b>ERR_BO</b>	Buffer Full,	<b>ERR_NO</b>	Channel not Open,
<b>ERR_NF</b>	Not Found,	<b>ERR_EX</b>	Already Exists,
<b>ERR_IU</b>	In Use,	<b>ERR_EF</b>	End of File,
<b>ERR_DF</b>	Drive Full,	<b>ERR_BN</b>	Bad Name,
<b>ERR_TE</b>	Transmit Error,	<b>ERR_FF</b>	Format Failed,
<b>ERR_BP</b>	Bad Parameter,	<b>ERR_FE</b>	Bad or Changed Medium,
<b>ERR_XP</b>	Error in Expression,	<b>ERR_OV</b>	Overflow,
<b>ERR_NI</b>	Not Implemented,	<b>ERR_RO</b>	Read Only,
<b>ERR_BL</b>	Bad line		

and return the value **TRUE** if the error, which caused the **WHEN ERROR** clause to be invoked, is of that type.

example: **10 WHEN ERROR**  
**20 IF ERR\_BP THEN PRINT "Bad Parameter error"**  
**30 IF ERR\_OV THEN PRINT "An Overflow has occurred"**  
**40 IF ERR\_NO THEN PRINT "Channel is not open"**  
**50 END WHEN**

## **WIDTH** devices

**WIDTH** allows the default width for non-console based devices to be specified, for example printers.

syntax: *line\_width:= numeric\_expression*

**WIDTH** [*channel*,] *line\_width*

example: i. **WIDTH 80** {set the device width to 80}  
ii. **WIDTH #6,72** {set the width of the device attached to channel 6 to 72}

## **WINDOW** windows

Allows the user to change the position and size of the *window* attached to the specified or default channel. Any borders are removed when the window is redefined.

Coordinates are specified using the *pixel system* relative to the screen origin.

syntax: *width:= numeric\_expression*  
*depth:= numeric\_expression*  
*x:= numeric\_expression*  
*y:= numeric\_expression*

**WINDOW** [*channel*,] *width, depth, x, y*

example: **WINDOW 30, 40, 10, 10** {window 30x40 pixels at 10,10}

## **WIN\_DRIVE**

### **WIN\_DRIVE\$**

**WIN\_DRIVE** allows you define the Windows path and filename for the WIN directory devices.

**WIN\_DRIVE\$** is a function to return the currently defined Windows path and filename of WIN directory devices.

syntax: **WIN\_DRIVE** *drive\_number, filename*  
**WIN\_DRIVE\$** (*drive\_number*)

example: i. **WIN\_DRIVE 2,"D:\QPC.WIN"** {WIN2\_ is assigned to the WIN file QPC.WIN}  
ii. **PRINT WIN\_DRIVE\$(2)** {will tell you the current filename}

## WIN\_FORMAT

Before you can issue the **FORMAT** command for a WIN device, you have to allow the drive to be formatted. SMSQ/E has a two-level protection scheme, to make sure you (or somebody else) cannot format your hard disk accidentally. All drives are protected by default, so you have to declare them to be formattable before you issue the **FORMAT** command.

**FORMAT** will fail if there is not sufficient space left on the specified drive, if the medium is write-protected, or if the file \*.WIN already exists and contains invalid information (e.g. a DOS-subdirectory).

syntax: *switch* := 0 | 1

**WIN\_FORMAT** *drive* [ ,*switch* ]

example: **WIN\_FORMAT 1** {Allow WIN1\_ to be formatted}  
**FORMAT WIN1\_10** {Create a 10 Megabyte WIN device on...  
you have to echo the two characters displayed ...  
**WIN\_FORMAT 1,0** {protect WIN1\_ again against unwanted formatting}

## WIN\_REMV

WIN\_REMV allows support for removable drives, like ZIP or SyQuest. It allows you to declare a WIN device to be removable.

When a drive is declared removable the .WIN file is closed after all SMSQ files on it are closed. This can also be used to share a single .WIN file over a network (files on a remote computer are automatically set to removable). Just as long as one QPC instance has any open files on the drive, all others cannot access it.

syntax: *switch* := 0 | 1

**WIN\_REMV** *drive\_number*, *switch*

example: i. **WIN\_REMV 2** {declares WIN2\_ to be a removable}  
ii. **WIN\_REMV 2,1** {does the same to WIN2\_}  
iii. **WIN\_REMV 2,0** {declares WIN2\_ is not a removable}

## WIN\_SLUG

### WIN\_START, WIN\_STOP

These commands have no effect in QPC.

## WIN\_USE directory devices

**WIN\_USE** allows renaming of the WIN device. **WIN\_USE** without a parameter will reset the name of WIN back to WIN.

syntax: **WIN\_USE** [ *name* ]

example: i. **WIN\_USE dos : LOAD dos2\_prog** {loads 'prog' from WIN2\_ }  
ii. **WIN\_USE** {and now its name is WIN again}  
iii. **WIN\_USE ram : DIR ram1\_** {displays directory of WIN1\_ }

## **WIN\_WP** directory devices

Write protects a WIN drive.

syntax: WIN\_WP *drive* [, *status* ]

example: i. **WIN\_WP 1** {write protect WIN1\_}  
ii. **WIN\_WP 1,0** {remove write protection from WIN1\_}

## **WMON**

### **WTV** windows

There are two commands for resetting the windows to the turn-on state.

**WMON** will reset the windows #0, #1, and #2 into 'Monitor' mode.

**WTV** will reset the windows #0, #1, and #2 into 'TV' mode.

A border has been added to window #0 to make it clearer where an SBASIC Job is on the screen.

Only the window sizes, positions and borders are reset by these commands, the paper strip and ink colours remain unchanged.

If you have a screen larger than 512x256 pixels, it is useful to be able to re-position the SBASIC windows. The **WMON** and **WTV** commands may take an extra pair of parameters: the pixel position of the top left hand corner of the windows. If only one extra parameter is given, this is taken to be both the x and y pixel positions.

If the mode is omitted, the mode is not changed, and, if possible, the contents are preserved and the outline (if defined) is moved.

syntax: *mode* := *numeric\_expression*  
*xpos* := *numeric\_expression*  
*ypos* := *numeric\_expression*

**WMON** *mode* [ , *xpos*, *ypos* ]

**WTV** *mode* [ , *xpos*, *ypos* ]

example: i. **WMON 4,50** {reset windows to standard monitor layout displaced 50 pixels to the right and 50 pixels down}  
ii. **WMON ,80,40** {reset windows to standard monitor layout displaced 80 pixels to the right and 40 pixels down, preserving the contents}



<b>A</b>		
ABS.....	3	
Absolute values.....	3	
ACOT.....	3	
ADATE.....	3	
ALARM.....	3	
ALCHP.....	4	
ALTKEY.....	4	
AJOB.....	4	
ARC		
cotangent.....	5	
tangent.....	5	
ARC_R.....	5	
Arctangent.....	5	
Arguments.....	28, 29	
Arrays		
DIM.....	33	
DIMN.....	33	
Assignment.....	69	
AT.....	5	
ATAN.....	3	
AUTO.....	6	
<b>B</b>		
BAUD.....	6	
Baudrates.....	6	
BEEP.....	7	
BEEPING.....	7	
BGCOLOUR_QL.....	8	
BGCOLOUR_24.....	8	
BGET.....	8	
BGIMAGE.....	9	
BIN.....	9	
BIN\$.....	9	
BLOCK.....	9	
BORDER.....	10	
BPUT.....	8	
<b>C</b>		
CACHE_OFF.....	10	
CACHE_ON.....	10	
CALL.....	11	
CDEC\$.....	45	
CD_ALLTIME.....	11	
CD_CLOSE.....	11	
CD_EJECT.....	11	
CD_FIRSTTRACK.....	11	
CD_HOUR.....	12	
CD_HSG2RED.....	12	
CD_INIT.....	12	
CD_ISPLAYING.....	12	
CD_ISCLOSED.....	12	
CD_ISINSERTED.....	12	
CD_ISPAUSED.....	12	
CD_LASTTRACK.....	11	
CD_LENGTH.....	13	
CD_MINUTE.....	12	
CD_PLAY.....	13	
CD_RED2HSG.....	12	
CD_RESUME.....	13	
CD_SECOND.....	12	
CD_STOP.....	13	
CD_TRACK.....	13	
CD_TRACKLENGTH.....	13	
CD_TRACKTIME.....	14	
CD_TRACKSTART.....	14	
Channel		
CLOSE.....	18	
Character		
CODE.....	18	
repetition.....	64	
size.....	23	
CHAR_DEF.....	14	
CHAR_INC.....	15	
CHAR_USE.....	15	
CHK_HEAP.....	15	
CHR\$.....	16	
CIRCLE.....	16	
CIRCLE_R.....	16	
CKEYOFF.....	16	
CKEYON.....	16	
CLCHP.....	17	
CLEAR.....	17	
BASIC.....	17	
screen.....	18	
window.....	18	
CLOCK.....	17	
Clock		
ADATE.....	3	
DATE.....	26	
DATE\$.....	26	
DAY\$.....	26	
SDATE.....	103	
CLOSE.....	18	
Closing		
channels.....	18	
CLS.....	18	
CODE.....	18	
Colour		
INK.....	63	
MODE.....	74	
PAPER.....	80	
RECOL.....	95	
recolour.....	95	
COLOUR_NATIVE.....	19	
COLOUR_PAL.....	19	
COLOUR_QL.....	19	
COLOUR_24.....	19	
Comments.....	96	
Communications		
baud rates.....	6	
networks.....	75	
Conditions		
IF.....	61	
SElect.....	103	
CONTINUE.....	19	
COPY.....	20	
COPY_H.....	20	
COPY_N.....	20	
COPY_O.....	20	
COS.....	22	
cosine.....	22	
COT.....	22	
cotangent.....	22	
CSIZE.....	23	
CURDIS.....	23	
CURSEN.....	23	
CURSOR.....	24	

<b>D</b>	
DATA.....	24
structures.....	33
DATA_USE.....	25
DATA\$.....	25
DATE.....	26
DATE\$.....	26
DEFine.....	28, 29
DAY\$.....	26
DDOWN.....	27
DNEXT.....	27
DEFine	
FuNction.....	28
PROCedure.....	29
DEG.....	30
Degrees.....	30
Delay.....	84
DELETE.....	30
files.....	30
lines.....	36
DEL_DEFB.....	30
DESTD\$.....	25
DEST_USE.....	31
Devices	
CLOSE.....	18
directory.....	34
FOPEN.....	49
FOP_DIR.....	49
FOP_IN.....	49
FOP_NEW.....	49
FOP_OVER.....	49
LBYTES.....	69
LOAD.....	71
load and run.....	72
LRUN.....	72
MERGE.....	73
merge and run.....	75
MRUN.....	75
NET.....	75
network station.....	75
OPEN.....	77
OPEN_IN.....	77
open for input.....	77
OPEN_NEW.....	77
open new.....	77
OPEN_OVER.....	77
OPEN_DIR.....	77
RUN.....	99
SAVE.....	99
SBYTES.....	101
DEVTYPE.....	31
DEV_LIST.....	31
DEV_NEXT.....	32
DEV_NEXT\$.....	31
DEV_USE.....	32
DEV_USE\$.....	31
DEV_USEN.....	32
DIM.....	33
Dimension arrays.....	33
DIMN.....	33
DIR.....	34
Directory.....	34

Directory devices	
COPY.....	20
.....	20
copying.....	20
DELETE.....	30
deleting files.....	30
FORMAT.....	50
LOAD.....	71
loading SBASIC programs.....	71
SAVE.....	99
saving SBASIC programs.....	99
Display directory.....	34
DISP_BLANK.....	35
DISP_COLOUR.....	35
DISP_INVERSE.....	35
DISP_RATE.....	35
DISP_SIZE.....	35
DISP_TYPE.....	35
DIV.....	35
DLINE.....	36
DLIST.....	36
DMEDIUM_DENSITY.....	37
DMEDIUM_DRIVE\$.....	37
DMEDIUM_FORMAT.....	37
DMEDIUM_FREE.....	37
DMEDIUM_NAMES\$.....	37
DMEDIUM_RDONLY.....	37
DMEDIUM_REMOVE.....	37
DMEDIUM_TOTAL.....	37
DMEDIUM_TYPE.....	37
DO.....	38
Documentation.....	95
DOS_DRIVE.....	38
DOS_DRIVES\$.....	38
DOS_USE.....	38
Dots.....	86

<b>E</b>	
ED.....	39
EDIT.....	39
ELLIPSE.....	16
ELLIPSE_R.....	16
END	
DEFINE.....	28, 29
FOR.....	49
IF.....	61
REPeat.....	97
SELection.....	103
WHEN.....	114
EOF.....	40
EOFW.....	40
Equals.....	69
Errors	
CONTINUE.....	19
RETRY.....	19
EPROM_LOAD.....	41
ERLIN.....	41
ERT.....	41
ET.....	42
EW.....	42
EX.....	42
EXEC.....	42
EXEC_W.....	42
EXEP.....	44
EXIT.....	44

with FOR.....	49
with REPEAT.....	97
EXP.....	44
Exponentiation.....	44
EXTRAS.....	45

## F

FBKDT.....	47
FDAT.....	47
FDEC\$.....	45
Files	
COPY.....	20
COPY_H.....	20
COPY_N.....	20
COPY_O.....	20
DELETE.....	30
DIR.....	34
directory.....	34
FOPEN.....	49
FOP_DIR.....	49
FOP_IN.....	49
FOP_NEW.....	49
FOP_OVER.....	49
LBYTES.....	69
LOAD.....	71
load and run.....	72
LRUN.....	72
MERGE.....	73
merge and run.....	75
MRUN.....	75
OPEN.....	77
open for input.....	77
OPEN_IN.....	77
open new.....	77
OPEN_NEW.....	77
OPEN_OVER.....	77
OPEN_DIR.....	77
PRINT.....	88
RUN.....	99
SAVE.....	99
FILL.....	46
FILL\$.....	46
FLASH.....	46
FLEN.....	47
FLP_DENSITY.....	47
FLP_SEC.....	48
FLP_START.....	48
FLP_STEP.....	48
FLP_TRACK.....	48
FLP_USE.....	48
FLUSH.....	48
FMAKE_DIR.....	73
FN.....	28
FNAME\$.....	47
FOPEN.....	49
FOP_DIR.....	49
FOP_IN.....	49
FOP_NEW.....	49
FOP_OVER.....	49
FOR.....	49
with EXIT.....	49
with NEXT.....	49
FORMAT.....	50
FPOS.....	51

FREE_MEM.....	51
FTEST.....	51
FTYP.....	47
FUPDT.....	47
FuNction.....	28
DEFine.....	28, 29
RETurn.....	97
FVERS.....	47
FXTRA.....	47

## G

GET.....	52
GOSUB.....	52
GOTO.....	53
Graphics	
ARC.....	5
ARC_R.....	5
CIRCLE.....	17
CIRCLE_R.....	16
ELLIPSE.....	16
ELLIPSE_R.....	16
FILL.....	46
fill shape.....	46
LINE.....	70
LINE_R.....	70
POINT.....	85
POINT_R.....	85
SCALE.....	101
Turtle	
FILL.....	46
MOVE.....	74
TURN.....	112
TURNT0.....	112
SCALE.....	101
PENDOWN.....	85
PENUP.....	85

## H

HEX.....	53
HEX\$.....	53
HGET.....	53
HPUT.....	53
Highlighting.....	110
HOT_CHP.....	54
HOT_CHP1.....	54
HOT_CMD.....	55
HOT_DO.....	55
HOT_GETSTUFF\$.....	56
HOT_GO.....	56
HOT_KEY.....	56
HOT_LIST.....	56
HOT_LOAD.....	57
HOT_LOAD1.....	57
HOT_NAME\$.....	57
HOT_OFF.....	58
HOT_PICK.....	58
HOT_REMV.....	59
HOT_RES.....	54
HOT_RES1.....	54
HOT_SET.....	58
HOT_STOP.....	56
HOT_STUFF.....	59
HOT_THING.....	59
HOT_THING1.....	59

HOT_TYPE.....	60
HOT_WAKE.....	61
<b>I</b>	
IDECS.....	45
I/O	
INKEY\$.....	63
keyboard input.....	67
KEYROW.....	67
printing.....	88, 89
IF.....	61
nesting.....	62
INK.....	63
INKEY\$.....	63
INPUT.....	64
IO_PRIORITY.....	64
INSTR.....	65
INSTR_CASE.....	65
INT.....	65
Integer divide.....	35
<b>J</b>	
JOBS.....	65
JOB\$.....	65
JOB_NAME.....	66
Job control	
AJOB.....	4
JOBS.....	65
JOB\$.....	65
JOB_NAME.....	66
RJOB.....	97
OJOB.....	66
NXJOB.....	66
PJOB.....	66
SPJOB.....	108
Jump.....	53
<b>K</b>	
Keyboard input.....	63, 67
KEYROW.....	67
<b>L</b>	
LANGUAGE.....	68
LANGUAGE\$.....	68
LANG_USE.....	68
LBYTES.....	69
LEN.....	69
Length of strings.....	69
LET.....	69
LGET.....	8
LINE.....	70
delete.....	36
DLINE.....	36
editor.....	39
numbering.....	6
renumbering.....	96
RENUM.....	96
LINE_R.....	70
LIST.....	70
LN.....	71
LOAD.....	71
load and run.....	72
LPUT.....	8

LRESPR.....	72
LRUN.....	72
Local variables.....	
in functions.....	28
in procedures.....	29
Logarithm.....	71
LOG10.....	71
Loop epilogue.....	49
Loop repetition	
EXIT.....	44
FOR.....	49
NEXT.....	76
REPEAT.....	95
<b>M</b>	
MACHINE.....	72
Machine code.....	
CALL.....	11
SEXEC.....	107
SEXEC_O.....	107
saving.....	107
ET.....	42
EW.....	42
EX.....	42
EXEC.....	42
EXEC_W.....	42
EXEP.....	44
loading.....	42
RESPR.....	98
Maths functionsABS.....	
.....	3
absolute value.....	3
ACOT.....	3
arc cotangent.....	3
ATAN.....	3
Arc tangent.....	3
common logarithm.....	71
COS.....	22
cosine.....	22
COT.....	22
EXP.....	44
exponentiation.....	44
INT.....	65
integer part.....	65
LOG.....	71
LN.....	71
natural logarithm.....	71
RAD.....	95
radians conversion.....	95
SIN.....	107
sine.....	107
SQRT.....	109
square root.....	109
TAN.....	110
tangent.....	110
Merge and run.....	75
MAKE_DIR.....	73
MERGE.....	73
MOD.....	73
MODE.....	74
modulus.....	73
MOUSE_SPEED.....	74
MOUSE_STUFF.....	74
MOVE.....	74

MRUN.....	75
Multitasking	
PAUSE.....	84
SEXEC.....	107
SEXEC_O.....	107

## N

NET.....	75
Networks.....	75
NEW.....	75
NEXT.....	76
with FOR.....	49
with REPeat.....	97
Restarting SBASIC .....	75
NXJOB.....	66

## O

OJOB.....	66
ON GOSUB.....	76
ON GOTO.....	76
OPEN.....	77
channel.....	77
serial port.....	77
window.....	77
OPEN_IN.....	77
open existing file.....	77
OPEN_OVER.....	77
OPEN_DIR.....	77
OPEN_NEW.....	77
open new file.....	77
Operators	
INSTR.....	65
MOD.....	73
OUTLN.....	77
OVER.....	78
overprinting.....	78

## P

PALLETTE.....	79
PAN.....	80
PAPER.....	80
Parameters.....	28,29
PARNAME\$.....	81
PARSTR\$.....	82
PARTYP.....	81
PAR_ABORT.....	82
PAR_BUFF.....	82
PAR_CLEAR.....	82
PAR_DEFAULTPRINTER\$.....	82
PAR_GETFILTER.....	82
PAR_GETPRINTER\$.....	82
PAR_PRINTERCOUNT.....	83
PAR_PRINTERNAME\$.....	83
PAR_PULSE.....	83
PAR_SETFILTER.....	82
PAR_SETPRINTER.....	83
PAR_USE.....	83
PAR_WAIT.....	83
PAUSE.....	84
PEEK.....	84
PEEK\$.....	85
PEEK_L.....	84

PEEK_W.....	84
PEEK\$.....	85
PENDOWN.....	85
PENUP.....	85
PJOB.....	66
PI.....	85
Plotting points.....	86
POINT.....	86
POINT_R.....	86
POKE.....	86
POKES.....	87
POKES\$.....	87
POKE\$.....	87
POKE_L.....	86
POKES_L.....	87
POKE_W.....	86
POKES_W.....	87
PRINT.....	88
OVER.....	78
UNDER.....	112
Printout.....	88
PRINT_USING.....	89
PROCESSOR.....	90
Procedures	
DEFine.....	28, 29
LOCal.....	71
RETurn .....	98
Programs	
CONTINUE.....	19
RETRY.....	19
RUN.....	99
SAVE.....	99
PROG\$.....	25
PROG_USE.....	90
PROT_DATE.....	91
PROT_MEM.....	91
PRT_ABORT.....	91
PRT_BUFF.....	91
PRT_CLEAR.....	91
PRT_USE.....	92
PRT_USE\$.....	92
PUT.....	52

## Q

QLOAD.....	71
QMERGE.....	73
QMRUN.....	75
QPC_CMDLINE\$.....	92
QPC_EXEC.....	92
QPC_EXIT.....	92
QPC_HOSTOS.....	93
QPC_MAXIMIZE.....	93
QPC_MINIMIZE.....	93
QPC_MSPEED.....	93
QPC_NETNAME\$.....	93
QPC_QLSCREMU.....	93
QPC_RESTORE.....	93
QPC_SYNCSCRAP.....	94
QPC_VER\$.....	94
QPC_WINDOWTITLE.....	94
QSAVE.....	99
QSAVE_O.....	99
QUIT.....	94

## R

RAD.....	95
RAM_USE.....	95
Random numbers.....	99
RANDOMISE.....	95
READ.....	24
RECHP.....	4
RECOL.....	95
REM .....	96
REMark.....	96
RENAME.....	96
RENUM.....	96
Renumber lines.....	96
REPeat.....	97
EXIT.....	44
NEXT.....	76
REPORT.....	97
Repetition	
FOR.....	49
NEXT.....	76
RESET.....	97
Reset clock.....	3, 103
Resolution.....	74
RESPR.....	98
RESTORE.....	24
RETRY.....	19
RETurn.....	98
with FuNction.....	28
with PROCedures.....	29
RJOB.....	98
RND.....	99
Routines.....	29
RS-232-C.....	6
RUN.....	99
LRUN.....	72
load and run.....	72

## S

SAVE.....	99
SAVE_O.....	99
machine code .....	101
programs.....	99
SBASIC.....	100
SBYTES.....	101
SBYTES_O.....	100
SCALE.....	101
Screen	
BLOCK.....	9
BORDER.....	10
character size.....	23
clear.....	18
FLASH .....	46
INK.....	63
MODE.....	74
output.....	87
OVER.....	78
overprinting.....	78
PAN.....	80
PAPER.....	80
PRINT.....	88
RECOL.....	95
recolouring.....	95
SCROLL.....	102

STRIP.....	110
UNDER.....	112
underlining.....	112
WINDOW.....	115
SCROLL.....	102
SCR_BASE.....	102
SCR_LLEN.....	102
SCR_XLIM.....	102
SCR_YLIM.....	102
SDATE.....	103
SElect.....	103
Setting clock.....	103
Setting station number.....	75
Shapes	
ARC.....	5
CIRCLE.....	16
ELLIPSE.....	16
FILL.....	46
LINE.....	70
SEND_EVENT.....	104
SER_BUFF.....	104
SER_CDEOF.....	104
SER_CLEAR.....	105
SER_FLOW.....	105
SER_GETPORT.....	105
SER_PAUSE.....	105
SER_ROOM.....	106
SER_SETPORT.....	105
SER_USE.....	106
SET_FBKDT.....	106
SET_FUPDT.....	106
SET_FVERS.....	106
SEXEC.....	107
SEXEC_O.....	107
SIN.....	107
sine.....	107
Size of characters.....	23
Sound	
BEEP.....	7
BEEPING.....	7
SLUG.....	107
SPJOB.....	108
SPL.....	108
SPLF.....	108
SPL_USE.....	109
SQRT.....	109
Square root.....	109
STAT.....	109
Starting programs .....	72, 99
Station number.....	75
STOP.....	110
Strings	
CHR\$.....	16
FILL\$.....	46
INSTR.....	65
LEN.....	69
length.....	69
STRIP.....	110
subroutines.....	28, 29

## T

TAN .....	110
Tangent.....	110
THEN.....	61

TH_FIX.....	110
Time	
ALARM.....	3
clock adjust.....	3
clock reset.....	103
date.....	26
PAUSE.....	84
PROT_DATE.....	91
TK2_EXT.....	110
TRA.....	111
TRUNCATE.....	111
TURN.....	112
TURNT0.....	112
Turtle graphics	
FILL.....	46
MOVE.....	74
PENUP.....	85
PENDOWN.....	85
TURN.....	112
TURNT0.....	112
<b>U</b>	
Unconditional jump.....	53
UNDER.....	112
Underlining.....	112
UPUT.....	8
<b>V</b>	
Value absolutes.....	3
VER\$.....	112
VIEW.....	113
<b>W</b>	
WAIT_EVENT.....	113
WCOPY.....	20
WDEL.....	30
WDIR.....	114
WGET.....	8
WHEN_ERROR.....	114
WIDTH.....	115

Windows	
AT.....	5
BLOCK.....	9
BORDER.....	10
CSIZE.....	23
Character size.....	23
clear.....	18
cursor control.....	5, 24
FILL.....	46
FLASH.....	46
INK.....	63
MODE.....	74
OVER.....	78
overprinting.....	78
PAN.....	80
PAPER.....	80
print position .....	5
SCROLL.....	102
STRIP.....	110
UNDER.....	112
underlining .....	112
WINDOW.....	115
WIN_DRIVE.....	115
WIN_DRIVE\$.....	115
WIN_FORMAT.....	116
WIN_REMV.....	116
WIN_SLUG.....	116
WIN_START.....	116
WIN_STOP.....	116
WIN_USE.....	116
WIN_WP.....	117
WMON.....	117
WTV.....	117
WPUT.....	8
WSTAT.....	114