Q68
Keywords

This Keyword Reference Guide lists all the Q68 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 Q68 manual.

ABS maths functions

ABS returns the absolute value of the parameter. It will return the value of the parameter if the parameter is positive and will return zero minus the value of the parameter if the parameter is negative.

syntax. **ABS(***numeric_expression***)**

example: i. **PRINT ABS(0.5)**
ii. **PRINT ABS(a-b)**

ACOS, ASIN
ACOT, ATAN maths functions

ACOS and **ASIN** will compute the arc cosine and the arc sine respectively. **ACOT** will calculate the arc cotangent and **ATAN** will calculate the arc tangent. There is no effective limit to the size of the parameter.

ATAN will provide a 4 quadrant result by taking two parameters. If x is greater than 0, **ATAN** (x,y) give the same results as ATAN (y/x). Otherwise it returns values in the other quadrants (>PI/2 and <-PI/2).

syntax: *angle:= numeric_expression* [in radians]

ACOS (*angle***)**	**ACOT (***angle***)**
ASIN (*angle***)**	**ATAN (***angle* [,*angle*]**)**

example: i. **PRINT ATAN(angle)**
ii. **PRINT ASIN(1)**
iii. **PRINT ACOT(3.6574)**
iv. **PRINT ATAN(a-b)**

ADATE clock

ADATE allows the clock to be adjusted.

Note that **ADATE** does not set the battery backed clock in the Q68. To set the battery backed real time clock. First set the date and time with **ADATE**, then execute the clock utility program named 'Q68SETRTC'

syntax: *seconds*:= *numeric_expression*

ADATE *seconds*

example: i. **ADATE 3600**	{advance the clock 1 hour}
ii. **ADATE -60**	{move the clock back 1 minute}

AJOB **job control**

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.

ALARM **timekeeping**

ALARM is a procedure to set up an alarm using the Q68's system clock.

The time should be specified as two numbers: hours (24 hour clock) and minutes.

syntax: *time*:= *numeric_expression* , *numeric_expression*
ALARM *time*

example: **ALARM 14,30** {alarm will sound at half past two}

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** back 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}

ALFM **memory management**

The function **ALFM** will allocate requested amount of memory from the 'Fast Memory' area and return the base address of the space.

If you try to reserve more memory than is available, then the function returns with an out of memory error and no memory will have been reserved.

Initially there is about 10K bytes of memory available.

syntax: *number_of_bytes*:= *numeric_expression*

ALFM (*number_of_bytes***)**

example: **base = ALFM (3000)** {allocate 3000 bytes from the fast memory}

note: Once memory has been allocated with **ALFM**, it cannot be de-allocated

warning: The system does NOT stop you from using more than the amount of memory that you requested, but sooner or later the system is likely to crash.

ALPHA_BLEND **graphics**

The **ALPHA_BLEND** command sets the transparency of shapes and text drawn to the screen, Allowing the underlying graphics and text to show through.

The level of the transparency may be set from 0, fully transparent. To 255, opaque

syntax: **ALPHA_BLEND** *numeric_expression*

example: **ALPHA_BLEND 128** {make output half transparent}

ALTKEY **console driver**

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.

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 co-ordinate* 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 last point plotted to the specified point turning through the specified angle

example:
i. **ARC 15,10 TO 40,40,PI/2**
{draw an are from 15,10 to 40,40 turning through PI/2 radians}
ii. **ARC TO 50,50,PI/2**
{draw an are 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 are, starting 10,10 from the last point plotted to 55,45 from the start of the are, 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 **SBASIC editor**

AUTO has been replaced by **ED**.

BAUD **communications**

BAUD sets the baud rate for communication via the serial channels. The speed of the channels may 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 the Q68:

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 1,9600**	{set SER1 to 9600 baud}
	ii. **BAUD print_speed**	{set SER1 to ‘print_speed’ baud}

BEEP sound

The Q68 tries to emulate the QL's **BEEP** command. This is far from perfect. The sound will often be too “clean”.

BEEP can accept a variable number of parameters to give various levels of control over the sound produced. However in the Q68, the “wrap”, “random” and “fuzziness” parameters of the **BEEP** command are simply ignored.

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}	ignored in the Q68
	fuzzy:= *numeric_expression*	{range 0..15}	ignored in the Q68
	random:= *numeric_expression*	{range 0..15}	ignored in the Q68

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 a 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 the Q68 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 windows, To one of the QL compatible colours, or to a plain true colour.

syntax: *colour* := *numeric_expression*

BGCOLOUR_QL *colour*	{range 0 … 255}
BGCOLOUR_24 *colour*	{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.

If the position pointer is a floating point variable, rather than an expression. Then, when all items have been read from, or written to the channel. The pointer will be updated to the new position.

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$(1 to 6)	{get 6 bytes from #3 into a$}
iv.	**WGET#4\ptr,a**	{ptr will be incremented by 2}
v.	**WGET#4\prt+4,a**	{ptr will not be incremented}

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 in the current screen mode. 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_YSIZE
```

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
WM_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*.

WM_BLOCK will fill a block using one of the Windows Manager colour palettes.

BLOCK and **WM_BLOCK** use *the pixel coordinate system*.

syntax:	*width*:=	*numeric_expression*	
	height:=	*numeric_expression*	
	x:=	*numeric_expression*	
	y:=	*numeric_expression*	
	wm_colour:=	*numeric_expression*	{range 0 … 65535}

BLOCK [*channel*,] *width*, *height*, *x*, *y*, *colour*
WM_BLOCK [*channel*,] *width*, *height*, *x*, *y*, *wm_colour*

example: i. **BLOCK 10,10,5,5,7** {10x10 pixel white block at 5,5}
ii. **WM_BLOCK #4,100, 10, 0, 0, $0202**
{100x10 block in window foreground colour}

BORDER
WM_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.

WM_BORDER acts as **BORDER** but will use one of the Windows Manager colour palettes.

syntax:	*width*:= *numeric_expression*	
	wm_colour:= *numeric_expression*	{range 0 … 65535}

BORDER [*channel*,] *width* [, *colour*]
WM_BORDER [*channel*,] *width*, *wm_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
```
iii. **WM_BORDER 4, $0216** {create an application window border}

CACHE_OFF
CACHE_ON **memory management**

Turns on or off the processor cache. Not used in the Q68.

CALL **machine code**

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 68000 data and address registers (D1 to D7, A0 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
```

CARD_CREATE **SD cards**

CARD_CREATE allows the creation of a file in the FAT32 file system of the first partition on a SD card. Remember, this partition must be a FAT32 partition. There must be an empty slot for this file within the first 16 entries in the root directory of this partition (which you can check with the **CARD_DIR$** command).

syntax: *card*:= *numeric_expression*	{1 or 2}
size:= *numeric_expression*	{in megabytes, max 16000}
file_name:= *filename*	{must obey the “8.3 rule”}

CARD_CREATE *card, size, file_name$*

example:	i. **CARD_CREATE 1,200,”test.win”**	{This will create a file called “TEST.WIN” on card 1, with a size of 200 Megabytes}
	ii. **10 CARD_CREATE 1,200,”test.win”**	{create empty file, 200MB in size}
	20 WIN_DRIVE 4,1,”test.win”	{point win4_ to it}
	30 WIN_FORMAT 4	{allow formatting of win4_}
	40 FORMAT win4_your_name	{format win4_}

note: Lines 30 and 40 MUST be made from job 0, the main SBasic job, not from a daughter job.

CARD_DIR$ **SD cards**

The **CARD_DIR$** function shows the first 16 entries in the FAT32 root directory of the first partition on an SD card. **CARD_DIR$** will return a large string with the 8.3 formatted names of the 16 entries, each separated by a linefeed (CHR$(10)).
These names may also be shown as “-- Empty --” which shows that the corresponding entry in the FAT32 root directory is empty, or “-Long name-” which shows that this entry does not point to a file but to a long name. The latter is NOT considered by SMSQ/E to be an empty entry in the directory.

syntax:	*card*:= *numeric_expression*	{1 or 2}
	CARD_DIR$(*card***)**	
example:	**PRINT CARD_DIR$(1)**	

CARD_INIT **SD cards**

CARD_INIT initialises the SD card reader in the Q68. Card1 is automatically initialised at boot time.

By design, card2 is not initialised at boot time, though this will depend on your configuration options. If it is not initialised, you have to initialise it yourself. You can do this with the **CARD_INIT** command. The card itself is not touched by this command (it is not formatted, written to or anything).

syntax:	*card*:= *numeric_expression*	{1 or 2}
	CARD_INIT *card*	
example:	**CARD_INIT** 2	

CARD_RENF **SD cards**

CARD_RENF allows you to rename a file already existing in the first 16 entries in the FAT32 root directory of the first partition on a SD card.

syntax:	*card*:= *numeric_expression*	{1 or 2}
	old_name:= *filename*	{must obey the “8.3 rule”}
	new_name:= *filename*	{must obey the “8.3 rule”}
	CARD_ RENF *card, ”old_name”, ”new_name”*	
example:	**CARD_RENF 1,"old.win","new.win"**	

CARD_SPEEDUP **SD cards**

The **CARD_SPEEDUP** function will identify if your Q68 supports fasted transfer speeds of 40MHz. **CARD_SPEEDUP** will return 1 if it can, or 0 if it cannot.

Normal transfers between the SD Card and the Q68 are done at 25Mhz, in compliance with the specifications for SD cards. It has been experimentally found that it is possible to speed up the transfer to 40Mhz.

Using the 40MHz speed takes the SD cards outside their operating specifications, and could damage the SD card, or corrupt the data stored on it. To avoid this being accidentally switched on, you must configure it in the SMSQ/E file, where you can switch this on individually for each drive.

syntax: **CARD_SPEEDUP**

example: **PRINT CARD_SPEEDUP**

note: Only later, or firmware updated Q68's support faster SD card operation.

warning: Activating this feature is done at your own risk.

CHAR_DEF **windows**

The Q68 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 Q68 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 Q68 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

Undocumented command.

Believed to be used to check whether the heap has become corrupted.

The SMSQ/E source code refers to it as a ‘heap checking patch’

CHR$ SBASIC

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_expression***)**

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 SBASIC
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 Q68's system clock. If no window is specified, then a default window is set up in the top right hand side 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,'Q68 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 the 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 SBASIC

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**, and **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 mdvl_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 **maths 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 C0S(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 C0T(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.

width	size	height	size
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**

CURSPRLOAD **window manager**
CURSPRLOAD will load a new system cursor sprite into memory ready to be activated by a **CURSPRON** command.

The cursor sprite must –

i. Have a size of 6 X 10.
ii. Set at position 36 in the system sprites.
iii. Showable in the current screen resolution.

If any of the above conditions are not met then a normal cursor will be shown.

syntax: **CURSPRLOAD** *device*

example: **CURSPRLOAD flp1_new_spr**

CURSPROFF
CURSPRON **window manager**
CURSPRON and **CURSPROFF** enable and disable the use of a sprite to replace the cursor in a window.

To use a new cursor sprite, it has to be first loaded into SBASIC with a **CURSPRLOAD** command.

syntax: *job_identifier*:= | *job_number* , *tag_number*
| *job_number* + (*tag_number* * 65536)
id:= *job_identifier*

CURSPRON *id*
CURSPROFF *id*

example:

i.	**10 CURSPRLOAD flp1_newCursor_spr**	{load new sprite}
	20 CURSPRON 0	{enable new sprite in job 0}
ii.	**CURSPRON “xchange”**	{enable new cursor in job ‘xchange’}
iii.	**CURSPROFF -1**	{sets this job to a normal cursor}

DATA
READ
RESTORE SBASIC

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*,l*

RESTORE restores the data pointer, i.e. the position from which subsequent **READ**s 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.

DATAD$
PROGD$
DESTD$ defaults functions

DATAD$**, **PROGD$, 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.

DATE$

DATE clock

DATE$** is a function which will return the date and time contained in the Q68's clock. The format of the string returned by **DATE$ is:

"*yyyy mmm dd hh*:*mm*:*ss*"

where	*yyyy*	is the year 2022, 2023, etc
	mmm	is the month Jan, Feb etc
	dd	is the day 01 to 28, 29, 30, 31
	hh	is the hour 00 to 23
	mm	are the minutes 00 to 59
	ss	are the seconds 00 to 59

DATE will return the date as a floating point number which can be used to store dates and times in a compact form.

If **DATE$** is used with a numeric parameter then the parameter will be interpreted as a date in floating point form and will be converted to a date string.

syntax:	**DATE$**	{get the time from the clock)
	DATE$ (*numeric_expression***)**	{get time from supplied parameter}
	DATE [**(***yyyy*,*m*,*d*,*h*,*m*,*s***)**]	

example: i. **PRINT DATE$** {output the date and time}
ii. **PRINT DATE$(234567)** {convert 234567 to a date}
iii. **PRINT DATE** {output today's date as a floating point number}
iv. **PRINT DATE (2002,7,23,10,32,15)**
{output 23rd July 2002 at 10:32:15 as a floating point number}

DAY$ clock

DAY$** is a function which will return the current day of the week. If a parameter is specified then **DAY$ will interpret the parameter as a date and will return the corresponding day of the week.

syntax:	**DAY$**	{get day from clock}
	DAY$ (*numeric_expression***)**	{get day from supplied parameter}

example: i. **PRINT DAY$** {output the day}
ii. **PRINT DAY$(234567)** {output the day represented by 234567 (seconds)}

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
DDOWN john	flp2_john_	flp1_	ser
DNEXT fred	flp2_fred_	flp1_	ser
PROG_USE flp2_fred	flp2_fred_	flp2_fred_	ser
DNEXT john	flp2_john_	flp2_john_	ser
DUP	flp2_	flp2_	ser
DEST_USE flp1	flp2_	flp2_	flp1_
DDOWN john	flp2_john_	flp2_john_	flp1_john_
SPL_USE 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 [*identifier type*]

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 mean
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 [*identifier*]

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 O : 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 slow_scroll
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
FDEL 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.

The **FDEL** function will return 0 upon successfully deleting a file, or -9 for an 'In use' error, should the specified file be open.

If the file is not found, then **FDEL** will return 0.

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

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}
v. **err=FDEL(win1_memo_txt)**	{err will be set to -9 if the file is open}

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:

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

DEL_DEFB memory management

DEL_DEFB will delete file definition blocks from the 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$ 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.

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_}

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.

OPEN#3,dev1_f1	Opens "ram1_f1"
OPEN#3,dev2_bankmanager	Opens "flp1_letters_bankmanager"
OPEN#3,dev3_f1	Opens "win1_work_newf1"
DELETE dev3__junk	Deletes "win1_work_new_junk"
LOAD dev4_prog_bas	Tries "ram2_prog_bas", then "flp1_latest_ prog_bas", and then finally "win1_work_prog_ bas"
LOAD dev5_DiskCheck	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.

DEV_USE 1,flp2_myprogs_	"dev1_" is "myprogs_ "on drive 2}
DEV_USE 2,flp1_ex_,1	"dev2_" is "flp1_ex_", or "flp2_myprogs_"
DEV_USE flp	"flp1_ "is now really "flp2_myprogs_and "flp2_" is "flp1_ex_"}
DEV_USE	"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*
dimension:= *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 the Q68.

DISP_COLOUR

DISP_COLOUR has no effect in the Q68.

DISP_INVERSE

DISP_INVERSE has no effect in the Q68.

DISP_MODE **graphics device 2**

DISP_MODE sets the Q68's display mode. There are 8 different screen modes presenting different screen sizes and colours.

The available Q68 screen modes are:

Mode 0	QL 8 colour mode The standard QL 256 x 256 pixels mode in 8 colours. In this mode you can also set mode 4, with the usual **MODE** keyword. This is then equivalent to setting **DISP_MODE 1**.
Mode 1	QL 4 colour mode The standard QL 512 x 256 pixels mode in 4 colours. In this mode you can also set mode 8, with the usual **MODE** keyword. This is then equivalent to setting **DISP_MODE 0**.
Mode 2	Small 16 bit mode 512 x 256 pixels in mode 33, with 16 Million colours (16 bits per pixel).
Mode 3	Large 16 bit mode 1024 x 512 pixels in mode 33, with 16 Million colours (16 bits per pixel). Please note that this mode will slow down the Q68, you should not use this mode when doing something time-critical.
Mode 4	Large QL Mode 4 1024 x 768 pixels in QL 4 colours mode (there is no mode 8 in this display mode).
Mode 5	Aurora compatible 8 bit colours 1024 x 768 pixels in Aurora 256 colours mode. This allows you to have a big screen with nicer colours while still being reasonably fast (but slower than the QL modes).
Mode 6	Medium 16 bit mode 512 x 384 pixels in mode 33, with 16 Million colours (16 bits per pixel).
Mode 7	Very large 16 bit mode 1024 x 768 pixels in mode 33, with 16 Million colours (16 bits per pixel). Please note that this mode will severely slow down the Q68, you should not use this mode when doing something time-critical.

syntax:	*mode*:= *numeric_expression*	
	DISP_MODE *mode*	{0 to 7}
example:	**DISP_MODE 1**	{sets a 512 x 256 QL mode 4 screen display}

note: The more colours that are displayed, and the higher the resolution. The slower the Q68 will become.

DISP_RATE

DISP_RATE has no effect in the Q68.

DISP_SIZE

DISP_SIZE has no effect in the Q68.

DISP_TYPE

DISP_TYPE will return a value indicating the type of display mode you are using.

0	– QL Colours display MODE 4	**DISP_MODE**'s	1, and 4
8	– QL Colours display MODE 8	**DISP_MODE**	0
16	– 8 bit Colour display (256 colour) mode	**DISP_MODE**	5
33	– High Colour 16-bit colour mode	**DISP_MODE**'s	2,3,5, and 7

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).

DMEDIUM_NAME$	Returns the medium name of the specified device.
DMEDIUM_DRIVE$	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 **RAM_USE**, **FLP _USE**, **WIN_USE** etc. This function also allows you to discover the "real" device which may be hidden behind "DEV".
DMEDIUM_RDONLY	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.
DMEDIUM_REMOVE	Returns 1 if the specified device is a removable hard disk.
DMEDIUM_DENSITY	Returns the density: 1=DD, 2=HD etc. RAM-Disks return -1, as they have no density.
DMEDIUM_FORMAT	Returns the logical format of the medium or partition: 1=QDOS/SMSQ, 2=DOS/TOS.
DMEDIUM_TYPE	Returns information about the physical drive: 0=RAM-Disk, 1=Floppy Disk, 2=Hard disk, 3=CD-ROM.
DMEDIUM_TOTAL	Returns the total number of free sectors (in 512 bytes sectors).
DMEDIUM_FREE	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(flp2_)**
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.

ED
EDIT

ED is a window based editor for editing SBASIC programs which are already loaded into the Q68.

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:

TAB	tab right (columns of 8)
SHIFT TAB	tab left (columns of 8)
ENTER	accept line and create a new line
ESC	escape - undo changes or return to SBASIC
up arrow	move cursor up a line
down arrow	move cursor down a line
ALT up arrow	scroll up a line (the screen moves down!)
ALT down arrow	scroll down a line (the screen moves up!)
SHIFT up arrow	scroll up one page
SHIFT down arrow	scroll down one page
left arrow	move cursor left one character
right arrow	move cursor right one character
SHIFT left arrow	move cursor left one word
SHIFT right arrow	move cursor right one word
ALT left arrow	move to start of line
ALT right arrow	move to end of line
CTRL left arrow	delete character to left of cursor
CTRL right arrow	delete character under cursor
CTRL SHIFT left arrow	delete word to left of cursor
CTRL SHIFT right arrow	delete word to right of cursor
CTRL ALT left arrow	delete line to left of cursor
CTRL ALT right arrow	delete line to right of cursor

CTRL down arrow	delete whole line
F9 or **SHIFT F4**	change between overwrite and insert mode
F10 or **SHIFT F5**	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 **HOT_GO** to activate)

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

EOF
EOFW devices

EOF and **EOFW** are functions 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.

If an end of file condition cannot be determined immediately, **EOF** will wait a certain amount of time before returning. **EOFW** will wait indefinitely.

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

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

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 are required 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 the Q68 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
ERNUM **error handling**

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

ERNUM is a function that will return the error number.

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

syntax:	**ERLIN**
	ERNUM

example:	i.	**PRINT ERLIN**
	ii.	**last_error = ERNUM**

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, EX_M multitasking

EX, **EW** and **EX_M** 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.

EX_M behaves like **EX** in that the calling job continues executing, But the job created is owned by the calling job. This means that if you kill the calling job, you will also kill the created job.

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*]
EW *program* [*,*file* *] [;*parameters*]
ET *program* [*,*file* *] [;*parameters*]
EX_M *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 the Q68 it is possible to have a chain of co-operating jobs engaged in processing the same data in a form of a 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 strings 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 the Q68.

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.

Executing a SBASIC program

If you execute a SBASIC program that ends in **_bas,** It will be loaded and started in a new daughter SBASIC job.

```
EXEC a_basic_program_bas[;”cmd_string”]
```

Note that no channels #0,#1,or #2 are initially opened in the new SBASIC job, and must be opened specifically if required. Otherwise any commands which try to use any of these channels will cause #0 to be opened as a small window in the center of the SBASIC job.
Once this channel has been opened, then #1 and #2 will also use this channel.

The optional **cmd_string** will be passed to a variable named **CMD$** in the new daughter SBASIC.

EXEP hotkey system

EXEP is a supplement to the **EXEC** (or **EX**) command. It has all the options of the **HOT_RES**, **HOT_CHP**, **HOT_LOAD** and **HOT_THING** functions. It does not set up a Hotkey but executes a program directly, either from an Executable Thing, or from a file.

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 there own code.

syntax: *params*:= *string* {list of parameters for individual programs}
options:= [I,] U
| G [*width*, *height*, *xorg*, *yorg*]
| P [*memory*] {in kilobytes}
| F

EXEP *filename* [;*params*] [,*jobname*] [,*options*] **)**
EXEP *thingname* [;*params*] [,*jobname*] [,*options*] **)**

example: i. **EXEP Quill,p,40** {execute Quill in 40 kbytes}
ii. **EXEP Capsclock,u** {execute capslock in unlockable window}
iii. **EXEP SBASIC;"lrun 'win2_program_bas'"**
{starts an SBASIC daughter job and sends the string 'lrun win2_program_bas' to #0 of the SBASIC job}

EXIT repetition

EXIT will continue processing after the **END** of the named **FOR** or **REPeat** structure.

syntax: **EXIT** *identifier*

example: i.
```
100 REM start Looping
110   LET count = 0
120 REPeat Loop
130  LET count = count +1
140  PRINT count
150  IF count = 20 THEN EXIT Loop
160 END REPeat loop
```
{the loop will be exited when count becomes equal to 20}

ii.
```
100 FOR n =1 TO 1000
110   REM program statements
120   REM program statements
130   IF RND >.5 THEN EXIT n
140 END FOR n
```
{the loop will be exited when a random number greater than 0.5 is generated}

EXP **maths functions**

EXP will return the value of e raised to the power of the specified parameter.

syntax: **EXP (***numeric_expression***)** {range -500..500}

example: i. **PRINT EXP(3**)
ii. **PRINT EXP(3.141592654)**

EXTRAS

EXTRAS will output to the specified or default channel, a list of commands and functions available to SBASIC

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

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

FAT_DRIVE

FAT_DRIVE allows you to set any FAT drive to the SD card and the partition it is to be found on.

syntax: *drive*:= *numeric_expression* {range 1 to 8}
card:= *numeric_expression* {range 1 to 2}
partition:= *numeric_expression* {range 1 to 4}

FAT_DRIVE *drive, card, partition*

example: i. **FAT_DRIVE 1,1,2** {sets FAT1_ to be partition 2 of card 1}
ii. **FAT_DRIVE 4,2,3** {sets FAT4_ to be partition 3 of card 2}

note: Only FAT16 partitions on a SD card can be used for FAT drives.

FAT_DRIVE$

The **FAT_DRIVE$** function will return the card and partition for which a FAT drive is configured.

The string returned by **FAT_DRIVE$** will be formatted as follows:
Card: <card_number>, Partition: <partition_number>

For example **PRINT FAT_DRIVE$**(1) might return "Card: 1, Partition: 2".

If the drive isn't configured for any card, the card number will be "N" (for "None"). If the drive isn't configured for any partition, the partition number will be 0.

syntax: *drive*:= *numeric_expression* {range 1 to 8}

FAT_DRIVE$ (*drive*)

example: i. **PRINT FAT_DRIVE$(1)**
ii. **a$=FAT_DRIVE$(4)**

FAT_USE **directory devices**

FAT_USE allows renaming of the FAT device. **FAT_USE** without a parameter will reset the name of FAT back to FAT.

syntax: **FAT_USE** [*name*]

example:	i. **FAT _USE flp : LOAD flp2_prog**	{loads 'prog' from FAT2_ }
	ii. **FAT _USE**	{and now its name is FAT again}
	iii. **FAT_USE ram : DIR ram1_**	{displays directory of FAT1_}

FAT_WP

FAT_WP sets the write protection on a FAT device.

syntax: *drive*:= *numeric expression*
flag:= *numeric expression* {0 or 1}

FAT_WP *drive*, *flag*

example:	i. **FAT_WP 1,1**	{set write protect for the drive accessed by FAT1}
	ii. **FAT_WP 1,0**	{clear write protect for the drive accessed by FAT1}

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.

FEP, FET
FEW, FEX
FEX_M, EXF **multitasking**

FEP, **FET**, **FEW**, **FEX** and **FEX_M** will load a sequence of programs and execute them in parallel and return the ID of the job which is created.

This ID can be used to manipulate the job in various ways by using the other job control commands.

These commands perform the same functions as the commands **EXEP**, **ET**, **EW**, **EXEC_W**, **EX**, and **EXEC**. But they also return the job ID of the created job. Except for the FEW command which returns the error code, returned by the (first) job.

FEX_M behaves like **FEX** in that the calling job continues executing, But the job created is owned by the calling job. This means that if you kill the calling job, you will also kill the created job.

EXF is functionally equivalent to **FEX**. It is included as **FEX** may clash with the **FEX** keyword contained in a commercial application named FileInfo II.

```
syntax:   program:= device
          parameters:= string_expression
          file:=  filename, or channel_number
          options:=     [ I,]  U
                             | G [ width, height, xorg, yorg ]
                             | P  [ memory ]       {in kilobytes}
                             | F


          FEP ( filename [;parameters] [;jobname] [,options] )
          FEP ( thingname [;parameters] [;jobname] [,options] )
          FET ( program [ *,file * ] [;parameters] )
          FEW ( program [ *,file * ] [;parameters] )
          FEX ( program [ *,file * ] [;parameters] )
          FEX_M ( program [ *,file * ] [;parameters] )
          EXF ( program [ *,file * ] [;parameters] )

example:  i.  PRINT FEP (flp1_Quill,p,40)      {print job number for flp1_Quill in 40k bytes}
          ii  PRINT FET (win1_Clock_exe)     {print job number for win1_clock_exe}
```

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'}
```

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. **FLASH** is only effective in the *low resolution* mode of **DISP_MODE 0**. **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.

note: At this time, **FLASH** does not appear to be implemented in SMSQ/E.
However it does work in MODE 8 with Minerva4Q68.

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}

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.

The specified device is the drive (physical or virtual) to be used for formatting and an identifier part used as the medium or volume name for the drive, 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 to the default or on the specified channel.

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

For WIN drives, SMSQ/E has a two-level protection scheme to prevent accidental formatting of WIN drives. THE command **WIN_FORMAT** must first be used from the first console window of job 0, (the first SBASIC) Followed by the **FORMAT** command. You must then type in the two characters that are displayed on the screen before the format will commence.

In the Q68, you can only format an existing container file. You cannot create a new container file on the card with the **FORMAT** command (you can do this with the **CARD_CREATE** command).

syntax: *device*:= *device_name* **|** *name*
| *number*

FORMAT [*channel*,] *device*

example:	i. **FORMAT fat1_data_disk**	
	ii. **FORMAT ram2_20**	{format RAM2_ to 10K bytes}
	iii. **WIN_FORMAT 2**	{allow WIN2_ to be formatted}
	FORMAT win2_backup	
	iv. **FORMAT fat2_costs**	
	v. **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)}
```

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**

FREE_FMEM memory management

The function **FREE_FMEM** will return the amount of free memory available in the ‘Fast Memory’ area.

syntax: **FREE_FMEM**

example: **PRINT FREE_FMEM**

note: In a freshly booted system **FREE_FMEM** should return around 10 Kilobytes.

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 IO, 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$='Smith'
30 PUT #3\fpoint, wally%, salary, name$
```

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 1k Byte data space.

HOME_CSET, HOME_CUR$, HOME_DEF, HOME_DIR$, HOME_FILE$, HOME_SET, HOME_VER$

This is a set of commands and functions for controlling the Home Thing.

HOME_CSET Sets the current directory for the job indicated. The job ID is optional, in that case -1 (meaning the current job), will be assumed if no job_ID is given.

HOME_CURR$ This function returns the current directory for the job given as job_id. The job ID is optional, in that case -1, meaning the current job, will be assumed.

HOME_DEF This sets a default filename for a job with the name given as the first parameter. This is useful for "executable things", where no filename is readily available, or for file managers that haven't integrated calls to the home thing. With this keyword, you set up the default job name and filename that is to be used for the home/current file/dir.
Please note that the file_name$ parameter must indeed be a FILENAME, not a directory name.

HOME_DIR$ This function returns the home directory for the job given as job_id. The job ID is optional, in that case -1,meaning the current job, will be assumed. To avoid programs stopping with an error if the home directory cannot be found for some reason, this function returns an empty string if that error happens.

HOME_FILE$ This function returns the home filename for the job given as job_id. The job ID is optional, in that case -1,meaning the current job, will be assumed.

HOME_SET Normally, jobs should not try to set up a home directory for themselves. This should be left to the system/filemanager. When a job is started with EX, EW or any of the similar commands, this is done automatically. However, filemanager writers may be interested in this information.
The **HOME_SET** command can be used to set the home directory, home filename and current directory. You pass the thing the job ID of the job for which this is to be set up and the entire filename, including the device and directory. The thing extracts the home directory from the filename. If you want to set up the home directory for the current job, you may pass -1 as parameter.
Since there can only be one home directory for a job and since that can only be defined once, the keyword will give an ‘in use’ error if the home directory is already set for this job. Otherwise, this keyword will set the home directory, the home file and the current directory.
This keyword exists mainly for testing purposes.

HOME_VER$ This function returns the version number of the HOME thing.

syntax: *job_id*:= *job_number* + (*tag_number* * 65536)

HOME_CSET [*job_id*],directory$
HOME_CURR$ [**(***job_id***)**]
HOME_DEF *job_name*$**,** *file_name*$
HOME_DIR$ [**(***job_id***)**]
HOME_FILE$ [**(***job_id***)**]
HOME_SET *job_id***,***device_directory_and_filename*$
HOME_VER$

example: **HOME_CSET 262148,'Win1_Launchpad_'**
{set Current Directory for job with ID of 262148 ($00040004) to Win1_Launchpad_}
result$=HOME_CURR$
{return the Current Directory for the current job}
HOME_DEF "Sbasic", "dev1_sbasic_test_bas"
{set default filename for Sbasic to dev1_sbasic_test_bas}
result$=HOME_DIR$(-1)
{return the Home Directory for the current job (job's own Home Directory)}
result$=HOME_DIR$(JOBID('launchpad'))
{returns the Home Directory for job called 'Launchpad', using the JOBID function to provide the job ID of 'Launchpad'}
result$=HOME_FILE$
{return the Home Filename for the current job}
HOME_SET -1,'win1_dir_myprog_exe'
{set job's own home directory, home file and current directory }
result$ = HOME_VER$
{get the HOME thing version number into the string result$}
PRINT HOME_VER$
{display the version number of the HOME thing}

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 there 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' , f lp1_qtyp)** {just one copy on the specified drive}
iii. **ERT HOT_RES (' t' ,' f lp1_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, capsclock, u)** {set up unlocked "capsclock" 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 flpl_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_GETSTUFF$ will return the current, or previous content of the Stuffer Buffer.

If no parameter is supplied, or the parameter is 0, Then the current content of the Stuffer Buffer is returned. If the supplied parameter is –1, Then the previous content of the Stuffer Buffer is returned.

syntax: **HOT_GETSTUFF$** [**(** 0 | -1 **)**]

example:

i.	**HOT_STUFF "abc","def"**	{fill Stuffer Buffer}
ii.	**PRINT HOT_GETSTUFF$**	{displays "abcdef"}
iii.	**HOT_STUFF "123"**	{fill Stuffer Buffer again}
iv.	**PRINT HOT_GETSTUFF$ (0)**	{displays "123"}
v.	**PRINT HOT_GETSTUFF$ (-1)**	{displays "abcdef"}

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 the Q68, 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 one or more new lines after the last **HOT_KEY** string, you should put one of 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 very similar 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 there 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
WM_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*.

WM_INK will set the colour of the ink using one of the Windows Manager colour palettes.

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

example:
i. **INK 5**
ii. **INK 6,2**
iii. **INK #2,255**
iv. **WM_INK $0202**

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 immediatly (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
iv.  INPUT#3,x$
```

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 intepreted 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)**

IO_PRIORITY

IO_PRIORITY sets the priority of the I/O retry operations. In effect, this sets a limit on the time spent by the scheduler retrying I/O operations.

A priority of one sets the I/O 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.}

JOBID multitasking

JOBID will return the 32-bit ID of the given job details as a decimal value. The optional parameters may be either a job number and job tag (as displayed by the **JOBS** command), or the job name.

If no parameters are supplied, the Job ID number of the current job is returned.

syntax: *job_identifier*:= | *job_number* , *tag_number*
| *job_number* + (*tag_number* * 65536)
id:= *job_identifier*
name:= | *name*
| *string_expression*

JOBID [(*id* | *name***)]**

example: i. **PRINT JOBID**
ii. **PRINT JOBID(6,5)**
iii. **PRINT JOBID(pick)**

JOBS **multitasking**

JOBS is a command to list to the window attached to the specified or default channel, all the Jobs running in the Q68 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 Q68 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 **multitasking**

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 the Q68. Thus a scan starting with id = 0 and top Job id = 0 will scan all Jobs in the Q68.

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 **multitasking**

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

KILLSOUND **sound**

KILLSOUND will stop any sound generation.

Please note that this also removes the first job called "SOUNDFILE JOB" that it can find. NORMALLY, this should be the only sound playing job that runs in the machine. It is indeed not advisable to have several jobs all trying to make sounds in the machine at the same time, since the sounds will be totally intermingled and garbled.

syntax: **KILLSOUND**

example **KILLSOUND**

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**
39	IT	Italian (in Italy)
34	E	Spanish (in Spain)
33	F	French (in France)
44	GB	English (in England)
45	DK	Danish (in Denmark)
46	S	Swedish (in Sweden)
47	N	Norwegian (in Norway)
49	D	German (in Germany)
1	USA	English(US) (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
39	IT	Italian (in Italy)
34	E	Spanish (in Spain)
33	F	French (in France)
44	GB	English (in England)
45	DK	Danish (in Denmark)
46	S	Swedish (in Sweden)
47	N	Norwegian (in Norway)
49	D	German (in Germany)
1	USA	English(US) (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 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.

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)**

LOAD
QLOAD devices, directory devices

LOAD will load a SBASIC program from any Q68 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**
vi. **OPEN_IN#4,pipe_alpha**
LOAD#4 {load a program from a channel }

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 **devices**

LRESPR opens the file to be loaded and finds the length of the file, then reserves space for the file in the resident procedure area, or the common heap, 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 18 for the Q68.

MAKE_DIR
FMAKE_DIR **directory devices**

The command **MAKE_DIR** is used to create a new subdirectory on a directory device. 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	**-7**	**not found**	Medium or drive is not available
	-8	**already exists**	Already directory/file of that name
	-9	**in use**	Already directory/file of that name
	-15	**bad parameter**	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 it’s 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 the Q68 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 now only seems to have any effect in 512 x 256 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. The wakeup factor ranges from 1 to 9 with 1 being the most sensitive one.

syntax: *acceleration*:= *numeric_expression*
wakeup:= *numeric_expression* {1 to 9}

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 **TURNTO** 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 allows the network station number to be set. If a station number is not explicitly set then the Q68 assumes station number 1.

Although the Q68 does not natively support the QL network. With some additional hardware and software, the Q68 can be connected to a QL network.

syntax: *station*:= *numeric_expression* {range 1 to 127}

NET *station*

example: i. **NET 63**
ii. **NET 1**

comment: Confusion may arise if more than one station on the network has the same station number.

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: **SELect** 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 Q68 *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**.

comment: See also **FOPEN**, **FOP_IN**, **FOP_OVER**, **FOP_DIR**, and **FOP_NEW** for function versions of the above commands.

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}

The following example will create a pop up window that will restore the background when it has finished.

```
100 WMON
110 OUTLN                      {set the screen to be managed}
120 ch=FOPEN('con')            {opens a secondary window}
130 OUTLN#ch,100,100,200,10,4,4 {saves the background under the secondary}
140 CLS#ch
150 PRINT#ch,"Hello"
160 PAUSE#ch,-1
170 OUTLN#ch,0,0,200,10        {restores the background, note no size given}
180 CLOSE#ch
```

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 hardware that does 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
WM_PAPER windows

PAPER sets a new paper colour (i.e. 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*.

WM_PAPER will set the colour of the paper using one of the Windows Manager colour palettes.

syntax: **PAPER** [*channel*,] *colour*
WM_PAPER [*channel*,] *wm_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
```

PARNAM$ procedures

The function **PARNAM$** 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 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***)**

PAUSE

PAUSE will cause a program to wait a specified period of time. Delays are specified in units of 20ms in the UK only, otherwise 16.67ms. If no delay is specified, or the delay is -1, 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}

PE_BGON, PE_BGOFF **extended environment**

PE_BGON allows printing to continue to partially covered windows.

PE_BGOFF blocks printing to partially covered windows.

By default, background printing is turned off. So use the **PE_BGON** command in your boot file if you want to keep it on.

syntax:	**PE_BGOFF**	{turn off background window drawing}
	PE_BGON	{turn on background window drawing}

PEEK, PEEK_W
PEEK_L, PEEK_F SBASIC

PEEK is a function which returns the contents of the specified memory location. **PEEK** has four forms which will access a byte (8 bits), a word (16 bits), a long word (32 bits), or a six byte floating point number.

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_expressionl* \ *numeric_expression*

PEEK(*address***)**	{byte access}
PEEK_W(*address***)**	{word access}
PEEK_L(*address***)**	{long word access}
PEEK_F(*address***)**	{floating point 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.	**PRINT PEEK_L(12200)**	{floating point number contents of location 12200}
v.	**ramt = PEEK_L (! !$20)**	{find the top of RAM $20 bytes on from the base of the system variables}
vi.	**job1 = PEEK_L (!$68!4)**	{find the base address of Job 1 (4 bytes on from base of Job table)}
vii.	**dal = PEEK_W (\\$94)**	{find the current data line number
viii.	**n6 = PEEK_W (\$18\2+6*8)**	{find the pointer for the 6th name in the name table}
ix.	**nl6 = PEEK (\$20\n6)**	{...and the length of the name}
x.	**n6$ = PEEK$ (\$20\n6+1, nl6)**	{...and the name itself}

comment: **PEEK_W** will return negative numbers for values above 32768

warning: For word and long word access the specified address must be an even address.

PEEKS, PEEKS_W
PEEKS_L, PEEKS_F **SBASIC**
Supervisor mode equivalents of **PEEK** for access to I/O hardware in Atari ST & Q40 systems.

PEEK$ **SBASIC**
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$ **SBASIC**
Supervisor mode equivalent of **PEEK$** for access to I/O hardware in Atari ST & Q40 systems.

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 π.

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, POKE_F SBASIC

POKE allows a memory location to be changed. For word and long word accesses the specified address must be an even address.

POKE has four forms which will access a byte (8 bits), a word (16 bits), a long word (32 bits), or a six byte floating point number.

POKE may be referenced form 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 **POKE**d 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 **POKE**d in order. The length is not **POKE**d.

syntax: *address*:= *numeric_expression*
| !! *numeric_expression*
| ! *numeric_expression* ! *numeric_expression*
| \\ *numeric_expression*
| \ *numeric_expressionl* \ *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}
POKE_F *address*, *data* [* ,*data* *]	{floating point 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_F 131072,12345**	{set six bytes at 131072 to the floating point version of 12345}
iv.	**POKE_W ! !$8E,3**	{set the auto-repeat speed to 3}
v.	**POKE !$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, POKES_F SBASIC

Supervisor mode equivalents of **POKE** for access to I/O hardware in Atari ST & Q40 systems.

POKE$ SBASIC

POKE$ will pokes the supplied string of bytes into memory, starting from the supplied address.

syntax: *start_address*:= *numeric_expression*

POKE$ *start_address*, *string*

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$ SBASIC

Supervisor mode equivalent of **POKE$** for access to I/O hardware in Atari ST & Q40 systems.

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 Q68 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 0 for the Q68.

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 Protects the backup time and date in (Super)Gold card systems. Not used in the Q68.

PROT_MEM

Sets the memory protection level in Atari ST & Q40 systems. Not used in the Q68

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$ devices

The **PRT_USE$** function will return as a string the name of the device associated to the PRT device by the **PRT_USE** command.

syntax: **PRT_USE$**

example: **PRINT PRT_USE$**

QUB_DRIVE

QUB_DRIVE allows you to set the QUB drive to the SD card it is to be found on.

syntax:	*drive*:= *numeric_expression*	{range 1 to 8}
	card:= *numeric_expression*	{range 1 to 2}

QUB_DRIVE *drive, card, filename*

example:	i. **QUB_DRIVE 1,1,QL_bdi4.bin**	{sets QUB1_ to be ql_bdi4.bin on card 1}
	ii. **QUB_DRIVE 4,2,games.bin**	{sets QUB4_ to be games.bin on card 2}

note: Only FAT32 partitions on a SD card can be used for QUB drives.

QUB_DRIVE$

The **QUB_DRIVE$** function will return the QUB filename and card number for which a QUB drive is configured.

The string returned by **QUB_DRIVE$** will be formatted as follows:
<container_file>,<card_number>

For example **PRINT QUB_DRIVE$**(1) might return "games.bin,2".

syntax: *drive*:= *numeric_expression* {range 1 to 8}

QUB_DRIVE$ (*drive*)

example:
i. **PRINT QUB_DRIVE$(1)**
ii. **a$=QUB_DRIVE$(4)**

QUB_USE directory devices

QUB_USE allows renaming of the QUB device. **QUB_USE** without a parameter will reset the name of QUB back to QUB.

syntax: **QUB_USE** [*name*]

example:	i. **QUB _USE flp : LOAD flp2_prog**	{loads 'prog' from QUB2_ }
	ii. **QUB _USE**	{and now its name is QUB again}
	iii. **QUB_USE ram : DIR ram1_**	{displays directory of QUB1_}

QUB_WP

QUB_WP sets the write protection on a QUB device.

syntax: *drive*:= *numeric expression*
flag:= *numeric expression* {range 0 or 1}

QUB_WP *drive*,*flag*

example: i. **QUB_WP 1,1** {set write protect for the drive accessed by QUB1}
ii. **QUB_WP 1,0** {clear write protect for the drive accessed by QUB1}

QUIT SBASIC

QUIT will end any SBASIC daughter jobs whether it has been created by the **SBASIC** command, **EX** or any other means.

An optional negative error code may returned to the calling program, when the SBASIC program has been started via **EW**.

syntax: *error_code*:= *negative_numeric_expression*

QUIT [*error_code*]

example: **QUIT – 4** {return a ‘value out of range’ error to calling program}

comment: **QUIT** will not end the primary SBASIC job (job 0).

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}

RCNT_ADDF **recent thing**

RCNT_ADDF add a file name to the recent thing list.

The optional Job ID, which defaults to -1 for the current job. Is the job that is supposed to have opened the file.

The **USE OF THIS KEYWORD IS STRONGLY DISCOURAGED.**
Normally, a program should not call this, the adding of file is handled by the system whenever a file is opened.

syntax: *job_identifier*:= *job_number* + (*tag_number* * 65536)
id:= job_identifier

RCNT_ADDF [*id*,] *filename*

example:
i. **RCNT_ ADDF win1_letter_txt**
ii. **RCNT_ ADDF 4+(3*65536), win2_test_txt**
iii. **RCNT_ADDF 4, ram3_temp**
iv. **RCNT_ADDF JOBID(6,5), win1_fred**

note: The Job ID MUST be passed as a long word.

RCNT_GALL **recent thing**

The **RCNT_GALL** function gets all filenames of the general list into a buffer. The filenames will be copied one after the other, the name of the most recently opened file being the first one to be copied. If the filenames don't all fit, as many as possible will be copied and the error **buffer full** is returned.

Filenames are typical SMSQ/E strings, with a length word in front and evened out to start at an even address.

The length of the supplied buffer should be at least as much as that returned by **RCNT_INFO**

The returned integer result will be 0, or the number of files retrieved if all went OK. Otherwise an error code:

-5	Buffer full	Buffer too small, as much as possible is filled in the buffer
-15	Bad parameter	Wrong number of parameters

Any error from the thing use routine

syntax: *length*:= *numeric_expression*
buffer:= *numeric_expression*

RCNT_GALL (*length*, *buffer*)

example:
i. **PRINT RCNT_GALL (200,buff_base)**
ii. **result% = RCNT_GALL (length,buffer)**
iii.

```
100 str_len%=0
110 str_nbr%=0
120 str_max%=0
130 blength=RCNT_INFO(,0,str_nbr%,str_len%,str_max%)
140 buffer=ALCHP(blength)
150 result%=RCNT_GALL (blength,buffer)
```

note: It may be a better way to get the filenames with the **RCNT_GARR** function.

warning: In V3.38 of SMSQ/E this command may cause SBASIC and system problems.

RCNT_GALJ **recent thing**

The **RCNT_GALJ** function gets all filenames of the list for a job into a buffer. The filenames will be copied one after the other, the name of the most recently opened file being the first one to be copied. If the filenames don't all fit, as many as possible will be copied and the error **buffer full** is returned.

Filenames are typical SMSQ/E strings, with a length word in front and evened out to start at an even address. This might be used as follows:

The length of the supplied buffer should be at least as much as that returned by **RCNT_INFO**

The returned integer result will be 0, or the number of files retrieved if all went OK. Otherwise an error code:

-5	Buffer full	Buffer too small, as much as possible is filled in the buffer
-15	Bad parameter	Wrong number of parameters
-2	Invalid job	Wrong job ID

Any error from the thing use routine

syntax: *job_identifier*:= | *string_expression*
| *job_number* + (*tag_number* * 65536)

id:= job_identifier
length:= *numeric expression*
buffer:= *numeric expression*

RCNT_GALJ ([*id*,] *length*, *buffer*)

example:
i. **PRINT RCNT_GALJ (200,buff_base)**
ii. **result% = RCNT_GALJ ("harry",length,buffer)**
iii.
```
100 str_len%=0
110 str_nbr%=0
120 str_max%=0
130 blength=RCNT_INFO("Prowess",str_nbr%,str_len%,str_max%)
140 buffer=ALCHP(blength)
150 result%=RCNT_GALJ ("Prowess",blength,buffer)
```
iv.
```
100 REMark Get and display a list of filenames for job 0
110 PRINT
120 blength=1000
130 buffer=ALCHP(blength)
140 result%=RCNT_GALJ(0,blength,buffer)
150 PRINT "result ";result%
160 PRINT
170 ptr=0
180 REPeat loop
190  slength=PEEK_W(buffer+ptr)
200  IF slength=0 THEN EXIT loop
210  FOR x=0 TO slength-1
220   PRINT CHR$(PEEK(buffer+ptr+2+x));
230  END FOR x
240  PRINT
250  IF slength MOD 2 THEN slength=slength+1
260  ptr=ptr+2+slength
270 END REPeat loop
280 PRINT
290 RECHP buffer
```

RCNT_GARR **recent thing**

RCNT_GARR will fetch all filenames from the general list, and place them into a 2 dimension string array.

This is similar to the **RCNT_GALL** function, in that all, or as many as possible, filenames will be copied. Here however, they will be copied into what must be a two-dimensional string array (i.e. **DIM a$(xx,yy)**. If the filenames do not all fit, as many as possible will be copied, and no error is returned.

An error of **bad parameter** will however be returned if:

The array isn't a two-dimensional string array
The second dimension of the array is too small for the longest element in the list.

Note that filenames will not be longer than 41 characters, so a **DIM a$(x,41)** will guarantee that that error won't happen.

The first array element to be filled in will be element 0.

syntax: *string_array*:= *2 dimensional string array*

RCNT_GARR *string_array*

example:

```
100 str_len%=0
110 str_nbr%=0
120 str_max%=0
130 blength=RCNT_INFO(0,str_nbr%,str_len%,str_max%)
140 DIM files$(str_nbr%,str_len%)
150 RCNT_GARR files$
```

RCNT_GFFA$ **recent thing**

The **RCNT_GAAFA$** function will return the name of the first (i.e. most recently opened) file from the general list. If the list is empty, this will be a null length string.

syntax: **RCNT_GFFA$**

example: **PRINT RCNT_GFFA$()**

RCNT_GFFJ$ **recent thing**

The **RCNT_GFFJ$** function will return the name of the first (i.e. most recently opened) file for the job passed as parameter. If the parameter omitted, it will default to -1, i.e. the current job. If the list is empty, the result will be a null length string.

syntax: *job_identifier*:= | *string_expression*
| *job_number* + (*tag_number* * 65536)
id:= *job_identifier*

RCNT_GFFJ$ ([*id*])

example:
i. **PRINT RCNT_GFFJ$ ("SBASIC")**
ii. **a$=RCNT_GFFJ$ (4+(3*65536))**
iii. **a$=RCNT_GFFJ$ (JOBID(4,3))**
iv. **PRINT RCNT_GFFJ$ (0)**

RCNT_HASH$ **recent thing**

The **RCNT_HASH$** function will return the hash, as used by the RECENT Thing for job names, from the string passed as parameter.

The hash algorithm used is a very simple one, the consideration was speed over anything else. So this hash is certainly very easy to break and probably not very collision proof....
That said, running it over an entire qxl.win file with about 10,000 files did not give any collision for any of the filenames.

syntax: *string*:= *string_expression*

RCNT_HASH$ (*string***)**

example: **hash$=RCNT_HASH$(string$)**

RCNT_INFO **recent thing**

The **RCNT_INFO** function returns some information about the lists of files maintained by the Thing, either the general list (job 0), or the list for a certain job. The Job ID may be passed as a long word, or, preferably, as a string with the entire name of the job. The job ID will default to the current job (-1).

On return the function returns the size for using the **RCNT_GALL** or **RCNT_GALJ** keywords. The size is the size necessary to store all strings, plus a length word for each string, and a possible byte necessary to even out each individual string length.

The three integer variables will also be set on return as follows:

The *string_number*% parameter contains the number of files in the list. This number may vary, though, if a new file is later opened.

The *string_length*% parameter contains the length of the longest filename currently being stored by the Recent list for this job, in the general list. This length may vary, though, if a new file with a longer name is later opened. It will never exceed 41 characters, though.

The *max_number*% parameter contains the maximum number of files a list may hold (as configured by the user).

syntax: *job_identifier*:= | *string_expression*
| *job_number* + (*tag_number* * 65536)
id:= job_identifier
string_number%:= *integer variable*
string_length%:= *integer variable*
max_number%:= *integer variable*

RCNT_INFO ([*id*,] *string_number*, *string_length*, *max_number*)

example:
i. **length=RCNT_INFO (str_nbr%,str_len%,max_nbr%)**
ii. **length=RCNT_INFO ("harry", str_nbr%,str_len%,max_nbr%)**
iii. **length=RCNT_INFO (4, str_nbr%,str_len%,max_nbr%)**
iv. **length=RCNT_INFO (JOBID(2,2), str_nbr%,str_len%,max_nbr%)**
v.
```
100 REMark Show INFO for job 0
110 str_len%=0
120 str_nbr%=0
130 str_max%=0
140 PRINT
150 blength=RCNT_INFO(0,str_nbr%,str_len%,str_max%)
160 PRINT "info length ";blength
170 PRINT "str_len ";str_len%,"str_nbr ";str_nbr%,"str_max ";str_max%
```

RCNT_JOBS **recent thing**

The **RCNT_JOBS** function returns a listing in the supplied buffer, of all jobs for which the recent thing holds lists of files.

For each job, the listing in the buffer holds the Job ID in a long word followed by a standard string with the job name (which my be 0 if the job has no name).

The list is terminated by a long word of -1. If the buffer is too small to hold all job names, the buffer will be filled in as much as possible, and an error of -5, **Buffer full** will be returned.

There is no guarantee that any of the Job IDs returned are still valid, If the job with that ID has been removed, in that case, then the Job ID will no longer be valid.

The returned value will be negative for an error, and 0, or a positive number, for the number of jobs in the list.

The buffer should preferably be a space allocated with **ALCHP**.

syntax: *length*:= *numeric_expression*
buffer:= *numeric_expression*

RCNT_JOBS (*length*, *buffer*)

example: i. **PRINT RCNT_JOBS (200,buff_base)**
ii. **result% = RCNT_JOBS (length,buffer)**
iii.

```
100 REMark List jobs using the Recent Thing
110 PRINT
120 blength=1000
130 buffer=ALCHP(blength)
140 result%=RCNT_JOBS(blength,buffer)
150 PRINT "result ";result%
160 PRINT
170 ptr=0
180 REPeat loop
190  job=PEEK_L(buffer+ptr)
200  IF job=-1 THEN EXIT loop
210  slength=PEEK_W(buffer+ptr+4)
220  PRINT "Job name - ";
230  FOR x=0 TO slength-1
240   PRINT CHR$(PEEK(buffer+ptr+6+x));
250  END FOR x
260  PRINT
270  IF slength MOD 2 THEN slength=slength+1
280  ptr=ptr+6+slength
290 END REPeat loop
300 PRINT
310 RECHP buffer
```

RCNT_LOAD **recent thing**

RCNT_LOAD will load the lists as saved by **RCNT_SAVE**. All lists existing in the thing, except for the general list, will be removed prior to loading. The file from which the lists are loaded is as configured by the user, you can not specify another file.

Thus, if you LOAD the lists as the very first thing in your boot file, they will also fill up with the files opened up during your normal boot.

If you LOAD the lists at the end of your boot file, they will replace all lists generated up to that time (except for the general list).

When SAVEing or LOADing, the name of the SAVE file is NOT added to any list of files, not even the general one.

It is possible to save the lists, re-configure SMSQ/E to use lists with a different size, and load the lists after a reboot with the newly configured SMSQ/E. In that case:

> If the new list size is smaller than the saved size, only some files will be copied to the new list.
> THERE IS NO GUARANTEE THAT THESE will include the newest files opened.
>
> If the list size is larger than the saved size, all filenames will be copied.

In the latter case, and also when the sizes stay the same between saving and loading, the order of the filenames will be preserved.

syntax: **RCNT_LOAD**

example: **RCNT_LOAD**

RCNT_REMV **recent thing**

RCNT_REMV will remove a list for a job. This removes the list for the job passed as parameter. If no such job exists, it returns an error.

syntax: *job_identifier*:= | *string_expression*
| *job_number* + (*tag_number* * 65536)
id:= *job_identifier*

RCNT_REMV [*id*]

example:
i. **RCNT_REMV**
ii. **RCNT_REMV -1**
iii. **RCNT_REMV 4+(3*65536)**

RCNT_SAVE **recent thing**

RCNT_SAVE will save the lists for all jobs currently held in the thing, into the file configured by the user. The file is overwritten. You can't specify another file. The general list is NOT saved. The name of the SAVE file is NOT added to any list of files, not even the general one, when SAVEing or LOADing.

syntax: **RCNT_SAVE**

example: **RCNT_SAVE**

RCNT_SYNC **recent thing**

Job IDs stored in the Recent Things may not correspond to the Job IDs of the jobs currently executing, for example after loading the lists.

RCNT_SYNC runs through the list of all iobs currently executing in the system and if a list exists for a job with that name, it sets the Job ID of that list to that name.

syntax: **RCNT_SYNC**

example: **RCNT_SYNC**

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 512 x 256 QL colour mode. Using it in other screen modes gives unpredictable effects.

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:	*start_line*:=	*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*];] [*start_number*] [,*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, i.e. 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 identifer 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 of default channel. An optional negative 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*]

example: **REPORT –1** {display a **Not Complete** error message}

comment: The default channel is #0

RESET

RESET will perform a complete system reset of the computer. The same as turning the power off, and then back on again.

Using this command could result in loss of data (e.g. when you **RESET** while sectors are being written to container files, or the SD card), therefore much care should be taken if this command is used without the control of the user.

syntax: **RESET**

RESPR memory management

RESPR is a function which will reserve some of the resident procedure space. (For example to expand the SBASIC procedure list.)

If resident procedure space is not available, then space will be reserved in the common heap.

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: **RETern** [*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 job control

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 Q68 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.

when saving to a non directory device, The device name may be replaced with a channel number.

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 }
v. **OPEN_NEW#4,pipe_alpha_1000**
SAVE#4
{save the entire program to a channel }

SBASIC

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.

You may start a SBASIC with the **EXEP** command, which allows you to provide a string that is sent to #0 of the SBASIC job.

EXEP “SBASIC”;”lrun ‘win1_program_bas’”

Starts a SBASIC job, which then attempts to load and run the program ‘win1_program_bas’

When a SBASIC job is started with **EXEC**, then no channels are initially opened. So if any commands try to use #0, #1, or #2. It will cause #0 to be opened as a small control window.

SBYTES
SBYTES_O devices, directory devices

SBYTES allows areas of the Q68 memory to be saved on a Q68 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 the Q68's clock to be reset.

Note that **SDATE** does not set the battery backed clock in the Q68. To set the battery backed real time clock. First set the date and time with **SDATE**, then execute the clock utility program named 'Q68SETRTC'

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
FSEND_EVENT pointer environment

SEND_EVENT 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.

The **FSEND_EVENT** function is the same as the **SEND_EVENT** command, except that it returns an error code, rather than stopping the program.

syntax:
jobID:= *numeric_expression*
  | *job_number* , *job_tag*
  | *job_name*
event:= *numeric_expression* {in the range 1 to 256}

SEND_EVENT *jobID*, *event*
FSEND_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}
iv. **result = FSEND_EVENT(20,4,8)**	{send event 8 to job 20, tag 4}

comment: **FSEND_EVENT** will return either 0. For no error, or -2. For an invalid job number.

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 Carrier Detect 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 Carrier Detect 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 1,H**	{Hardware (default) handshaking on SER1}

note: Hardware flow control is not available in the Q68.

SER_PAUSE

Not used in the Q68. 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 10th 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 or 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 **job creation**

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 the Q68 to be slowed down.

syntax: **SLUG** *numeric_expression* {0 to 255}

example: **SLUG 15** {add a 15 thousandths of a second delay}

SOUNDFILE **sound**

SOUNDFILE will load and play a sound file through the SMSQ/E Sampled Sound System.

The supplied filename to be played is not loaded into memory all at once. Hence, it seems desirable to load it from a fast device (e.g. a ram disk) to avoid the music being broken up.

The optional *repetitions* parameter means that, when the end of the file is reached, the file will be replayed again as often as indicated by the *repetitions* parameter. So if the parameter is 1, it will be replayed once again, which means that it will be played twice in total (once + 1 repetition). Please use only positive values from 1 to 32766.

syntax: *repetitions*:= *integer_expression* {1 to 32766}
filename:= *string_expression*

SOUNDFILE *filename* [, *repetitions*]

example: i. **SOUNDFILE "win2_sounds_zap_ub"** {plays sound 1 time}
ii. **SOUNDFILE "win2_sounds_zap_ub", 2** {plays sound 3 times}
iii. **SOUNDFILE "win2_sounds_zap_ub", count** {plays sound count + 1 times}

SOUNDFILE2 sound

SOUNDFILE2 will load and play a sound file through the SMSQ/E Sampled Sound System.

SOUNDFILE2 does just about the same as **SOUNDFILE**, but the sound is played through another job created especially for this. This means that the command comes back immediately after the sound playing job has been set up – it allows your program to continue whilst the sound is being played.

The sound playing job is owned by the job issuing the **SOUNDFILE2** command, so if you remove that job, the sound playing job will be removed, too.

The supplied filename to be played is not loaded into memory all at once. Hence, it seems desirable to load it from a fast device (e.g. a ram disk) to avoid the music being broken up.

The optional *repetitions* parameter means that, when the end of the file is reached, the file will be replayed again as often as indicated by the *repetitions* parameter. So if the parameter is 1, it will be replayed once again, which means that it will be played twice in total (once + 1 repetition). Please use only positive values from 1 to 32766.

syntax: *repetitions*:= *integer_expression* {1 to 32766}
filename:= *string_expression*

SOUNDFILE2 *filename* [,*repetitions*]

example:
i. **SOUNDFILE2 "win2_sounds_zap_ub"** {plays sound 1 time}
ii. **SOUNDFILE2 "win2_sounds_zap_ub", 2** {plays sound 3 times}
iii. **SOUNDFILE2 "win2_sounds_zap_ub", count** {plays sound count + 1 times}

note: Please note that the sound may continue to play for a few seconds after the sound job is killed, as there is an internal buffer. Use **KILLSOUND** to stop them

SOUNDFILE3 sound

SOUNDFILE3 will load and play a sound file through the SMSQ/E Sampled Sound System.

SOUNDFILE3 does just about the same as **SOUNDFILE2**, but the job playing the sound is totally independent of the one issuing the command. This means that the command comes back immediately after the sound playing job has been set up – it allows your program to continue whilst the sound is being played.

The sound playing job is owned by the job issuing the **SOUNDFILE3** command, so if you remove that job, the sound playing job will be removed, too.

The supplied filename to be played is not loaded into memory all at once. Hence, it seems desirable to load it from a fast device (e.g. a ram disk) to avoid the music being broken up.

The optional *repetitions* parameter means that, when the end of the file is reached, the file will be replayed again as often as indicated by the *repetitions* parameter. So if the parameter is 1, it will be replayed once again, which means that it will be played twice in total (once + 1 repetition). Please use only positive values from 1 to 32766.

syntax: *repetitions*:= *integer_expression* {1 to 32766}
filename:= *string_expression*

SOUNDFILE3 *filename* [,*repetitions*]

example:
i. **SOUNDFILE3 "win2_sounds_zap_ub"** {plays sound 1 time}
ii. **SOUNDFILE3 "win2_sounds_zap_ub", 2** {plays sound 3 times}
iii. **SOUNDFILE3 "win2_sounds_zap_ub", count** {plays sound count + 1 times}

note: Please note that the sound may continue to play for a few seconds after the sound job is killed, as there is an internal buffer. Use **KILLSOUND** to stop them

SPJOB **job control**

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

SP_GET **system palette**

SP_GET will retrieve the colour definition of system palette colour scheme and store it at the supplied address.

The colour definitions may then be changed as required, and written back to be made available for use with the command **SP_SET**.

The optional ‘number’ parameter, selects which palette to use (default is 0).

‘address’ is the base of an area in memory to store the colour definitions.

‘first’ is the number of the first system palette colour to retrieve (starting from 0).

‘count’ is the number of colour definitions to retrieve.

syntax:

number:=	0 | 1 | 2 | 3
address:=	*numeric_expression*
first:=	*numeric_expression*
count:=	*numeric_expression*

SP_GET [*channel*,] [*number*,] *address*, *first*, *count*

example:

```
10 totcol% = SP_GETCOUNT                {get all the colours of a system palette}
20 address = ALCHP( totcol% * 2 ) + 4
30 first = 0
40 SP_GET #1, 0, address, first, totcol%
```

warning: The space pointed to by 'address' must have enough space for the number of colours to be retrieved. This is not checked by the system.

SP_GETCOUNT **system palette**

The function **SP_GETCOUNT** will return the number of elements contained in the system palette scheme.

Each system palette has the same number of elements.

syntax: **SP_GETCOUNT**

example: **PRINT SP_GETCOUNT**

SP_JOBOWNPAL **job palette**

SP_JOBOWNPAL allows you to use an externally defined system colour palette in a job.

This allows for more than the 4, internally defined, system palette colour schemes to be used.

syntax: *job_identifier*:= | *job_number*, *tag_number*
| *job_number* + (*tag_number* * 65536)
id:= *job_identifier*
palette_pointer:= *numeric_expression*

SP_JOBOWNPAL [*channel*,] *id* **|** *name***,** *palette_pointer*

example: i. **SP_JOBOWNPAL #4, 2, 1, palette%**
ii. **SP_JOBOWNPAL -1, palette%**

SP_JOBPAL **job palette**

SP_JOBPAL will make active one of the 4 internally defined system palette colour schemes in a job.

syntax: *job_identifier*:= | *job_number*, *tag_number*
| *job_number* + (*tag_number* * 65536)
id:= *job_identifier*
palette_number:= 0 **|** 1 **|** 2 **|** 3

SP_JOBPAL [*channel*,] *id* **|** *name***,** *palette_number*

example: i. **SP_JOBPAL #4, 2, 1, 0** {set job '2,1' system palette to scheme 0}
ii. **SP_JOBPAL -1,3** {set this jobs system palette to scheme 3}

SP_RESET **system palette**

SP_RESET will restore the system palette colour scheme (default 0) to it's original values.

syntax: *palette_number*:= *numeric_expression*

SP_RESET [*channel*,] [*palette_number*]

example: **SP_RESET 2**

SP_SET **system palette**

SP_SET will set the colour definitions of a system palette colour scheme.

The optional ‘number’ parameter, selects which system palette colour scheme to use (default 0).

‘address’ is the base of an area in memory where the colour definition entries are stored.

‘first’ is the number of the first system palette colour to set (starting from 0).

‘count’ is the number of colours to set.

syntax: *number*:= 0 | 1 | 2 | 3
address:= *numeric_expression*
first:= *numeric_expression*
count:= *numeric_expression*

SP_SET [*channel*,] [*number*,] *address*, *first*, *count*

example:
```
10 totcol% = SP_GETCOUNT               {get all the colours of a system palette}
20 address = ALCHP( totcol% * 2 ) + 4
30 first = 0
.
.                                  {change colours as required}
.
.
100 SP_SET #1, 0, address, first, totcol%
```

warning: The space pointed to by '*address*' must have enough space for the number of colours to be set. This is not checked by the system.

SQRT **maths function**

The **SQRT** 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.

If a backslash (\) and a name is supplied in place of a channel, then the statistics are sent to the name.

syntax: **STAT** [#*channel*,] *name*
STAT \name1, *name2*

example: i. **STAT #3,win1_** {sends the statistics of win1_ to #3}
ii. **STAT \ram1_file,win1_** {sends the statistics of win1_ to ram1_file}

comment: Both the channel and the name are optional

STOP **SBASIC**
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
WM_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.

WM_STRIP will set the colour of the strip using one of the Windows Manager colour palettes.

syntax: *wm_colour*:= *numeric_expression* {range 0 … 65535}

STRIP [*channel*,] *colour*
WM_STRIP [*channel*,] *wm_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.

SUSJB **multitasking**
SUSJB will suspend a job for a given number of 20mS ticks. If the number of ticks is set to -1, then the wait will be infinite.

The job identifier may be either a job number and job tag (as displayed by the **JOBS** command), or the job name.

syntax: *job_identifier*:= | *job_number* , *tag_number*
| *job_number* + (*tag_number* * 65536)
id:= *job_identifier*
name:= | *name*
| *string_expression*

ticks:= *numeric expression*

SUSJB [*id* | *name*] , *ticks*

example: i. **SUSJB 5,7,50** {suspend job 5,7 for 50 ticks (1 second)}
ii. **SUSJB 'myprog',10** {suspend the job 'myprog' to 10 ticks (1/5 second)

SYSSPRLOAD **sprites**
SYSSPRLOAD allows you to replace any of the default system sprites with new ones.

syntax: *sprite_number*:= *numeric_expression*

SYSSPRLOAD *sprite_number*, *filename*

example: i. **SYSSPRLOAD 0, win1_newarrow_spr** {replace default arrow pointer}
ii. **SYSSPRLOAD 36, win1_newcursor_spr** {replace cursor}

comment: **SYSSPRLOAD 36, win1_newcursor_spr** is equivalent to the commands **CURSPRLOAD win1_newcursor_spr** followed by a **CURSPRON**

TAN **maths functions**
The **TAN** function 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. **PRINT TAN(3)** {print tan 3}
ii. **PRINT TAN(3.141592654/2)** {print tan PI/2}

TH_FIX
Fix the THING system to the old type (SuperBASIC call)

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 though 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 **directory devices**

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 0^0 , 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 90^0 }
	ii. **TURNTO 0**	{turn to heading 0^0 }

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$(1)**	{prints 3.38 (or later SMSQ version number)}
	iv. **PRINT ver$(-1)**	{print the Job ID (0 for initial SBASIC)}
	v. **PRINT ver$(-2)**	{prints the address of the system variables (163840)}

VIEW **directory devices**

VIEW allows a file to be examined in a window on the Q68's display, or sent to a device. 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 **pointer environment**

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

ERR_NC	Not Complete,	**ERR_NJ**	Invalid Job,
ERR_OM	Out of Memory,	**ERR_OR**	Out of Range,
ERR_BO	Buffer Full,	**ERR_NO**	Channel not Open,
ERR_NF	Not Found,	**ERR_EX**	Already Exists,
ERR_IU	In Use,	**ERR_EF**	End of File,
ERR_DF	Drive Full,	**ERR_BN**	Bad Name,
ERR_TE	Transmit Error,	**ERR_FF**	Format Failed,
ERR_BP	Bad Parameter,	**ERR_FE**	Bad or Changed Medium,
ERR_XP	Error in Expression,	**ERR_OV**	Overflow,
ERR_NI	Not Implemented,	**ERR_RO**	Read Only,
ERR_BL	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_CHECK

WIN_CHECK will test a WIN container file on an SD card is in contiguous sectors on the SD card.

If the command does not return an error, then the container file corresponding to the drive is OK.

syntax: *drive*:= *numeric_expression* {range 1 to 8}

WIN_CHECK *drive*

example: **WIN_CHECK 2** {checks WIN2_ is contiguous}

WIN_DRIVE
WIN_DRIVE$

WIN_DRIVE allows you to assign a container file on a SD card to become a WIN directory device.

The **WIN_DRIVE** command has an intended side-effect in that, If the command is applied to a card that isn't present (any more), all channels to all drives that would have corresponded to files on that card are closed, and the drive definition blocks for such drives are removed.
This is a protection against a card being ripped out of the drive, a new one inserted and a write operation to the new card being made. That way, at least, no old information will be written to the new card.

Please note that **WIN_DRIVE** does not check that the file is actually present and readable on the card.

The **WIN_DRIVE$** function returns, as a string, the name of the container file assigned to the drive passed as parameter, as well as the number of the card this file is on (1 or 2), separated from the name by a comma. Please note that this does not tell you whether such a file actually exists on the card.

syntax:	*drive*:= *numeric_expression*	{range 1 to 8}
	card:= *numeric_expression*	{range 1 to 2}
	file_name:= *string_expression_DOS_filename*	{must obey the “8.3 rule”}
	WIN_DRIVE *drive, card, file_name*	
	WIN_DRIVE$ (*drive_number***)**	

example:	i. **WIN_DRIVE 6,2,"QXL.WIN"**	{The file QXL.WIN on SD card 2 is assigned to as WIN6_}
	ii. **PRINT WIN_DRIVE$(6)**	{after the above example, this will return "QXL.WIN,2"}

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.

syntax:	*drive*:= *numeric_expression*	{range 1 to 8}
	switch:= **0 | 1**	
	WIN_FORMAT *drive* [,*switch*]	

example:	**WIN_FORMAT 1**	{Allow WIN1_ to be formatted}
	FORMAT WIN1_files	{Format win1_ with a medium name of files… you have to echo the two characters displayed ...
	WIN_FORMAT 1,0	{protect WIN1_ again against unwanted formatting}

WIN_SAFE

WIN_SAFE will check whether it is safe to remove a SD card as far as the WIN driver is concerned.

If this command returns without error, then, as far as the WIN driver is concerned, the SD card may be safely removed. If not, it will return the error “is in use”. Be patient then and retry a few seconds later.

syntax:	*card*:= *numeric_expression*	{range 1 to 2}
	WIN_SAFE *card*	
example:	**WIN_SAFE 1**	{checks whether card 1 may be safely removed}

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 flp : LOAD flp2_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

WIN_WP sets the write protection on a WIN device.

syntax:	*drive*:= *numeric expression*	
	flag:= *numeric expression*	{range 0 or 1}
	WIN_WP *drive*, *flag*	

example:	i. **WIN_WP 1,1**	{set write protect for the drive accessed by WIN1}
	ii. **WIN_WP 1,0**	{clear write protect for the drive accessed by 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 comer 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}

WM_MOVEALPHA **window manager**

WM_MOVEALPHA will set the amount of transparency a managed window should have when moved around the screen. Values from 1 (nearly transparent) to 255 (totally opaque) are used. A value of 0 is allowed, but this would make the window completely transparent and you could only see the background, so a value of 255 will actually be used.

syntax: **WM_MOVEALPHA** *numeric_expression* {in the range 0 to 255}

example: i. **WM_MOVEALPHA 1** {window move is almost completely transparent}
ii. **WM_MOVEALPHA 128** {window move is half way between transparent and opaque}
iii. **WM_MOVEALPHA 255** {window move is opaque}

WM_MOVEMODE **window manager**

WM_MOVEMODE will change the way that managed windows may be moved around the screen.

There are four ways for a window to be moved-

0 The original method. The pointer changes to the ‘move window’ sprite which is moved around the screen.

1 The Outline method. Click on the move icon with the mouse, keep holding the button down. An outline of the window appears, which you can move around and position to where you want it. Then release the mouse button.

2 The Full Window mode. This is the same as 1 above, but instead of an outline, the entire window is moved.

3 The Full window with transparency is the same as 2 above, but the window to be moved is made transparent. This is done via "alpha blending".
This type of move is only implemented for display modes where alpha blending actually makes sense, i.e. modes 16, and 32. In other display modes, such as the QL screen modes, it will be redirected to move mode 2.

WM_MOVEMODE will effect all programs running on the system except those which do not use the Window Manager.

syntax: *mode*:= **0 | 1 | 2 | 3**

WM_MOVEMODE *mode*

example: **WM_MOVEMODE 1** {set the ‘Outline’ mode}

comment: You cannot use this move mode with anything but the mouse – the keyboard (cursor keys) will not work.

YEAR%, MONTH%
DAY%, WEEKDAY% **date conversions**

These functions complement the **DATE** and **DATE$** functions, by providing extensions to return the year, month, day and weekday numbers corresponding to the current system date. Or an optional date as supplied in the standard QL format of the number of seconds since the 1st January 1961.

WEEKDAY% returns the day number of the week (0…6 Sunday…Saturday).

syntax: *date*:= **|** *numeric_expression* {number of seconds since 1st January 1961}
| *yyyy,m,d,h,m,s*

YEAR%[(*date*)]	{returns 1961 to 2097}
MONTH%[(*date*)]	{returns 1 to 12}
DAY%[(*date*)]	{returns 1 to 31}
WEEKDAY%[(*date*)]	{returns 0 to 6}

example:

i. **PRINT YEAR%**	{returns current year}
ii. **m%=MONTH%(1234567)**	{returns 1}
iii. **today=DAY%(2002,7,23,10,32,15)**	{returns 23}
iv. **PRINT WEEKDAY%**	{returns current day}

comment: **YEAR%** and **YEAR%(DATE)** are functionally identical.