QL

Keywords

© 1984 SINCLAIR RESEARCH LIMITED
By Stephen Berry *(Sinclair Research Limited)*

QL KEYWORDS

The Keyword Reference Guide lists all SuperBASIC keywords in alphabetical order: A brief explanation of the keywords function is given followed by loose definition of the syntax and examples of usage. An explanation of the syntax definition is given in the *Concept Reference Guide* under the entry *syntax*.

Each keyword entry indicates to which, if any, group of operations it relates, i.e. **DRAW** is a *graphics operation* and further information can be obtained from the *graphics* section of the *Concept Reference Guide*.

Sometimes it is necessary to deal with more than one keyword at a time, i.e. **IF, ELSE, THEN, END, IF,** these are all listed under **IF**.

An index is provided which attempts to cover all possible ways you might describe a SuperBASIC keyword. For example the clear screen command, **CLS**, is also listed under *clear screen* and *screen clear*.

Keyword Index

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.

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

ACOS (*angle*)
ACOT (*angle*)
ASIN (*angle*)
ATAN (*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.

syntax: *seconds:= numeric_expression*

ADATE *seconds*

example:

i. `ADATE 3600` {will advance the clock 1 hour}
ii. `ADATE -60` {will move the clock back 1 minute}

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 arc from 15,10 to 40,40 turning through PI/2 radians}

ii.
```
ARC TO 50,50,PI/2
```
{draw an arc from the last point plotted to 50,50 turning through PI/2 radians}

iii.
```
ARC_R 10,10 TO 55,45,0.5
```
{draw an arc, starting 10,10 from the last point plotted to 55,45 from the start of the arc, turning through 0.5 radians}

AT

windows

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

syntax:

line:= numeric_expression
column:= numeric_expression

AT *[channel,] line, column*

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

AUTO

AUTO allows line numbers to be generated automatically when entering programs directly into the computer. **AUTO** will generate the next number in sequence and will then enter the SuperBASIC line editor while the line is typed in. If the line already exists then a copy of the line is presented along with the line number. Pressing **ENTER** at any point in the line will check the syntax of the whole line and will enter it into the program.

AUTO is terminated by pressing CTRL SPACE

Syntax: *first_line:= line_number*
gap:= numeric_expression

AUTO *[first_line] [,gap]*

example:

i. `AUTO` {start at line 100 with intervals of 10}
ii. `AUTO 10,5` {start at line 10 with intervals of 5}
iii. `AUTO ,7` {start at line 100 with intervals of 7}

BAUD

communications

BAUD sets the baud rate for communication via both serial channels. The speed of the channels cannot be set independently.

syntax: *rate:= numeric_expression*

BAUD *rate*

The value of the numeric expression must be one of the supported baud rates on the QL:

75
300
600
1200
2400
4800
9600
19200 (transmit only)

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

Example:

```
 i.   BAUD 9600
ii.   BAUD  print_speed
```

BEEP

sound

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

syntax: *duration:= numeric_expression* {range -32768..32767}
pitch:= numeric_expression {range 0..255}

grad_x:= numeric_expression {range -32768..32767}
grad_y:= numeric_expression {range -8..7}
wrap:= numeric_expression {range 0..15}
fuzzy:= numeric_expression {range 0..15}
random:= numeric_expressian {range 0..15}

BEEP [duration, pitch
[,pitch_2, grad_x, grad_y
[, wrap
[, fuzzy
[, random]]]]]

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

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

pitch_2 specifies an second pitch level between which the sound will 'bounce'

grad_x defines the time interval between pitch steps.

grad_y defines the size of each step, grad_x and grad_y control the rate at which the pitch bounces between levels.

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

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

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

BEEPING

sound

BEEPING is a function which will return zero (false) if the QL 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
```

BLOCK

windows

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

syntax: *width:= numeric_expression*
height:= numeric_expression
x:= numeric_expression
y:= numeric_expression

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

example:

i.
```
BLOCK 10,10,5,5,7
```
{10x10 pixel white block at 5,5}

ii.
```
100 REMark "bar chart"
110 CSIZE 3,1
120 PRINT "bar chart"
130 LET bottom =100 : size = 20 : left =  10
140 FOR bar =1 to 10
150   LET colour = RND(O TO 255)
160   LET height = RND(2 TO 20)
170   BLOCK size, height, Left+bar*size, bottom-height,0
180   BLOCK size-2, height-2, left+bar*size+l, bottom-
height+l,colour
190 END FOR bar
```

{use **LET** colour = **RND**(0 **TO** 7) for televisions}

BORDER

windows

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

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

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

syntax: *width:= numeric_expression*

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

example:

i.
```
BORDER 10,0,7
```
{black and white stipple border}

ii.
```
100 REMark Lurid Borders
110 FOR thickness = 50 to 2 STEP -2
120   BORDER thickness, RND(0 TO 255)
130 END FOR thickness
140 BORDER 50
```

CALL

Qdos

Machine code can be accessed directly from SuperBASIC by using the **CALL** command. **CALL** can accept up to 13 long word parameters which will be placed into the 68008 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 SuperBASIC use the instructions:

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

CHR$

BASIC

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

syntax: **CHR$**(*numeric_expression*)

example:

```
i.  PRINT CHRS(27)      {print ASCII escape character}
ii. PRINT CHR$(65)      {print A}
```

CIRCLE CIRCLE_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_expession
radius:= numeric_expression
eccentricity:= numeric_expression
angle:= numeric_expression {range 0 to 2 PI}

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}

CLEAR

CLEAR will clear out the SuperBASIC variable area for the current program and will release the space for Qdos.

syntax: **CLEAR**

example: `CLEAR`

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

CLOSE

devices

CLOSE will close the specified channel. Any window associated with the channel will be deactivated.

syntax: *channel:= #numeric_expression*
CLOSE channel

example: i. `CLOSE #4`
ii. `CLOSE #input, channel`

CLS

windows

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

syntax: *part:= numeric_expression*

CLS *[channel,] [part]*

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

example:

i. **CLS** {the whole window}
ii. **CLS** 3 {clear the cursor line}

iii. **CLS** #2,2 {clear the bottom of the window on channel 2}

CODE

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

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

syntax: CODE (string_expression)

example:

i. PRINT CODE("A") {prints 65}
ii. PRINT CODE ("SuperBASIC") {prints 83}

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.

syntax: **CONTINUE**
RETRY

example:
```
CONTINUE
RETRY
```

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 Microdrive 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. **mdv1** 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 mdvl_data_file TO con_ {copy to default window}
ii. COPY neti_3 TO mdvl_data {copy data from network station to mdv_data.}

iii. COPY_N mdvl_test_data TO ser1_ {copy mdvl_test_data to serial port 1 removing header information}

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 COT(3.141592654/2)`

CSIZE

window

Sets a new character size for the window attached to the specified or default *channel*. The standard size is 0,0 in *512 mode* and 2,0 in *256 mode*.

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.

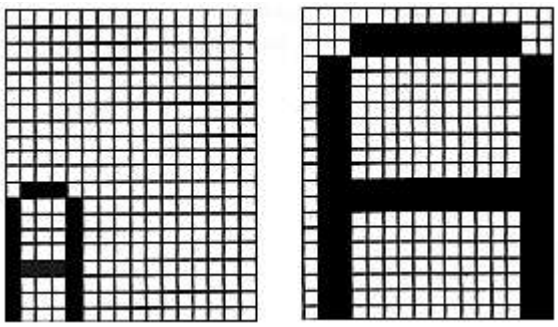

Figure A Character Square

width	size	height	size·
0	6 pixels	0	10 pixels
1	8 pixels	1	20 pixels
2	12 pixels		

syntax: *width:= numeric_expression* {range 0..3}
height:= numeric_expression {range 0..11}

CSIZE *[channel,]- width, height*

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

CURSOR

windows

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

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

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

This allows diagrams to be annotated relatively easily.

syntax: x:= numeric_expression
y:= numeric_expression

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

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

DATA READ RESTORE

BASIC

READ, **DATA** and **RESTORE** allow embedded data, contained in a SuperBASIC 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 SuperBASIC 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$(l2,9)
      110 RESTORE
      120 REMark Data statement example
      130 FOR count=l TO 12 : month$(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.

DATE$ DATE

clock

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

"yyyy mmm dd hh:mm:ss"

where:	*yyyy*	is the year 1984, 1985, 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}

example:
```
i. PRINT DATE$                  {output the date and time}
ii. PRINT DATE$(234567)         {convert 234567 to a date}
```

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)}
```

DEFine FuNction END DEFine

functions and procedures

DEFine FuNction defines a SuperBASIC 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 SuperBASIC *expression*.

Function calls in SuperBASIC 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 x[, identifier]*]*
statements
RETurn *expression*
END DEFine

RETurn can be at any position within the procedure body. **LOCal** statements must preceed the first executable statement in the function.

example:
```
10 DEFine FuNction mean(a, b, c)
20   LOCaL answer
30   LET answer = (a + b + c)/3
40   RETurn answer
```

```
50 END DEFine
60 PRINT mean(1,2,3)
```

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

DEFine PROCedure END DEFine

functions and procedures

DEFine PROCedure defines a SuperBASIC 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 SuperBASIC statement together with a list of actual parameters. Procedure calls in SuperBASIC 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 SuperBASIC; many of the system commands are themselves defined as procedures.

syntax: *formal_parameter:= (expression *[, expression]*)*
*actual_parameters:= expression *[, expression]**

DEFine PROCedure *identifier {forma_parameters}*
*[LOCal identifier *[, identifier]*]*
statements
[**RETurn**]
END DEFine

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

example:

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

ii.
```
100  DEFine  PROCedure  slow_scroll(scroll_limit)
110   LOCal count
120   FOR count = 1 TO scroll
130      SCROLL 2
140   END FOR count
150 END DEFine
160 slow_scroll 20
```

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

DEG

math 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

microdrives

DELETE will remove a file from the directory of the cartridge in the specified Microdrive.

syntax: **DELETE** *device*

The device specification must be a Microdrive device

Example:
i. `DELETE mdv1_old_data`
ii. `DELETE mdv1_letter_file`

DIM

Arrays

Defines an array to SuperBASIC. *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:= indentifier(index *[, index]*)*

DIM *array *[, array] **

example:
i. `DIM string_array$(10,10,50)`
ii. `DIM matrix(100,100)`

DIMN

arrays

DIMN is a function which will return the maximum size of a specified dimension of a specified array. If a dimension is not specified then the first dimension is assumed. If the specified dimension does not

exist or the identifier is not an array then zero is returned.

Syntax: *array:= identifier*
index:= numeric_expression {1 for dimension 1, etc.}

DIMN*(array [, dimension])*

example: consider the array defined by: `DIM a(2,3,4)`
i. `PRINT DIMN(A,1)` {will print 2}
ii. `PRINT DIMN(A,2)` {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

Microdrives

DIR will obtain and display in the *window* attached to the specified or default *channel* Microdrives the directory of the cartridge in the specified Microdrive.

Syntax:

DIR *device*

The device specification must be a valid Microdrive device

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

free_sectors:= the number of free sectors
available_sectors:= the maximum number of sectors on this cartridge
file_name:= a SuperBASIC file name

screen format:
Volume name
free_sectors | *available_sectors* **sectors**
file_name
.....
file__name

example: i. `DIR mdv1_`
ii. `DIR "mdv2_ "`
iii. `DIR "mdv" & microdrive_number$ & "_"`

screen format:

```
BASIC
183 / 221 sectors
demo_1
demo_1_old
demo_2
```

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 SuperBASIC 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}

EDIT

The **EDIT** command enters the SuperBASIC line editor.

The **EDIT** command is closely related to the **AUTO** command, the only difference being in their defaults. **EDIT** defaults to a line increment of zero and thus will edit a single line unless a second parameter is specified to define a line increment.

If the specified line already exists then the line is displayed and editing can be started. If the line does not exist then the line number is displayed and the line can be entered.

The cursor can be manipulated within the edit line using the standard QL keystrokes.

→ cursor right

→ cursor left

↑ cursor up - same as **ENTER** but automatically gives previous existing line to edit next

↓ cursor down - same as **ENTER** but automatically gives next existing line to edit next

CTRL → delete character right

CTRL ← delete character left

When the line is correct pressing **ENTER** will enter the line into the program.

If an *increment* was specified then the next line in the sequence will be edited otherwise edit will terminate.

syntax: *increment:= numeric_expression*

EDIT *line_number [,increment]*

example: i. `EDIT 10` {edit line 10 only}
ii. `EDIT 20,10` {edit lines 20, 30 etc.}

EOF

Devices

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

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

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

EXEC EXEC_W

Qdos

EXEC and **EXEC_W** will load a sequence of programs and execute them in parallel.

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

syntax: *program:* =device {used to specify a Microdrive file containing the program}

EXEC *program*

example: i.
```
EXEC mdv1_communcations
```
ii.
```
EXEC_W mdv1_printer_process
```

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)
```

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 low resolution mode. **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.

FOR END FOR

repetition

The **FOR** statement allows a group of SuperBASIC 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 SuperBASIC 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 SuperBASIC 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 SuperBASIC 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>1E20 THEN
160     PRINT "Very Large number"
170     EXIT value
180   END IF
190 END FOR value
```

Warning:
A floating point variable must be used to control a **FOR** loop.

FORMAT

microdrives

FORMAT will format and make ready for use the cartridge contained in the specified Microdrive.

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

Device specifies the Microdrive to be used for formatting and the identifier part of the specification is used as the medium or volume name for that cartridge. **FORMAT** will write the number of good sectors and the total number of sectors available on the cartridge on the default or on the specified channel.

It is helpful to format a new cartridge several times before use. This conditions the surface of the tape and gives greater capacity.

example:
i. `FORMAT mdv1_data_cartridge`
ii. `FORMAT mdv2_wp_letters`

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

GOSUB

For compatibility with other BASICs, SuperBASIC 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 SuperBASIC make the **GOSUB** statement redundant.

GOTO

For compatibility with other BASICs, SuperBASIC 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`
ii. `GOTO 9999`

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

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 SuperBASIC keyword. This sequence of SuperBASIC 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 SuperBASIC 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 SuperBASIC statements is written following the **IF** statements. The sequence is terminated by the **END IF** statement. The sequence of SuperBASIC statements is executed if the expression contained in the **IF** statement evaluates to be non zero. The **ELSE** keyword and second sequence of SuperBASIC statements are optional.

long 2:
The **THEN** keyword is the last entry on the logical line. A sequence of SuperBASIC 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 SuperBASIC statements is processed. After the **ELSE** keyword a second sequence of SuperBASIC statements is entered, terminated by the **END IF** keyword. If the expression evaluated by the **IF** statement is zero then this second sequence of SuperBASIC 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
```

In all three forms of the **IF** statement the **THEN** is optional. In the short form it must comment 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.

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**. SuperBASIC will match nested **ELSE** statements etc to the closest **IF** statement, for example:

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

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

INK

windows

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

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

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

INKEY$

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

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

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

where:	*time* = 1..32767	{wait for specified number of frames}
	time = -1	{wait forever}
	time = 0	{return immediately}

examples:	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}

INPUT

INPUT allows data to be entered into a SuperBASIC program directly from the QL keyboard by the user. SuperBASIC 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
```

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}
```

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)`

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	I	CAPS LOCK	K	S	F	=	G	;
2	|	Z	.	C	B	£	M	~
1	ENTER	←	↑	ESC	→	\	SPACE	↓
0	F4	F1	5	F2	F3	F5	4	7

LBYTES

devices
microdrives

LBYTES will load a data file into memory at the specified start address.

syntax: *start_address:= numeric_expression*

LBYTES *device, startaddress*

example: i.
```
LBYTES mdvl_screen, 131072
```
{load a screen image}
i. ii.
```
LBYTES mdvl_program, start_address
```
{load a program at a specified address}

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 SuperBASIC assignment statement. The use of the **LET** keyword is optional. The assignment may be used for both string and numeric assignments. SuperBASIC 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

Graphics

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 XO point (2)

parameter_1:= | TO point, angle (1)
| TO point (2)
| point (3)

LINE [channel,] parameter_1 *[, parameter_2]*
LINE_R [channel,] parameter_1 *[,parameter_2]*

where (1) will draw from the specified point to the next specified point
(2) will draw from the the last point plotted to the specified point
(3) will move to the specified point - no line will be drawn

example:
```
i.  LINE 0,0 TO 0,50 TO 50,0 TO 50,0 TO 0,0
ii. LINE TO 0.75, 0.5
iii.LINE 25,25
```
{a square}
{a line}
{move the graphics cursor}

LIST

LIST allows a SuperBASIC line or group of lines to be listed on a specific or default channel.

LIST is terminated by CTRL SPACE

syntax: *line*:= | *line_number* **TO** *line_number* (1)
| *line_number* **TO** (2)
| **TO** *line_number* (3)
| *line_number* (4)
| (5)

LIST *[channel,] line*[,line]**

Where
(1) will list from the specified line to the specified line
(2) will list from the specified line to the end
(3) will list from the start to the specified line
(4) will list the specified line
(5) will list the whole program

Example:
```
i.   LIST                {list all lines}
ii.  LIST 10 TO 300      {list lines 10 to 300}
iii. LIST 12,20,50       {list lines 12,20 and 50 only}
```

If **LIST** output is directed to a channel opened as a printer channel then **LIST** will provide hard copy.

LOAD

devices
Microdrives

LOAD will load a SuperBASIC program from any QL device. **LOAD** automatically performs a **NEW** before loading another program, and so any previously loaded program will be cleared by **LOAD**.

If a line input during a load has incorrect SuperBASIC 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*

example: i. `LOAD "mdv1_test_program"`
ii. `LOAD mdv1_guess`
iii. `LOAD neti_3`
iv. `LOAD ser1_e`

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** (*numenic_expression*) {range greater than zero}
LN (*numeric_expression*) {range greater than zero}

example: i. `PRINT LOG10(20)`
ii. `PRINT LN(3.141592654)`

LOCal

functions and procedures

LOCal allows *identifiers* to be defined to be **LOCal** to a *function* or *procedure*. Local identifiers only exist within the function or procedure in which they are defined, or in procedures and functions called from the function or procedure in which they are defined. They are lost when the function or procedure terminates. Local identifiers are independent of similarly named identifiers outside the defining function or procedure. *Arrays* can be defined to be local by dimensioning them within the **LOCal** statement.

The **LOCal** statement must precede the first executable statement in the function or procedure in which it is used.

syntax: **LOCal** *identifier *[, identifier]**

example: i. `LOCal a,b,c(10,10)`
ii. `LOCal temp_data`

comment:
Defining variables to be **LOCal** allows variable names to be used within functions and procedures without corrupting meaningful variables of the same name outside the function or procedure.

LRUN

devices
Microdrives

LRUN will load and run a SuperBASIC *program* from a specified device. **LRUN** will perform **NEW** before loading another program and so any previously stored SuperBASIC program will be cleared by **LRUN**.

If a line input during a loading has incorrect SuperBASIC 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*

example: i. `LRUN mdv2_TEST`

ii. `LRUN mdv1_game`

MERGE

devices
Microdrives

MERGE will load a file from the specified device and interpret it as a SuperBASIC *program*. If the new file contains a *line number* which doesn't appear in the program already in the QL 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.

If a line input during a **MERGE** has incorrect SuperBASIC 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*

example:
i. `MERGE mdv1_overlay_program`
ii. `MERGE mdv1_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

screen

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.

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`

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

devices
Microdrives

MRUN will interpret a *file* as a SuperBASIC *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**.

If a line input during a merge has incorrect SuperBASIC 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*

example: i. `MRUN mdv1_chain_program`
ii. `MRUN mdv1_new_data`

NET

network

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

syntax: *station:= numeric_expression* {range 1..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 "sti LI 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
```

If **NEXT** is used inside a **REPeat - END REPeat** construct it will force processing to continue at the statement following the matching **REPeat** statement.

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, SuperBASIC 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_NEW

devices
Microdrives

OPEN allows the user to link a logical *channel* to a physical QL *device* for I/O purposes.

If the channel is to a Microdrive then the Microdrive file can be an existing file or a new file. In which case **OPEN_IN** will open an already existing Microdrive file for input and **OPEN_NEW** will create a new Microdrive file for output.

syntax: *channel:= # numeric_expression*

OPEN *channel, device*

example: i.
```
OPEN #5, f_name$
```

ii.
```
OPEN_IN #9,"mdv1_filename"
```
{open file mdv1_file_name}

iii.
```
OPEN_NEW #7,mdv1_datafile
```
{open file mdvl_datafile}

iv.
```
OPEN #6,con_10x20a20x20_32
```
{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,mdv1_read_write_file.
```

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
```

PAN

windows

PAN the entire current window the specified number of pixels to the left or the right. **PAPER** is scrolled in to fill the clear area.

An optional second parameter can be specified which will allow only part of the screen to be panned.

syntax: *distance:= numeric_expression*
part:= numeric_expression

PAN *[channel,] distance [, part]*

where *part* = 0 - whole screen (or no parameter)
part = 3 - whole of the cursor line
part = 4 - right end of cursor line including the cursor position

If the expression evaluates to a positive value then the contents of the screen will be shifted to the right.

example:
i. `PAN #2,50` {pan left 50 pixels}
ii. `PAN -100` {pan right 100 pixels}
iii. `PAN 50,3` {pan the whole of the current cursor line 50 pixels to the right}

warning:

If *stipples* are being used or the screen is in low resolution mode then, to maintain the stipple pattern, the screen must be panned in multiples of two pixels.

PAPER

windows

PAPER sets a new paper colour (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.

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

example:
i. `PAPER #3,7` {White paper on channel 3}

ii. `PAPER 7,2` {White and red stipple}

iii. `PAPER 255` {Black and white stipple}

iv.
```
10 REMark Show colours and stipples
11 FOR colour = 0 TO 7
12   FOR contrast = 0 TO 7
13     FOR stipple = 0 TO 3
14       PAPER colour, contrast, stipple
15        SCROLL 6
16     END FOR stipple
17   END FOR cent rest
18 END FOR colour
```
{not suitable for televisions}

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 then the program will pause indefinitely. Keyboard input will terminate the **PAUSE** and restart program execution.

syntax: *delay:= numeric_expression*

PAUSE *[delay]*

example: i. `PAUSE 50` {wait 1 second}
ii. `PAUSE 500` {wait 10 seconds}

PEEK PEEK_W PEEK_L

BASIC

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

syntax: *address:= numeric_expression*

PEEK(*address)* {byte access}
PEEK_W(*address)* {word access}
PEEK_L(*address)* {long word access}

example: i. `PRINT PEEK(12245)` {byte contents of location 12245}
ii. `PRINT PEEK_W(12)` {word contents of locations 12 and 13}
iii. `PRINT PEEK_L(1000)` {long word contents of location 1000}

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

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 x.

syntax: `PI`

example: `PRINT PI`

POINT POINT_R

graphics

POINT plots a point at the specified position in the *window* attached to the specified or default *channel*. The point is plotted using the *graphics coordinates system* relative to the graphics origin. If **POINT_R** is used then all points are specified relative to the graphics cursor and are plotted relative to each other.

Multiple points can be plotted with a single call to **POINT**.

Syntax: *x:=numeric_expression*
y:=numeric_expression

parameters:= x,y

POINT *[channel,] parameters* [,parameters]**

example: i.
```
POINT 256,128
```
{plot a point at (256,128)}

ii.
```
POINT x,x*x
```
{plot a point at (x,x*x)}

iii.
```
10 REPeat example
20   INK RND(255)
30   POINT RND(100),RND(100)
40 END REPeat example
```

POKE POKE_W POKE_L

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

POKE has three forms which will access a byte (8 bits), a word (16 bits), a long word (32 bits).

syntax: *address:= numeric_expression*
data:= numeric_expression

POKE *address, data* {byte access}
POKE_W *address, data* {word access}
POKE_L *address, data* {long word access}

example: i.
```
POKE 12235,0
```
{set byte at 12235 to 0}
ii.
```
POKE_L 131072,12345
```
{set long word at 131072 to 12345}

Warning:

Poking data into areas of memory used by Qdos can cause the system to crash and data to be lost. Poking into such areas is not recommended.

PRINT

devices
Microdrives

Allows output to be sent to the specified or default *channel*. The normal use of **PRINT** is to send data to the QL 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"`

separators

! 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, SuperBASIC 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.

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}

RANDOMISE

maths functions

RANDOMISE allows the random number generator to be reseeded. If a parameter is specified the parameter is taken to be the new seed. If no parameter is specified then the generator is reseeded from internal information.

syntax: **RANDOMISE** *[numeric_expression]*

example: i. `RANDOMISE` {set seed to internal data}

ii. `RANDOMISE 3.2235` {set seed to 3.2235}

RECOL

windows

RECOL will recolour individual pixels in the window attached to the specified or default channel according to some preset 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.

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, cl, 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 SuperBASIC.

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.

RENUM

RENUM allows a group or a series of groups of SuperBASIC line numbers to be changed. If no parameters are specified then **RENUM** will renumber the entire program. The new listing will begin at line 100 and proceed in steps of 10.

If a start line is specified then line numbers prior to the start line will be unchanged. If an end line is specified then line numbers following the end line will be unchanged.

If a start number and stop are specified then the lines to be renumbered will be numbered from the start number and proceed in steps of the specified size.

If a **GOTO** or **GOSUB** statement contains an expression starting with a number then this number is treated as a line number and is renumbered.

syntax: *startline:=* *numeric_expression* {start renumber}

end_line:=	*numeric_expression*	{stop renumber}
start_number:=	*numeric_expression*	{base line number}
step:=	*numeric_expression*	{step}

RENUM *[start_line [TO end_line];] [startnumber] [,step]*

example: i. `RENUM` {renumber whole program from 100 by 10}
ii. `RENUM 100 TO 200` {renumber from 100 to 200 by 10}

Comment:

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 SuperBASIC 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 SuperBASIC *statements* terminated by an **END REPeat** statement.

The statements between the **REPeat** and the **END REPeat** are repeatedly processed by SuperBASIC.

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.

RESPR

Qdos

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

syntax: *space:= numeric_expression*
RESPR (*space*)

example: `PRINT RESPR(1024)`
{will print the base address of a 1024 byte block}

RETurn

functions and procedures

RETurn is used to force a function or procedure to terminate and resume processing at the statement after the procedure or function call. When used within a function definition them RETurn statement is used to return the function's value.

syntax: **RETurn** [*expression*]

example: i.

```
100 PRINT ack (3,3)
110  DEFine FuNction ack(m,n)
120   IF m=0 THEN RETurn n+1
130   IF n=0 THEN RETurn ack(m-1,1)
140   RETurn ack (m-1,ack(m,n-1))
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**.

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 a SuperBASIC 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

devices
Microdrives

SAVE will save a SuperBASIC program onto any QL device.

syntax: *line:=* | *numeric_expression* **TO** *numeric_expression* (1)
| *numeric_expression* **TO** (2)
| **TO** *numeric_expression* (3)
| *numeric_expression* (4)
| (5)

SAVE *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 mdv1_program,20 TO 70
                    {save lines 20 to 70 on mdv1_program}

ii.  SAVE mdv2_test_program,10,20,40
                    {save lines 10,20,40 on mdv1_test_program}

iii. SAVE net3
                    {save the entire program on the network}
```

iv. `SAVE ser1`
{save the entire program on serial channel }

SBYTES

devices
Microdrives

SBYTES allows areas of the QL memory to be saved on a QL device

syntax: *start_address:= numeric_expression*
length:= numeric_expression

SBYTES *device, start_address, length*

example: i. `SBYTES mdv1_screendata,131072,32768`
{save memory 50000 length 10000 bytes on mdv1_test_program}

ii. `SBYTES mdv1_test_program,50000,10000`
{save memory 50000 length 1000 bytes on mdv1_test_program}

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}

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}

SDATE

clock

The SDATE command allows the QCs clock to be reset.

syntax: *year:= numeric_expression*
month:= numeric_expression
day:= numeric_expression
hours:= numeric_express,on
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 SuperBASIC *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 SuperBASIC 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 SuperBASIC 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.

SEXEC

Qdos

Will save an area of memory in a form which is suitable for loading and executing with the **EXEC** command.

The data saved should constitute a machine code program.

syntax:

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*

example: `SEXEC mdv1_program,262144,3000,500`

comment:

The Qdos 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)`

SQRT

maths function

SQRT will compute the square root of the specified argument. The argument must be greater maths functions 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}

STOP

BASIC

STOP will terminate execution of a program and will return SuperBASIC to the *command interpreter*.

syntax: **STOP**

example: i. `STOP`
ii. `IF n = 100 THEN STOP`

You may **CONTINUE** after **STOP**.

comment:

The last executable line of a program will act as an automatic stop.

STRIP

windows

STRIP will set the current strip colour in the window attached to the specified or default *channel*. The strip colour is the background colour which is used when **OVER 1** is selected. Setting **PAPER** will automatically set the strip colour to the new **PAPER** colour.

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

example: i. `STRIP 7` {set a white strip}

ii. `STRIP 0,4,2` {set a black and green stipple strip}

Comment:

The effect of **STRIP** is rather like using a highlighting pen.

TAN

maths functions

TAN will compute the tangent of the specified argument. The argument must be in the range -30000 to 30000 and must be specified in radians.

syntax: **TAN** (*numeric_expression*) {range -30000..30000}

example: i. `TAN(3)` {print tan 3}
ii. `TAN(3.141592654/2)` {print tan PI/2}

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 degrees, 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 degrees}
ii. `TURNTO 0` {turn to heading 0 degrees}

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}

WIDTH

windows

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}