CONVERTING OTHER BASICS TO QL SUPERBASIC

This document lists some suggestions for how to approach converting some of the keywords and structures in Microsoft-style BASICs which differ from QL SuperBASIC. Sometimes, the differences are major. Sometimes, only a slight change in syntax is required.

This list is not definitive - BASIC even on 1970s and 1980s computers differed greatly from machine to machine. However, this article should help you with the most common things.

In many cases, no simple conversion method is possible, so the document lists how the original command works so that you can devise your own conversion routines appropriate for what the program concerned tries to do.

The term 'QL BASIC' is collectively used to refer to either or both of SuperBASIC on QDOS systems and SBASIC on SMSQ/E systems. SMSQ/E is the successor operating system for QL compatible and derivative hardware systems and emulators. The term 'PC BASIC' is collectively used to refer to Microsoft BASIC (TM) **and compatibles and derivatives on PC systems.**

' comment

The apostrophe ' can be used as shorthand for REM in many cases.

Example:

```
100 'comment
```

can be converted to

```
100 REMark comment
```

The apostrophe comment can also appear after some commands without a colon to separate statements:

Example 1:

```
900 END 'comment
```

Convert it like this (END is equivalent to STOP on the QL):

```
900 STOP : REMark comment
```

Example 2:

```
900 DIM A(10)'setup an array of numbers
```

Convert it like this:

900 DIM A(10) : REMark set up an array of numbers

<> <= and >=

While all of these operators have the same meaning as in SuperBASIC, some older BASICs do not mind if a space is placed between the two characters - this must be removed for SuperBASIC.

I have come across one instance (only) of a BASIC which does not mind which order the symbols are placed in - that BASIC allowed you to use =< and => in place of <= and >=. QL SuperBASIC insists on >= and <=.

+ String Concatenation

Microsoft-style BASICs generally use the '+' operator to join together two or more strings. This must be converted to use a '&' ampersand symbol for SuperBASIC.

Example:

LET A$=B$+C$

is converted to SuperBASIC as

LET A$=B$&C$

?

The query symbol ? is sometimes used as a shorthand for the PRINT command in some versions of BASIC.

200 ?"Hello"

can usually be converted as

200 PRINT"Hello"

^ Exponentiation.

A value raised ot the power of another. For example, PRINT 2^3 gives 8. Same as the equivalent QL operator.

**** Integer Division

Integer division is denoted by the backslash symbol in some versions of BASIC, as opposed to the forward slash symbol to denote ordinary floating point division. Use the INT function in QL BASIC, or the DIV operator in SBASIC, to replace an integer divide statement.

3 \ 2 gives 1, whereas 3/2 gives 1.5.

Abbreviation Of Commands

Some BASICs allow you to abbreviate command names, e.g. P. for PRINT or L. for LIST. These must be entered in full on the QL.

Example:

P."Hello world" convert as PRINT"Hello world"

Labels

Some versions of BASIC permit named labels to be used in place of line numbers in GOTO and GOSUB statements. These labels may look similar to variable names and either enclosed in square brackets or followed by a colon symbol, e.g.

```
GOTO [label1]
GOSUB label2:
```

The labels themselves are placed at the start of a line delimited with the relevant punctuation for the version of BASIC concerned, or after a line number if used. For example:

```
100 GOSUB [label2]
110 ...
1000 [label2]
1010 ...
1020 RETURN
```

or:

```
100 GOTO label1:
110 ...
1000 label1:
1010 ...
```

There is no direct equivalent in QL BASIC - you need to use either line numbers or variables with the values of the line numbers concerned. It may help to place a REM statement at the destination line to help clarify what the original program did, e.g.

```
100 GOTO 1000 : REMark label1
110 ...
1000 REMark label1
1010 ...
```

Scope Of Variables

Some versions of BASIC such as QBASIC allow you to define variables to be local to a given part of a program, or to be

'global'. This varies between versions of BASIC on different systems.

The area of a program that a variable is accessible from is called its "scope".

In general, variables are local to procedures and functions by default (called "local scope"). However, there are times when it is useful to be able to share a variable between a number of procedures or functions. You can make a variable named in a subroutine identical to a variable of the same name in the main program by including a SHARED statement in the subroutine.

This is essentially the opposite to QL BASIC, where variables are generally global in scope unless explicitly declared to be local to a given routine by the use of a LOCal statement.

```
SUB sub1
  SHARED total
  ...
END SUB
```

In QBASIC, a variable may be made global in scope (i.e. accessible from all parts of a program including procedures and functions) by proceding the name in the main program body with a COMMON SHARED statement, e.g. COMMON SHARED total

As well as the scope of a variable there is also the question of when it exists. A variable defined in a subroutine is by default local and only accessible within that subroutine. Not only is access restricted to the subroutine, it is only brought into being when the subroutine is started and vanishes again when the subroutine ends. By using a STATIC statement you can ensure that a variable exists for the life of the entire program even when a subroutine isn't being run. A STATIC variable is still local in the sense that it doesn't have anything to do with a variable of the same name elsewhere in the program but it now also has a life of its own independent of the subroutine.

The keyword STATIC may also be appended to the end of a subroutine definition in QBASIC to make all of its local variables static.

Here is a summary of the scope keywords in QBASIC.

Type	**Accessible From**	**Existence**
local	subroutine that it is declared in	while subroutine is running
shared	main program and any sub-routine with SHARED name	always
global	main program and every subroutine	always
dynamic	one subroutine	while subroutine is

static	one subroutine	running always	

ABS

ABS(X) returns the absolute value of the expression X. Works the same as the QL BASIC function of the same name.

AND

AND is a logical operator which performs tests on multiple relations. Used in a statement such as IF x=1 AND y=2 THEN... its use is exactly the same in QL BASIC. If the tests are positive, it returns a "true" (non-zero) result, otherwsie a "false" value of 0.

Some BASICs allow its use as a bitwise operator where the outcome depends on a comparison of individual bits in two values. AND comparisons return a set bit value where the corresponding bit values are set in both values compared. For example, x = 255 AND 1 returns a value of 1, because that is the only bit set in both values when compared as binary values. Use a double ampersand in QL BASIC for this operator:

```
LET x = y AND z
```

would become `LET x = y && z` in QL BASIC.

Arrays

Arrays are generally dimensioned with the DIM command as in SuperBASIC. Mostly, arrays are dimensioned to have the zero subscript as the first entry in an array, but beware of versions of BASIC where the first element in an array is 1. Not all BASICs zero an array when redimensioning a second or subsequent time in the same run. Some BASICs will clear out arrays when programs stop or start, others may need explicit CLEAR statements to remove values from past runs - do not assume an array will always start zeroed and cleared if it has been run once already.

String arrays may require an extra dimension in SuperBASIC, as the maximum length of string array elements may not be specified in some BASICs. For example, DIM A$(10) would set up an array of 10 or 11 (depending on whether element 0 is allowed) strings, but without specifying the maximum length, so you would have to try to work out what the maximum length of text this array might be expected to hold and add it as an extra dimensions in SuperBASIC, e.g. DIM A$(10,50) if the strings needed to hold up to 50 characters of text.

Beware of versions of BASIC which have case sensitive variable and array names. In this case, DIM A(10),a(10) might create two

separate and distinct arrays called A() and a() - in other words, A(1) and a(1) are not the same things! There is no easy way around this in SuperBASIC, where names are case insensitive, other than to use different variable and array names, or doing something like keeping the upper case names as they are and adding a '_' to the lower case equivalents, e.g. DIM A(10),_a(10)

ASC

Returns a numerical value that is the ASCII code for the first character of the string given. Corresponds to the function CODE in QL BASIC. For example: 10 PRINT ASC("A") corresponds to 10 PRINT CODE("A"). Within the range of character codes up to 127, most characters have the same code value on the QL, although a few symbols vary - see ASCII Codes below.

ASCII Codes

Not all computershave the same character sets. The same symbol may have a different character code on different computers. That said, letters and numbers and most symbols are the same, but you should be wary of some characters:

```
#       QL code 35
[ \ ] ^ _       QL codes 91 to 95 inclusive
£       QL code 96, often a 'back-tick' character found on the
```

key above TAB on a U.K. keyboard. To get a Pound symbol when printing, some printers may require you to send character code 163 for a Pound symbol, e.g. PRINT CHR$(163)

Arrow symbols may be another source of confusion. On TRS-80 Level 2 BASIC they are at the following code locations:

	TRS-80	QL
Left	93	188
Right	94	189
Up	91	190
Down	92	191

Some computers may not have a lower case character set as such, both cases look the same - printing the CHR$ codes for A and a may both result in a capital A. But if sent to a printer they may actually print as separate upper and lower case characters even though they look the same on screen.

If you get stuck with character sets, especially in programs where symbols are printed with something like PRINT CHR$(n) you will need to compare the character sets of the original machine and the QL to see which codes correspond. If the manual has no information, or you don't have access to the manualof the original computer, try looking up character sets online, such as Code Page 437 or CP437, PC-8 or MS-DOS Latin US, espcially if the program uses an American character set.

ASCII codes below 32 and above 127 may have differences between computers. Those below 32 are often called 'control codes' and may have certain screen functions, e.g. CHR$(9) may be a tab function, a Carriage Return for CHR$(13), a Linefeed for CHR$(10), or a Form Feed for CHR$(12), a CHR$(7) may sound a beep... Additionally, codes below 32 may be used as various symbols, sometimes called 'dingbats'. All these control codes may vary - it can be hard to work out what does what!

Similarly, the high ASCII codes above 128 may vary from computer to computer. You will need to look up the relevant information for the computer and version of BASIC in question.

AT

This may be used on its own, like the QL AT command or used in a PRINT command as a PRINT position modifier. Used by itself, AT X,Y can simply be converted to AT Y,X on a QL. Note that the QL takes y coordinate first, followed by x coordinate. Most BASICs have the Y coordinate first, followed by the X, but a few have the X coordinate first.

In cases where there is no ambiguity caused, no punctuation is usually required after the parameter(s) of AT, for example PRINT AT Y,X"Hello" may be perfectly acceptable even if less readable.

If used in a PRINT modifier context, AT may take one of three forms:

```
PRINT AT P;A$    position from start of screen - usually
(y*screen_width)+x. The AT statement must be made a separate
command and P resolved down to Y and X coordinates, such as AT P
DIV screen_width, P MOD screen_width : PRINT A$
PRINT AT Y,X;A$  Change to AT Y,X : PRINT A$
PRINT AT X,Y;A$  Change to AT Y,X : PRINT A$
PRINT AT Y,X"OK" Change to AT Y,X : PRINT"OK"
```

The stand-alone version of AT may appear as a command called LOCATE in some dialects of BASIC.

The version of AT used in a PRINT statement to position the cursor may take additional forms in the various versions of BASIC. Some dialects use the '@' symbol in place of the AT keyword, e.g. PRINT @Y,X; or both commands may be combined into a single parameter which represents the number of text characters from the top left of the screen, e.g. PRINT @position; where "position" is (Y*line_width_in_characters)+X.

Some dialects of BASIC replace the PRINT AT Y,X; version with a TAB command with two parameters: PRINT TAB(Y,X); or PRINT TAB(X,Y); - you will have to check which way around the particular version of BASIC expects the parameters.

ATN

ATN(X) retuns the arctangent of X in radians. The result will be in the range -PI/2 to PI/2. The ATN function corresponds to the ATAN function on the QL.

AUTO

Used for automatic line numbering and entry of lines of BASIC. Generally works in the same way as AUTO on the QL. Allows both initial line number and step vaue to be specified. Unlike the QL version, on some machines it displays a '*' if the line already exists, so that you are aware you may be about to overwrite an existing line and so give you a chance to break out of the command.

BEEP

BEEP just issues a short medium pitched bleep from the computer's speaker. Just use any fairly short BEEP in QL BASIC, e.g. BEEP 2000,20

BLOAD

BLOAD filespec,offset loads binary data (such as a machine language program) into memory, rather like LBYTES in QL BASIC. A knoeledge of system data structures, memory layout and machine language is needed to convert such code.

BSAVE

BSAVE filespec,offset,length saves binary data to disk. Its use is similar to that of SBYTES in QL BASIC.

CALL

CALL calls a machine language subroutine. The syntax is CALL numvar [(variable [,variable]...)]

numvar is the name of a numeric variable. The value of the variable indicates the starting memory address of the subroutine being called as an offset into the current segment of memory (as defined by the last DEFSEG statement)

variable is the name of a variable which is to be passed as an argument to the machine language subroutine.

Example:

```
100 DEF SEG=&H8000
110 OZ=0
120 CALL OZ(A,B$,C)
```

The CALL statement is one way of interfacing machine language programs with BASIC. The other way is by using the USR function.

Machine language code varies greatly from processor to processor, and even from machine to machine with similar processors.

The CALL statement on the QL differs slightly in terms of parameters. Its syntax is CALL address [,registers] where 'registers' means the values to be placed into registers D1-D7 and A0-A5 in that order.

CDBL

CDBL(X) converts X to a double precision number. This has nodirect equivalent in QL BASIC, since SuperBASIC and SBASIC do not support two levels of numeric precision. In most programs, this can simply be omitted. Be aware of the possibility of differently formatted numbers when the QL's exponential number formatting kicks in.

CHAIN

This command may be similar to a QL BASIC LRUN or MRUN command.

CHAIN "filename" generally works like an LRUN "filename" command in QL BASIC, while CHAIN MERGE "filename" generally works like an MRUN "filename" command in QL BASIC. Both versions allow an optional line number to be specified, at which execution continues after the merging. This is not generally possible in QL BASIC. If no start line number is specified, execution begins at the first line.

CHR$

CHR$(X) Returns a string whose one character is ASCII character X. Equivalent to the same function in QL BASIC, but see ASCII Codes above.

CINT

CINT(X) converts X to an integer by rounding the fractional portion. X must be in the range -32768 to 32767. CINT will return 2 for CINT(1.6), or -2 for CINT(-1.6), for example. In QL BASIC, use an equivalent function such as:

```
1000 DEFine FuNction CINT(X)
1010   IF X>=0 THEN RETurn INT(X+0.5): ELSE RETurn INT(X-0.5)
1020 END DEFine CINT
```

CIRCLE

This keyword draws a circle or sllipse on the screen with centre (x,y) and radius r. The full syntax is:

CIRCLE (x,y),r [,colour [,start,end [,aspect]]]

(x,y) are the coordinates of the centre of the ellipse. The coordinates may be given in either absolute or relative form.

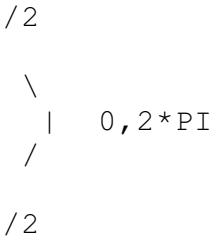
r is the radius (major axis) of the ellipse in points.

colour is a number which specifies the colour of the ellipse. The default is the current foreground (ink) colour.

start,end are angles in radians and may range from -2*PI to 2*PI, where PI=3.141593

aspect is a numeric expression.

start and end specify where the drawing of the ellipse will begin and end. The angles are positioned in the standard mathematical way, with 0 to the right and going counterclockwise:



If the start or end angle is negative (-0 is not allowed), the ellipse will be connected to the centre point with a line, and the angles will be treated as if they were positive (note that this is not the same thing as adding 2*PI). The start angle may be greater or less than the end angle. For example, this short listing will draw a three quarter circle with the top left cut out and joined to the centre of the circle by two straight lines from 12 o'clock and 9 o'clock.

```
10 PI=3.141593
20 SCREEN 1
30 CIRCLE (160,100),60,,-PI,-PI/2
```

aspect affects the ratio of the x-radius to the y-radius. This may be used to adjust the circularity of the shape to suit the screen aspect ratio of the graphics mode in use at the time.

If aspect is less than one, the r is the x radius. That is, the radius is measured in points in the horizontal direction. If aspect is greater than one, then r is the y-radius.

In many cases an aspect of 1 (one) will give nicer looking circles. This will also cause the circle to be drawn somewhat faster.

The last point referenced after a circle is drawn is the cente

of the circle. Points that are off screen are not drawn by CIRCLE.

As graphics commands like this one tend to vary from computer system to system, it is not possible to give a singular method of conversion. QL BASIC supports drawing of circles and ellipses using the CIRCLE and synonymous ELLIPSE commands.

CIRCLE x,y,major_radius, ratio, rotation

Rotation is not used in the Microsoft BASIC version, so you will generally require to draw the ellipse horizontally (rotation=PI/2) or vertically (rotation=0).

x, y and radius will be broadly the same, although dependent on units and screen resolution. The QL BASIC version does not support the 'start, end' options for drawing part of a circle, so you may need to use the ARC command in QL BASIC to achieve this. Aspect ratio is defined as the ratio of minor to major axis in QL BASIC, and x axis to y axis in Microsoft BASIC.

CLEAR

As in SuperBASIC, CLEAR by itself clears the values of all variables. Numeric variables are all reset to a value of 0, strings to nulls or 'empty'.

CLEAR n

When used with an argument n (n can be a constant or an expression), this statement causes the Computer to set aside n bytes for string storage. In addition, all variables are set to zero. When the TRS-80 is turned on, 50 bytes are automatically set aside for strings.

The amount of string storage CLEARed must equal or exceed the greatest number of characters stored in string variables during execution; otherwise an Out of String Space error will occur.

Example:

100 CLEAR 2000

makes 2000 bytes available for string storage. By setting string storage to the exact amount needed, the program can make more efficient use of memory on older systems with smaller memories. A program which uses no string variables could include a CLEAR 0 statement, for example. The CLEAR argument must be non-negative, or an error will result.

CLOAD and **CLOAD?**

Load a program from cassette tape, broadly equivalent to a LOAD command on a QL. Some systems will let you specify a blank

filename, which allows the computer to load the first BASIC programit comes across on the tape. On some systems, if a part filename or a single letter is specified the computer will only load a program whose filename starts with the given letter or letters. There is no direct equivalent to either version on the QL.

Some computers have a variant which uses a question mark to ask the computer to verify that a file on cassette matches that in the computer's memory (in other words, that it is not faulty). This usually takes the form CLOAD ?"filename". If no filename is given, the first program found on the cassette will be compared.

CLOSE

Closes a file channel which is already open. Works much like the QL version of the command, except that multiple channels may be closed by a single CLOSE command with channel numbers (called file numbers in the MS BASIC manuals) separated by commas.

CLOSE #channel is similar to the same command in QL BASIC
CLOSE #chan1, #chan2 is equivalent to CLOSE #chan1 : CLOSE #chan2 on QL.

A third version of the command with no parameters is able to close all open file numbers. There is no direct equivalent in SuperBASIC, unless you have Toolkit 2 which extends the CLOSE command to close all open channel numbers if no parameter is given to the CLOSE command.

CLS

Where implemented (some versions of BASIC just clear the screen automatically when the program starts to run), the implementation is similar to the QL CLS command. This may just clear the whole screen, not a given window number as in QL BASIC, if no parameter is given, otherwise CLS 1 clears the graphics viewport, while CLS 2 clears the text window.

COLOR - Colour Values

This may vary from system to system, of course. Older style 16 colour systems often use the following colour numbers, with the closest QL colour number (8 colours - the QL does not have a 16 colour mode).

Number	**PC Colour**		**QL Number**	**QL Colour**
0	Black	0	Black	
1	Blue	1	Blue	
2	Green	4	Green	
3	Cyan	5	Cyan	
4	Red	2	Red	
5	Magenta	3	Magenta	
6	Brown	6	Yellow	

7	White	7	White
8	Grey	0	Black
9	Light Blue	1	Blue
10	Light Green	4	Green
11	Light Cyan	5	Cyan
12	Light Red	2	Red
13	Light Magenta	3	Magenta
14	Yellow	6	Yellow
15	Bright White	7	White

Once you have established the relationship between PC and QL colours, you may find it easier to set up a colour translation table in an array and indirectly reference the colours rather than trying to amend all colour commands in a program. So, set up an array with the list of colour numbers in the third column, then instead of using INK 0 to convert the statement COLOR 0, for example, you could use INK colour(0). The keyword COLOR (note American spelling) is often used to set the colour of both ink and paper, so you have to use INK or PAPER as appropriate.

COM

The COM command enables or disables trapping of communications activity to the specified communications adapter. The format of the command is COM(n) ACTION, where:

n number of the communications adapter (1 or 2)

and ACTION is a one word parameter describing the action to be taken:

ON A COM(n) ON statement must be executed to allow trapping by the ON COM(n) statement. After COM(n) ON, if a non-zero line number is specified int he ON COM(n) statement, BASIC checks to see if any characters have come in to the communications adapter every time a new statement is executed.

OFF If COM(n) is OFF, no trapping takes place and any communication activity is not remembered even if it does take place.

STOP If a COM(n) STOP statement has been excuted, no trapping can take place. However, ay communications activity that does take place is remembered so that an immediate trap occurs when COM(n) ON is executed.

There is no equivalent command in QL BASIC.

COMMON

This command passes variables to a chained program. The command takes a series of parameters which are either simple variable names, or if brackets are appended to the variable name, it is regarded as the name of an array.

```
100 COMMON A,B(),C$
110 CHAIN "A:PROG2"
```

This example chains to a program called PROG2 on the A: drive of the PC, and passes to it the variable A, the array B() and the string variable C$.

There is no equivalent command in QL BASIC, although the use of MRUN in place of LRUN (i.e. merging a second program instead) may achieve the same purpose as long as you are careful about line numbering.

CONST

The CONST command in QBASIC fixes a value so that it cannot be changed within the program. The full syntax is:

CONST name [AS type]=expression or value

'name' is a name given to the value (like a variable name)

'type' indicates the type of constantand may be INTEGER or STRING.

'value' may be an expression, variable, or a number.

Examples:

```
CONST PI=3.1415926                assigns the value shown to the
                          variable PI.

CONST WHITE AS INTEGER=15         assigns the value 15 to the
                          colour name 'white'

INPUT"What is your name? ";N$     Fix the constant NAME$ as the
CONST NAME$=N$            name first entered.
```

There is no direct equivalent in QL BASIC, although simple LET statements could be used as long as you bear in mind that the assignments are not permanently fixed. You could possibly use environment variables to create such constants if those are implemented on your system.

CONT

Similar to CONTINUE on a QL, to resume program operation following a break or error.

Control Codes

Control codes are mostly keypresses corresponding to character codes 0 to 31. Sometimes they produce symbols known as 'dingbats', but were originally (as the name implies) used to

control printers and terminals, possibly via modems. They may also act as control codes to control screen output in some ways on some machines.

This is a list of control codes, their 2 or 3 letter symbol and a brief description. Many of these codes may vary in their use on different computers while some, such as 8(BS), 9(HT), 10(LF), 12 (FF), 13(CR) and 27 (ESC) generally work similarly on most systems. Many of these codes have no direct QL equivalent, you may have to be rather inventive!

 0 NUL Null char
 1 SOH Start of Heading
 2 STX Start of Text
 3 ETX End of Text
 4 EOT End of Transmission
 5 ENQ Enquiry
 6 ACK Acknowledgment
 7 BEL Bell
 8 BS Back Space - move back and erase last character. Generally, this just means move one space back to the left.
 9 HT Horizontal Tab - move to the right to the next tab stop (usually every 8 or 10 characters, although it can vary)
10 LF Line Feed - move the cursor down to the next line. On older terminal based system, this was equivalent to moving the paper up one line, without moving the cursor left or right horizontally at all. One some systems, a LF is sometimes interpreted as LF (move down one line) then CR (move to left hand column of screen or printer), as it would at the end of a line of text. By and large, the QL uses just an LF to signify the end of a line.
11 VT Vertical Tab
12 FF Form Feed - move to the start of the next sheet of paper, or clear the screen like a CLS command.
13 CR Carriage Return - move the print head or text cursor to the start of the line. Some systems treat this as being equivalent to a CR+LF.
14 SO Shift Out / X-On. On some systems, when a CHR$(14) is sent to the screen, it turns on the cursor, like a CURSEN command on a QL with Toolkit 2.
15 SI Shift In / X-Off. On some systems, when a CHR$(15) is sent to the screen, it turns off the cursor, like a CURDIS command on a QL with Toolkit 2.
16 DLE Data Line Escape
17 DC1 Device Control 1 (oft. XON)
18 DC2 Device Control 2
19 DC3 Device Control 3 (oft. XOFF)
20 DC4 Device Control 4
21 NAK Negative Acknowledgement
22 SYN Synchronous Idle
23 ETB End of Transmit Block. On a TRS-80 this code switches the screen to 32 characters wide mode.
24 CAN Cancel. On some systems this backspaces the cursor (move one to the left) without erasing the current character.

25 EM End of Medium. Move the cursor one position to the right.
26 SUB Substitute. Also, move the cursor down the screen one line on some systems such as TRS-80.
27 ESC Escape - used for expanded code sequences, where ESC signifies this is the start and at least one more code must follow. On a TRS-80, sending ESC to the screen acts as an upward linefeed (move the cursor up one line).
28 FS File Separator. Sent to the screen on some computers this acts as a 'Home' command to force the cursor to move to the top left origin, to text coordinates 0,0, like an AT 0,0 statement on the QL. May also be used to switch a TRS-80 into 64 characters per line screen mode.
29 GS Group Separator. On some computers, sending a CHR$ 29 to the screen moves the cursor back to the beginning of the current horizontal line.
30 RS Record Separator. On some computers, sending a CHR$ 30 to the screen erases text to the end of the line, rather like CLS 4 on a QL.
31 US Unit Separator. On some systems sending a CHR$ 31 to the screen is interpreted as a command to clear to the end of the form, in other words, clear the screen from the cursor position to the bottom, although it varies as to whether this includes the current line or not. Replicate as either CLS 3 : CLS 2 (to clear both the current line and below) or just CLS 2 (to clear below the current line).

COS

COS(X) returns the cosine of X in radians, similar to COS(X) in QL BASIC.

CSAVE

Saves a BASIC program with the given filename to cassette tape, similar to a SAVE command on a QL. On some computers, the filename may consist of a single character only. The filename should be in quotes, or expressed as a string variable.

CSNG

CSNG(X) converts the value of X to a single precision number. Since the QL BASIC does not support different precisions of floating point values, no conversion is required.

CSRLIN

This function returns the vertical coordinate of the cursor. On most systems, the top line of the screen is line number 1, although some may start at 0.

There is no direct equivalent in QL BASIC, although the information required to convert this function is contained within the channel definition blocks for the window channel concerned and may be read using the DIY Toolkit function CHAN_W%

for a given channel number. The vertical cursor position in pixels may be read from offset 36 decimal (hex 24). This starts from 0 and is in pixel units, so to get a text character position across, divide by the number of pixels per character in the current character size, e.g. 6 for CSIZE 0,0 text. The character spacing for a given window channel may be checked with CHAN_W%(#channel,40), so to read the current text position as characters across the screen, use something like:

```
LET pos=CHAN_W%(#channel,36) DIV CHAN_W%(#channel,40)
```

On computers where the POS function starts from 1 for the leftmost position, you should add 1 to this result:

```
LET pos=1+(CHAN_W%(#channel,36) DIV CHAN_W%(#channel,40))
```

or write it as a function in QL BASIC:

```
1000 DEFine FuNction CSRLIN(chan)
1010   RETurn 1+(CHAN_W%(#chan,36) DIV CHAN_W%(#chan,40))
1020 END DEFine CSRLIN
```

CVI CVS CVD

These three functions convert string representation values to numeric values, e.g. values that are read in from a random disk file. CVI converts a two byte string to an integer. CVS converts a 4-byte string to a single precision number. CVD converts an 8-byte string to a double precision number.

The exact conversion method depends on how the program created the file, and is likely to be specific to the BASIC concerned. It is better to rewrite the routines which create the file to create QL-specific data files.

DATA

Similar to the QL BASIC DATA command, so usually no change required as long as string data is quoted. If the version of BASIC concerned supports unquoted strings in DATA statements, something like DATA name1, name2, name3 may need to be converted as DATA "name1","name2", "name3". Unquoted text data is allowed in some versions of BASIC unless the string contains commas, colons, or significant leading or trailing spaces.

DATE$

DATE$ returns a ten character string, which is the system date, in the form mm-dd-yyyy. In QL BASIC, the information is contained in the first 11 characters of the string returned by DATE$ in QL BASIC, although the month is returned as a three character month name dependent on the system language configured., e.g. QL BASIC DATE$ returns 2016 Jan 03 whereas the same date in Microsoft-style BASICs might be 01-03-2016

To translate this to an approximate equivalent in QL BASIC, you could write a function which looks up the month names and converts them to a numeric value. Here, the function name is changed to DATE_$ t avoid a clash with the QL BASIC function name DATE$.

```
7000 DEFine FuNction DATE_$
7010   m$ = DATE$ : m$ = m$(1 TO 11)
7020   y$ = m$(1 TO 4)
7030   d$ = m$(10 TO 11)
7040   m$ = m$(6 TO 8)
7050   t = m$ INSTR 'JanFebMarAprMayJunJulAugSepOctNovDec'
7060   m$ = (t+2) DIV 3 : IF LEN(m$) < 2 THEN m$='0'&m$
7070   RETurn m$&'-'&d$&'-'&y$
7080 END DEFine DATE_$
```

DEF

The DEF keyword may appear in several contexts.

1. DEF FN
Define a function - similar to the QL BASIC's DEFine FuNction.

2. DEF USR
Specifies the start address of an assembly language subroutine. The syntax of this command is:
DEF USR [<digit>]=<integer expression>
where digit may be a number from 0 to 9 (0 assumed if not specified) and <integer expression> is the starting address of the USR routine. Broadly equivalent to a CALL command in QL BASIC, but since any assembler code is likely to be processor specific, you are unlikely to be able to easily convert a DEF USR statement.

DEFDBL letter range

Variables beginning with any letter in the specified range will be stored and treated as double-precision floating point, unless a type declaration character is added to the variable name. A variable defined as double-precision allows 17 digits of precision, with 16 being printed in most cases.

Examples:

200 DEFDBL A Variable names starting with A are double precision, unless ending in ! % or $. So A becomes a double-precision variable, but A$ remains a string variable.

300 DEFDBL A,C Variable names starting with A or C are double-precision, unless ending in ! % or $.

400 DEFDBL C-E Variable names startin with C, D, or E are double-precision, unless ending in ! % or $

DEFDBL and the other DEFxxx commands are generally placed near the beginning of programs for clarity.

DEFINT letter range

Variables beginning with any letter in the specified range will be stored and treated as integers, unless a type declaration character is added to the variable name. A variable defined as integer can only take on values between -32768 and +32767 inclusive.

Examples:

200 DEFINT A Variable names starting with A are integers, unless ending in ! # or $. So A becomes an integer variable, but A$ remains a string variable.

300 DEFINT A,C Variable names starting with A or C are integers, unless ending in ! # or $.

400 DEFINT C-E Variable names startin with C, D, or E are integers, unless ending in ! # or $

DEFINT and the other DEFxxx commands are generally placed near the beginning of programs for clarity.

DEFSNG letter range

Variables beginning with any letter in the specified range will be stored and treated as single-precision floating point variables unless a type declaration character is added to the variable name. A variable defined as single-precision are usually stored with 7 digits precision, although printed to 6 digits of precision.

Examples:

200 DEFSNG A Variable names starting with A are single-precision, unless ending in # % or $. So A becomes a single-precision variable, but A$ remains a string variable.

300 DEFSNG A,C Variable names starting with A or C are single-precision, unless ending in # % or $.

400 DEFSNG C-E Variable names startin with C, D, or E are single-precision, unless ending in # % or $

DEFSNG and the other DEFxxx commands are generally placed near the beginning of programs for clarity.

DEFSTR letter range

Variables beginning with any letter in the specified range will

be stored and treated as strings, unless a type declaration character is added to the variable name. A variable defined as string can usually store up to 255 characters, although some BASICs allocate only a fixed small string space which may have to expanded by a command such as CLEAR n

Examples:

200 DEFSTR A Variable names starting with A are strings, unless ending in ! # or %. So A becomes a string variable, but A% remains an integer variable.

300 DEFSTR A,C Variable names starting with A or C are strings, unless ending in ! # or %.

400 DEFSTR C-E Variable names startin with C, D, or E are strings, unless ending in ! # or %

DEFSTR and the other DEFxxx commands are generally placed near the beginning of programs for clarity, since they change the meaning of variable references without type declaration characters (variable names not ending in ! # % or $).

DELETE "filename"

Deletes a file from disk, just like DELETE on a QL. You may need to make changes to "filename" if the drive or file name syntax is different to that of the QL, e.g. '.' separators which correspond to the use of '_' in a QL filename.

DELETE line_no-line_no

Using a DELETE command with a line number or range of line numbers , rather like DLINE on the QL, except that '-' is used for the range rather than ' TO ' as on the QL.

Examples:

DELETE 100 Delete line 100 only: DLINE 100
DELETE 300-400 Delete lines from 300 to 400 inclusive. Equivalent to DLINE 300 TO 400 on a QL.
DELETE -200 Delete all lines up to and including 200. Equivalent to DLINE TO 200 on a QL.

Device Names

QL device names are usually one or three letter names such as MDV, FLP, WIN and N. Computers running Microsoft-style BASICs generally use PC-style device names, usually a short sequence of letters, a drive number and a colon delimiter.

KYBD: Keyboard. Input only. Equivalent to CON for input.
SCRN: Screen. Out put only. Equivalent to SCR or CON for output on the QL

LPT: Parallel port printer. Equivalent to PAR on QL.
LPTn: Parallel port number (n is usually a low number such as 1 or 2). Equivalent to PAR1 or PAR2 on QL.
COMn: Communications or serial port number. Equivalent to SERn on QL.
CAS1: Cassette tape port.
A: or B: Floppy disk drive, like FLP1_ or FLP2_ respectively.
C: etc Hard disk drive name. C: usually corresponds to WIN1_, although some QL emulators and compatibles allow the WIN drives to be assigned to any drive letter, so it does not necessarily follow that C: is WIN1_, D: is WIN2_ and so on.
PRN: Printer device. On the QL, it might be SER1, SER2, or PAR for example.

DIM

Like on the QL, DIM sets up an array. On most BASICs the arrays start with the subscript zero as the first element of the array, but some BASICs start from 1 - this should make no difference when converting to SuperBASIC which always starts with element zero, apart from wasting a little memory an unused '0' element perhaps.

Where the DIM statement may differ from SuperBASIC is with strings, which may be defined without an explicit maximum length specifier as the last part of a DIM statement.

DIM A$(10) generally sets up a 10 or 11 (depending on whether subscript zero is used by the computer in question) element string array - 10 or 11 lines of text for example. To convert this, you'd need to try to work out from the rest of the program what the length of maximum string to be stored is, and add this maximum length as an extra dimension at the end of the definition.

Examples:

DIM A$(10) convert as DIM A$(10,50) assuming you have established that the strings in the array need to be able to hold up to 50 characters each.

Beware of case sensitive variable and array names where DIM A(10) and DIM a(10) may refer to two separate and distinct arrays - in this case, A() and a() may be completely separate arrays and you'd have to work around this in SuperBASIC by creating different array names, or doing something like adding an underscore '_' to the array name.

Example:

DIM A(10),a(10) could be converted as DIM A(10),_a(10)

DO...LOOP

This is a conditional loop structure similar to the REPeat/END REPeat loops in QL BASIC, and also similar to the WHILE/WEND and REPEAT/UNTIL loop structures in other versions of BASIC.

DO/LOOP may have a pre-test with WHILE or UNTIL clauses at the DO end of the loop, or a post-test with a WHILE or UNTIL clause at the LOOP end of the loop structure.

```
REM press any key loop - Post-Test
DO
  a$=INKEY$
LOOP WHILE a$=""
```

The above example loop keeps going until you press a key. In QL BASIC, it could be written like this. Note that unless you are using SBASIC, REPeat loops needs a control variable name, and you may need to invert the test condition - the easy way to do this is to use NOT as in this example.

```
REPeat loop
  a$ = INKEY$
  IF NOT(a$="") THEN EXIT loop
END REPeat loop

REM increment with Post-Test
x=0
DO
x=x+1
LOOP UNTIL x=10

in QL BASIC:

x=0
REPeat loop
  x = x + 1
  IF x=10 THEN EXIT loop
END REPeat loop
```

The tests can be done at the DO end of the loop to prevent execution of any of the loop content if the test is already met by the time the loop starts:

```
DO WHILE INKEY$=""
LOOP
```

These can be converted to QL BASIC by placing the test just inside the REPeat end of the loop.

```
REPeat loop
  IF INKEY$<>"" THEN EXIT loop
END REPeat loop
```

DRAW

DRAW "string" draws an object as specified by "string". The text contained in "string" is a simple graphics definition language, where the action is specified by a single letter followed possibly by a space and a value.

U n Move up
D n Move down
L n Move left
R n Move right
E n Move diagonally up and right
F n Move diagonally down and right
G n Move diagonally down and left
H n Move diagonally up and left

n in each of the preceding commands indicates the distance to move. The number of points moved is n times the scaling factor (set by the S command)

M x,y Move absolute or relative. If x has a plus sign (+) or a minus sign (-) in front of it, it is relative. Otherwise, it is absolute.

The following two prefix commands may precede any of the above movement commands.

B Move, but don't plot any points
N Move, but return to the original position when finished.

The following commands are also available:

A n Set angle n. n may range from 0 to 3, where 0 is 0 degrees, 1 is 90, 2 is 180 and 3 is 270.

C n Set colour n

S n Sets scale factor. n may range from 1 to 255. n divided by 4 is the scale factor. For example, if n=1, then the scale factor is 1/4. The scale factor multiplied by the distances given with the U D L R E F G H and relative M commands gives the actual distance moved. The default value is 4 so the scale factor is 1.

X var; Execute substring. This allows you to execute a second string from within a string,

In all of these commands, the n, x, or y argument can be a constant like 123 or it can be =variable; where variable is the name of a numeric variable. The semicolon after the variable name is required when you use a variable this way, or int he X command. Otherwise a semicolon is optional between commands. Spaces are ignored int he string. For example, variables could be used in a move command this way:

M+=X1;,-=X2

Variables can also be specified inthe form VARPTR$(variable), instead of =variable; . This is useful in programs which were to be compiled later. For example:

```
One Method       Alternative Method

DRAW "XA$;"      DRAW "X"+VARPTR$(A$)
DRAW "S=SCALE;"  DRAW "S="+VARPTR$(SCALE)
```

The X command was commonly used for two pruposes. The first and most obvious was to allow command string sof longer than 255 characters. The second was to split up complex drawings into simpler parts, e.g. if drawing a person, the arms, head, legs and body could be separate strings.

When coordinates which are out of range are given to DRAW, the coordinate which is out of range is given the closest valid value. In other words, negative values become zero and and points to the right or below the screen are changed to the screen edge values.

There is no direct equivalent to this DRAW command in QL BASIC, although it might be possible to write a very simple parser which breaks up the string and calls suitable LINE and LINE_R commands or turtle graphics commands such as MOVE, TURN, TURNTO, PENUP, PENDOWN

EDIT

Equivalent to the same command on a QL - the command just enters the BASIC line editor at the given line number.

ELSE

Similar to the ELSE command in an IF...THEN clause in QL BASIC. Note that in a single line definition, the QL version will require a colon before the ELSE keyword, so something like

```
IF X=2 THEN PRINT"Two" ELSE PRINT"Not two"
```

becomes

```
IF X=2 THEN PRINT"Two" : ELSE PRINT"Not two"
```

ELSE within a multiple line IF...THEN...END IF clause works in the same way as ELSE in that construct in QL BASIC.

Note that if the statement does not contain the same number of THEN and ELSE clauses, each ELSE is matched with the closest unmatched THEN. This may cause some worry in conversion as to whether the QL version will work in the same way in such cases. Here is an example:

```
IF A=B THEN IF B=C THEN PRINT"A=C" ELSE PRINT "A<>C"
```

will not print "A<>C" when A<>B.

ELSEIF

Used as an alternative to starting multiple IF THEN ELSE statement for complex or cascaded condition checks, ELSEIF [conditional] THEN provides a means of making more efficient multiple conditional statements in QBASIC. Here is an example. Instead of this:

```
IF C=1 THEN CALL routine1
IF C=2 THEN CALL routine2
IF C=3 THEN CALL routine3
```

you could write:

```
IF C=1 THEN
  CALL routine1
ELSEIF C=2 THEN
  CALL routine2
ELSEIF C=3 THEN
  CALL routine3
END IF
```

Unfortunately, there is no direct equivalent in QL BASIC other than multiple multi-line IF THEN ELSE statements, although with a bit of effort a SELect clause can sometimes be used where simple values and ranges are used.

Here is an example of how to convert the above with multiple IF clauses.

```
IF C=1 THEN
  routine1
ELSE
  IF C=2 THEN
    routine2
  ELSE
    IF C=3 THEN
      routine3
    END IF
  END IF
END IF
```

This particular example is also easily converted using a SELect statement:

```
SELect ON C
  =1:routine1
  =2:routine2
  =3:routine3
END SELect
```

END

Terminates program execution, closes all files and returns to command level. An END statement at the end of a programis optional. END is broadly equivalent to a STOP command in QL BASIC, although STOP does not necessarily close any open files - you would need to add explicit CLOSE statements just before the STOP command in this case. Usefully, if you have Toolkit 2 on the QL system, the CLOSE command is extended such that a CLOSE with no parameter will close allopen BASIC channels with numbers higher than 2. So, END is broadly equivalent to CLOSE:STOP in this case.

EOF

Returns the value -1 (true) if the end of a sequential file has been reached. Broadly equivalent to the QL EOF function, although the QL version returns a value of 1.

EOF(file_number) is roughly equivalent to EOF(#channel) in QL BASIC.

EQV

EQV is an equivalence operator, where the result of the logical expression is true if both values are true or both values are false, but false if one operand is true and the other false. Here is the truth table:

X	Y	X EQV Y
True	True	True
True	False	False
False	True	False
False	False	True

You could write an approximate equivalent in QL BASIC like this. In Microsoft-style BASICs the expression would be written as LET R=X EQV Y whereas the QL BASIC version would be LET R=EQV(X,Y)

```
1000 DEFine FuNction EQV(X,Y)
1010   IF (X<>0 AND Y<>0) OR (X=0 AND Y=0) THEN
1020     RETurn 1
1030   ELSE
1040     RETurn 0
1050   END IF
1060 END DEFine EQV
```

ERASE

This command erases arrays from a program. The command takes a list of array names as its parameter. It can be used to free up memory taken by an array, or to allow an array to be redimensioned on systems where this would normally lead to a Duplicate Definition error. The array name parameter does not require brackets after it as would be needed in some commands.

There is no direct equivalent in QL BASIC, the closest being to redimension an array to a smaller size, e.g.

```
100 DIM array(100,100)
200 REMark use the array
300 DIM array(0)
```

ERL

Returns the line number at which an error occurred. Normally used in error trapping routines. Broadly equivalent to the ERLIN function in QL BASIC, although please note that ERLIN does not work on pre-JS ROM QLs.

ERR

Returns the error code for an error which occurred while running a BASIC program. Broadly equivalent to the QL ERNUM function, although the error codes and messages are different.

ERROR

ERROR number is used to simulate the occurrence of a BASIC error, or to allow error codes to be defined by the user. Can be used to test how a program copes with a specific error situation. There is no direct QL equivalent, although sometimes something like REPORT number:STOP may sometimes be used (note that REPORT may not work on pre-JS ROM QLs).

EXIT

Allows the immediate exit from a subroutine or loop, without processing the rest of that subroutine or loop code. The command EXIT in versions of BASIC such as QBASIC is followed by a second word which signifies the type of structure to be exited.

EXIT DEF	exits from a DEF FN function
EXIT DO	exits from a DO loop, execution continues with the command directly after the LOOP command
EXIT FOR	exits from a FOR loop, execution continues with the command directly after the NEXT command
EXIT FUNCTION	exits a FUNCTION procedure, execution continues with the statement directly after the function call
EXIT SUB	exits a SUB procedure, execution continues with the statement directly after the procedure call

In QL BASIC you will need to convert this as either an EXIT command (if a looping structure such as a FOR or REPEAT loop) or a RETURN statement in the case of a function or procedure.

EXP

EXP(X) returns e (base of natural logarithms) to the power of X. Equivalent to the same function in QL BASIC.

FIELD

FIELD allocates space for variables in a random access file buffer. It is used to define the field widths associated with given variable names.

FIELD #file_number,field_width AS string_variable_name

This allocates "field_width" bytes to data held in the given variable name, and the "field_width AS string_variable_name" part may be repeated as required.

To calculate the total record size, add up the widths of each field defined in this command, e.g.

FIELD #file_no,10 AS A$,20 AS B$

is 10+20 or 30 bytes wide.

There is no direct equivalent in QL BASIC. You will need to manually allocate field widths within the appropriate record size and use a file positioning command to move the file pointer around as required, then use something like PRINT #file_number,string_variable_name$; (note the semi-colon at the end to prevent an extra linefeed being sent to the file which might overwrite the start of the next field). If you have Toolkit 2 on your QL system, you can use the BGET or BPUT commands with backslashes to set the file pointer to move around records and fields.

FILES

FILES displays the name of files residing on a diskette. The FILES command in Microsoft BASIC is similar to the DIR command in DOS.

FILES [filespec] where filespec is a string expression for the file specification (drive name etc). If filespec is omitted, all the files on the DOS default drive will be listed.

In QL BASIC, use the DIR command.

FIX

FIX(X) returns the truncated integer part of X. FIX(X) is equivalent to SGN(X)*INT(ABS(X)). The major difference between FIX and INT in MS BASIC is that FIX does not return the next lower number for negative X. For example, PRINT FIX(-58.75) will

return -58, whereas on the QL PRINT INT(-58.75) will return -59. So we need to write a small function which takes the integer part of the absolute values then changes the sign if negative:

```
DEFine FuNction FIX(X)
  IF X<0 THEN RETurn -INT(ABS(X)): ELSE RETurn INT(X)
END DEFine FIX
```

FN

Used when defining functions, FN is pretty much like the equivalent keyword in QL BASIC. In MS BASIC, the definition is that of a single line function definition only, whereas in QL BASIC it extends over multiple lines. Also, when called, in MS BASIC the function name must be preceded by FN, whereas only the function name is required in QL BASIC.

```
100 DEF FNDOUBLE(X)=2*X
200 PRINT FNDOUBLE(8)
```

On a QL, you would need to rewrite this over multiple lines:

```
100 DEFine FuNction DOUBLE(X)
110   RETurn 2*X
120 END DEFine DOUBLE
200 PRINT DOUBLE(8)
```

FOR

Defines a FOR/NEXT loop, broadly equivalent to the QL version of the command.

```
100 FOR A=1 TO 11 STEP 2
110 PRINT A
120 NEXT a
```

QL BASIC will generally support ending of FOR loops with a NEXT command, although it is better to use END FOR in place of NEXT to indicate the end of a loop:

```
100 FOR A=1 TO 11 STEP 2
110 PRINT A
120 END FOR A
```

Note that FOR loops may have an integer index variable, so allows something like:

```
200 FOR X%=1 TO 4
210 PRINT X%
220 NEXT X%
```

Minerva and SBASIC may allow integer loop control variables, some Sinclair QL ROM versions may not, in which case you'll have to use a floating point control variable (use X instead of X% in

this example). Beware of non-integer calculation results where programs modify the loop control variable value within the loop.

If you plan to compile the converted BASIC program, you may wish to try using an implicit type definition for a variable name as an indirect method of converting FOR/NEXT loops which use integer variables. For QLiberator you will need a DEF_INTEGER statement to declare a variable with a conventionally floating point name type (e.g. X) as actually to be compiled as an integer. For Turbo compiler, the corresponding directive is IMPLICIT%

Inline FOR loops on a QL do not necessarily require a NEXT clause. The MS BASIC loop

```
100 FOR Y=0 TO 19 : PRINT Y : NEXT Y
```

could be converted as just:

```
100 FOR Y=0 TO 19 : PRINT Y
```

When loops are nested in MS BASIC, the NEXT clause can specify more than one variable name, whereas an explicit NEXT or END FOR is required for each variable name in QL BASIC:

```
100 FOR X=0 TO 31
110 FOR Y=0 TO 10
120 PRINT" ";
130 NEXT Y,X
```

Convert that to:

```
100 FOR X=0 TO 31
110 FOR Y=0 TO 10
120 PRINT" ";
130 END FOR Y
140 END FOR X
```

You may encounter NEXT statements with no variables specified. In this case the most recently executed FOR loop counter is updated.

```
100 FOR X=0 TO 31
110 NEXT
```

The loop control variable must be explicitly stated in QL SuperBASIC:

```
100 FOR X=0 TO 31
110 END FOR X
```

FRC

Returns the fractional part of a supplied value, the part left

after removing the whole number part. In other words, the complementary function to INT.

QL BASIC has no direct equvalent to FRC, so we need to write one. Note that as INT generally rounds down negative number, it is wise to use ABS in this function to ensure that FRC works correctly for negative numbers.

```
200 DEFine FuNction FRC(x)
210   RETurn ABS(x)-INT(ABS(x))
220 END DEFine FRC
```

FRE

There are two forms of this function. FRE returns the amount of free memory.

PRINT FRE(0) just returns the amount of free memory in bytes

PRINT FRE("") does a garbage collection before returning the amount of free memory.

Both can be replaced by the Toolkit 2 FREE_MEM function in QL BASIC. If the FREE_MEM function is not implemented on your system, you may be able to use the following PEEK as an approximation, which only works on a standard QL:

```
PRINT PEEK_L(163856)-PEEK_L(163852)
```

FUNCTION and **END FUNCTION**

This command is used in QBASIC to define a multi-line function, broadly equivalent to DEF FN is QL BASIC.

Such a function is essentially the same as a subroutine which returns a value. The return value is created by using the function name as a variable - the return value is then passed to the calling expression.

```
FUNCTION name (parameters)
  REM shared variables declarations here
  rem ...
  name = result
  REM ...
END FUNCTION
```

This corresponds to:

```
DEFine FuNction name(parameters)
  REM ...
  RETurn result
  REM ...
END DEFine name
```

QBASIC lso supports an older form of multi-line function definition, as defined with the DEF FN command.

```
DEF FNname(parameters)
  REM ...
  FNname = RESULT
  REM ...
END
```

This would be converted using the same code in QL BASIC.

GET

GET #file_number,record_number reads a record from a random access file into the random access file buffer. file_number is similar to a channel number in QL BASIC. record_number is usually the nth record stored in a file, up to a highest value of 32767. If record_number is not specified, the next record is fetched from the file.

The conversion method will depend on the way in which the program concerned works and how the file is saved in the first place. Remember that GET (and the opposite, PUT) works via the buffer.

You may be able to convert the program by PRINTing variables to file and using INPUT to retrieve them, although it's possible that fixed width fields may be involved in which case a large rewrite may be needed for QL BASIC which does not handle files in quite the same way.

GET (x1,y1)-(x2,y2),arrayname

This version of GET is a graphics command which reads points from a given area of the screen. It is used together with the PUT command (q.v.) to eprform an action similar to a cut and paste.

x1,y1 coordinates of top left corner of the block to be copied.
x2,y2 coordinates of bottom right corner of the block to be copied.
arrayname numeric array to hold colour values of each pixel in the block grabbed. The array must have the same dimensions as the block to be grabbed.

GET and PUT can be used for animation, for example, where an object is grabbed from the screen to be redrawn in a different position.

An approximate QL BASIC adaptation of GET relies on having a function to read the colour of a pixel on the screen. Such functions are available in some toolkits - we shall assume here you are using such an extension called PIXEL to check the colour

of a pixel.

```
1000 DEFine PROCedure GET2(x1,y1,x2,y2)
1010   w = x2-x1+1 : REMark width of block in pixels
1020   h = y2-y1+1 : REMark height of block in pixels
1030   DIM array(w-1,h-1)
1040   FOR x = 0 to w-1
1050     FOR y = 0 TO h-1
1060       array(x,y) = PIXEL(x,y)
1070     END FOR y
1080   END FOR x
1090 END DEFine GET2
```

The procedure is called GET2 to avoid a clash with the file handling procedure called GET in Toolkit 2. The block is read into an array imaginatively called array(). This block can then be restored with the PUT2 procedure listed under PUT below.

Some versions of this command store the width and height of the block in the first two entries of the array.

GOSUB

GOSUB is generally used in exactly the same way as the GOSUB command in QL BASIC. Additionally, some versions of BASIC allow GOSUB to a label within the program rather than a line number. Either substitute the actual line number for the label, or define a variable of same or similar name to the label name.

GOTO

GOTO is generally used in exactly the same way as the GOTO command in QL BASIC. Additionally, some versions of BASIC allow GOTO to a label within the program rather than a line number. Either substitute the actual line number for the label, or define a variable of same or similar name to the label name. In an IF THEN line involving GOTO lines, the keyword GOTO may sometimes be omitted, but must be included in QL BASIC:

```
1000 IF X=20 THEN 2000
```

Convert as

```
1000 IF X=20 THEN GOTO 2000
```

HEX$

v$ = HEX$(n) returns a string which represents the hexadecimal value of the decimal argument. n is a numeric expression in the range -32768 to 65535. If n is negative, the two's complement form is used. That is, HEX$(-n) is the same as HEX$(65536-n)

You could convert this function using the HEX$ function in Toolkit 2, although you'd need to specifiy the 'width' of the

value by giving the number of bits to use to represent the value. Here is a function written in QL BASIC which more closely mimics the HEX$ function as described here.

```
3000 DEFine FuNction HEX$(dec)
3010   LOCal x,c,c$
3020   x = dec : IF x < 0 THEN x = 65536-x
3030   c$ = ""
3040   REPeat loop
3050     c = x - (INT (x/16) * 16)
3060     IF c < 10 THEN
3070       c$ = CHR$(48+c) & c$ : REMark 0-9
3080     ELSE
3090       c$ = CHR$(55+c) & c$ : REMark A-F
3100     END IF
3110     x = INT (x/16)
3120     IF x = 0 THEN EXIT loop
3130   END REPeat loop
3140   RETurn c$
3150 END DEFine
```

IF

IF clauses in MS BASIC are generally single line clauses, and are generally the same as the equivalent single line IF statements in QL BASIC, albeit with minor variations possible mostly involving the ELSE part.

```
1000 IF X=2 THEN GOTO 2000 ELSE GOTO 3000
```

would need to be converted as having a ':' before the ELSE keyword:

```
1000 IF X=2 THEN GOTO 2000 : ELSE GOTO 3000
```

The 'THEN' before the 'GOTO' may sometimes be omitted in MS BASIC:

```
1000 IF X=2 GOTO 2000
```

Convert by adding a THEN:

```
1000 IF X=2 THEN GOTO 2000
```

IMP

The IMP or Implication operator is a logical operator, with the following truth table for results. Basically, the result of x IMP y is false only if x is true and y is false.

X	Y	X IMP Y
True	True	True
True	False	False
False	True	True

```
False   False   True
```

An approximate equivalent in QL BASIC could be written like this.

```
2000 DEFine FuNction IMP(X,Y)
2010   IF X<>0 AND Y=0 THEN RETurn 0 : ELSE RETurn 1
2020 END DEFine IMP
```

Whereas the original expression would be written as LET R=X IMP Y, the QL BASIC equivalent is a function written as LET R=IMP(X,Y)

INKEY$

Works like the equivalent INKEY$ function in QL BASIC, but does not allow channel numbers or timeouts.

INP

INP(I) returns the byte value read from port number I, which must be in the range 0 to 255. There is no direct equivalent in QL BASIC. INP is the opposite to the OUT statement (q.v.)

INPUT

Like the INPUT command in QL BASIC, this receives input from the keyboard during program execution.

```
INPUT[;]["prompt";] variable[,variable]...
```

When the program sees an INPUT statement, it pauses and displays a question mark on the screen to indicate that it is waiting for data. If a "prompt" is included, the string is displayed before the question mark. You may then enter the required data from the keboard.

INPUT can use a comma instead of a semicolon after the prompt string to suppress the question mark. For example, the statement INPUT"ENTER BIRTHDATE",B$ prints the prompt without the question mark.

As there is no equivalent to this in the QL BASIC version of INPUT, the question mark must be explicitly included in the prompt string if required.

Different versions of BASIC may or may not print spaces before a variable, so you may come across something like INPUT"ENTER";A where you would have to add a space to the prompt string in QL BASIC - INPUT"ENTER ";A

The data that is entered is assigned to the variable(s) given in the variable list. The data items supplied must be separated by commas, and the number of data items must be the same as the

number of variables in the list.

The type of each data item that you enter must agree with the type specified by the variable name. Strings entered in response to an INPUT statement need not be surrounded by quotation marks unless they contain commas or significant leading or trailing spaces.

Some versions of BASIC allow you to press ENTER with no characters inputted in response to INPUT of a numeric variable - the value of zero (0) is then given to the numeric variable.

Some versions of BASIC allow a single quote mark to be included between prompts to indicate a newline between prompt strings, e.g. INPUT "HELLO"'"THERE";A - this can be replaced by a backslash (\) character in QL BASIC:

```
INPUT"HELLO"\"THERE";A
```

INPUT #

Reads data items from a sequential device or file and assigns them to program variables.

```
INPUT #filenum,variable [,variable]...
```

This is similar to INPUT #channel,variable in QL BASIC, although for string items, the first character encountered that is not a space, carriage return or linefeed is assumed to be the start of the string item. If this first item is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second, so a quoted string may not contain a quotation mark as a character.

INPUT$

INPUT$(X,#Y) is a function which returns a string of X characters from the terminal (keyboard) if the file number (#Y) is not from specified, or from file number Y if specified. Unless you have an equivalent toolkit extension such as INPUT$ of Turbo Toolkit (parameters in reverse order - INPUT$(#Y,X) in the Turbo Toolkit version), the easiest way to replace this is to write a function which uses INKEY$ to fetch a string of a given number of bytes from a specified channel:

```
1000 DEFine FuNction INPUT_STRING$(X,Y)
1010   LOCal counter,string$
1020   string$ = ""
1030   FOR A = 1 TO X
1040     string$ = string$ & INKEY$(#Y)
1050   END FOR A
1060   RETurn string$
1070 END DEFine INPUT_STRING$
```

INSTR

INSTR([I,]X$,Y$) is a function which searches for the first occurrence of Y$ within X$ and returns the position at which the match was found, starting from 1, or returns 0 if no match was found. An optional first parameter allows the start position of the search within X$ to be specified.

The QL BASIC version of INSTR is broadly similar, although the parameters are given differently. LET P=INSTR(X$,Y$) may be replaced by LET P=Y$ INSTR X$.

The version with three parameters is a bit more difficult and fiddly to convert, but may be done using string slicing once we know how the MS BASIC version works, here's an example:

```
10 X$="ABCDEB"
20 Y$="B"
30 P=INSTR(4,X$,Y$)
```

This returns a value of 6 into the variable P. In other words, start searching from the fourth character, return the position at which a match was found in the full string - in this case, the last character position (sixth character of Y$).

QL BASIC version:

```
10 X$ = "ABCDEB"
20 Y$ = "B"
30 P = Y$ INSTR X$(4 TO LEN(X$) : IF P > 0 THEN P = P+4-1
```

INT

Exactly the same as the INT function on the QL.

KEY

Sets or displays soft keys, similar to altkeys or hotkeys in QL BASIC. This command lets you assign a string to any of the 10 function keys.

KEY n,x$ assigns the string expression x$ to function key number n. In QL BASIC, the function keys take the following key codes, bearing in mind that F6-F10 are SHIFT F1-SHIFT F5 respectively on a QL, as it only has 5 function keys.

KEY 1 to 5 are QL key codes 232, 236, 240, 244, and 248.
KEY 6 to 10 are QL key codes 234, 238, 242, 246, and 250.

So, KEY 6,x$ would be translated to QL basic as ALTKEY CHR$(234),x$ for example.

Other variations of this command are KEY ON, which displays the key definitions on the 25th line of the screen. KEY OFF turns

off this display. KEY LIST displays all the key definitions, rather like a HOT_LIST command in QL BASIC.

KILL

KILL "filename" is the same as DELETE "filename" in QL BASIC.

LEFT$

LET A$=LEFT$(X$,J) returns a string comprising the leftmost J characters of X$. This is equivalent to LET A$=X$(1 TO J) in QL BASIC. If J is 0, a null string is returned, while a value of J greater than the length of X$ means that the full length of X$ is returned. Both of these cases may cause an error in QL SuperBASIC. So the safest way is to control the range of values J can take in this example:

```
REM convert LEFT$(X$,J)
IF J=0 THEN
  A$=""
ELSE
  IF J>LEN(X$) THEN J=LEN(X$)
  A$ = X$(1 TO J)
END IF
```

LEN

LEN(A$) is a function which returns the length of the string given as a parameter. It works the same as the equivalent function in QL BASIC.

LET

The use of the LET command is broadly the same as in SuperBASIC. LET is optional in many BASICs, where you can use either 100 LET A=0 or 100 A=0.

Some BASICs permit multiple assignments to be made with one LET statement, e.g. LET A=1,B=2,C=3 so you need to break these apart into multiple statements such as LET A=1 : LET B=2 : LET C=3

LINE INPUT

LINE INPUT [;] [<"prompt string">;] <string variable>

Its purpose is to input an entire line of up to 254 characters to a string variable, without the use of delimiters. The prompt string works broadly like that on a QL BASIC INPUT command. Unlike standard MS BASIC INPUT, a '?' is not auotmatically printed. The optional ';' between LINE INPUT and the prompt string suppresses the carriage return typed by the user to end the input line is not echoed at the terminal.

LINE INPUT is broadly similar to string INPUT on the QL, except

that the use of the semi-colon to suppress the carriage return may be simulated by adding a semi-colon at the end:

```
100 INPUT ;"Enter something";S$
```

becomes

```
100 INPUT "Enter something";S$;
```

LINE INPUT#

LINE INPUT# file_number,string_variable$ is used to fetch a line of up to 254 characters of data from a sequential data file to a string variable. LINE INPUT# is similar to INPUT# in QL BASIC, although you may need to be careful about how the data is organised and what end of line characters are used if you work directly from the original MS BASIC data files.

LINE INPUT #file_number,S$ corresponds to INPUT#channel,S$ in QL BASIC.

LIST

Lists the given line number or range of line numbers to the screen. Equivalent to LIST on the QL, but uses '-' to specify range of line numbers instead of ' TO '.

Examples:

```
LIST            List all of the program to screen. Use the same command on a QL.
LIST n          Lists line number n to the screen. Use the same command on a QL.
LIST m,n,o      Lists the given line numbers only to the screen. Use the same command on a QL.
LIST m-n        Lists all line numbers from m to n inclusive to the screen, like LIST m TO n on a QL.
```

LLIST

Like LIST, but sends output to a printer device, or sometimes to a file. Use the OPEN/LIST#/CLOSE equivalent on a QL, e.g. to list program to SER1, use:

```
OPEN #3,"SER1":LIST #3: CLOSE #3
```

On a QL you could also use SAVE SER1, interestingly!

LOAD

The LOAD command corresponds to the QL BASIC LOAD command, apart from syntactical differences in the filename itself of course. The LOAD command may take a ",R" additional parameter, which signifies that the program should start running after it has

loaded, so LOAD "filename",R corresponds to LRUN "filename" in QL BASIC.

LOC

LOC file_number returns the record number just read or written from a GET or PUT statement with random files. If the file was opened but no disk I/O has been performed yet, LOC returns a 0. With sequential files, LOC returnsthe number of sectors (128-byte blocks) read from or written to the file since it ws OPENed. Such files do not work in the same way on the QL so there is no directly equivalent method of converting this.

LOCATE

LOCATE [row][,[col] [,[cursor] [,[start] [,stop]]]]
Positions the cursor on the active screen. Optional parameters turn the blinking cursor on and off and define the size of the blinking cursor.

row is a numeric expression in the range 1 to 25 (or as many lines as fit on the screen in this version of BASIC). It indicates the screen line number where you want to place the cursor. Note: some versions of BASIC may use 0 to 24 instead of 1 to 25.

col is a numeric expression in the range 1 to 40 or 1 to 80, depending on which screen width. It indicates the screen column number where you want to place the cursor. Note: some versions of BASIC may use 0 to 39 or 0 to 79 instead of 1 to 40 or 1 to 80.

cursor is a value indicating whether the cursor is visible or not. A 0 (zero) indicates off, 1 (one) indicates on.

start is the cursor starting scan line. It must be a numeric expression in the range 0 to 31.

stop is the cursor stop scan line. It also must be a numeric expression in the range 0 to 31.

cursor, start and stop do not apply when the computer is in graphics mode.

A LOCATE command with just two parameters can be converted using an AT command in QL BASIC. So LOCATE y,x would become AT y-1,x-1.

The "cursor" parameter could be emulated in QL BASIC by using a separate command such as CURSEN (cursor enable) or CURDIS (cursor disable) from Toolkit 2. The "cursor" parameter can be used with the "row" and "col" parameters, or without - either of these examples would be valid:

```
LOCATE row, col, cursor
LOCATE ,,cursor
```

There is no way to emulate the "start" and "stop" parameters.

LOF

LOF(file_number) returns the length (number of bytes) allocated to the file identified by "file_number". With communications files, LOF returns the amount of free space in the input buffers. If you have Toolkit 2 on your QL system, FLEN(#file_number) will perform an approximately equivalent function.

LOG

LOG(X) returns the natural logarithm of X. X must be greater than zero. Corresponds to LN(X) in QL BASIC.

LOWER, **UPPER** and **MIXED**

These functions alter the case of a given string. LOWER converts the string entirely to lower case, UPPER converts it to upper case and MIXED converts the first character too upper case and the remainder of the string to lower case. Similar to what is sometimes called 'Sentence case'

```
LOWER('HELLO') returns "hello" (without the quotes)
UPPER('Hello') returns "HELLO"
MIXED('hELLO') returns "Hello"
```

Some BASICs may append a '$' symbol to the names (i.e. LOWER$, UPPER$, MIXED$) to indicate that they are type string.

QL BASIC has no direct equivalent to these, so we have to write equivalent functions such as these.

```
2350 DEFine FuNction LOWER (u$)
2360   LOCal t$,a,c
2370   REMark convert u$ to lower case
2380   t$ = u$ : REMark working copy of u$
2390   FOR a = 1 TO LEN(t$)
2400      c = CODE(t$(a))
2410      IF c > 64 AND c < 91 THEN t$(a) = CHR$(c+32)
2420   END FOR a
2430   RETurn t$
2440 END DEFine LOWER
2450 :
2460 DEFine FuNction UPPER (u$)
2470   LOCal a,c,t$
2480   REMark convert u$ to upper case
2490   t$ = u$ : REMark working copy of u$
2500   FOR a = 1 TO LEN(t$)
2510     c = CODE(t$(a))
```

```
2520     IF c > 96 AND c < 123 THEN t$(a) = CHR$(c-32)
2530   END FOR a
2540   RETurn t$
2550 END DEFine UPPER
2560 :
2570 DEFine FuNction MIXED(u$)
2580   LOCal a,c,t$
2590   REMark convert string to have upper case first letter
only
2600   t$ = u$ : REMark working copy
2610   c = CODE(t$)
2620   IF c > 96 AND c < 123 THEN t$(1) = CHR$(c-32)
2630   FOR a = 2 TO LEN(t$)
2640     c = CODE(t$(a))
2650     IF c > 64 AND c < 91 THEN t$(a) = CHR$(c+32)
2660   END FOR a
2670   RETurn t$
2680 END DEFine MIXED
```

LPOS

Returns the current position of the print head within the printer buffer for the parallel port number given as a parameter in the function vall v=LPOS(n).

There is no equivalent function in QL BASIC.

LPRINT

Similar to the PRINT command, except that it sends output to a printer instead of the screen. To convert this, a channel should be open to a printer so that you can use PRINT#channel to ensure output is sent to the printer. You could also write a simple procedure called LPRINT to open a temporary channel to the printer to send output to the printer as a quick and dirty conversion method if it's only the occasional single item, instead of a range of items as might be the case with the full allowed PRINT-style syntax.

```
DEFine PROCedure LPRINT(text$)
  OPEN #3,"SER1": REMark printer port- change as required
  PRINT #3,text$
  CLOSE #3
END DEFine LPRINT
```

An extnded version, LPRINT USING "string_expression",item$ allows formatting of printed output in a similar way to the PRINT USING command.

LSET

```
LSET <string_variable>=<string_expression>
RSET <string_variable>=<string_expression>
```

The purpose of this keyword is to move data from a random file buffer (in preparation for a PUT statement).

If <string_expression> requires fewer bytes than were FIELDed to <string_variable>, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions). If the string is too long for the field, characters are dropped from the right. Numeric values must be converted to strings before they are LSET or RSET. See MKI$, MKS$ and MKD$

```
Examples: 1000 LSET A$=MKS$(AMOUNT)
          1010 LSET D$=DESC($)
```

LSET or RSET may also be used with a nonfielded string variable to left-justify or right-justify a string in a given field. For example, the program lines

```
110 A$ = SPACE$(20) : RSET A$=N$
```

right-justify the string N$ in a 20-character field. This can be very handy for formatting printed output.

There is no direct equivalent to this in QL BASIC, but it is fairly easy to approximate its function via QL string handling functions.

The basic idea is to try to force strings to a given length so that they will fit in a field of a random access file with that length. So if the field is designed to hold a 20 character string, you would need to pad out or truncate a string to exactly 20 characters long by either adding the required number of spaces, or chopping characters off the appropriate end of the string if it is already too long.

So we could write procedures like this:

```
1000 DEFine PROCedure Lset(variable$,length,expression$)
1010   variable$ = expression$
1020   IF LEN(variable$) < length THEN
1030     variable$ = variable$ & fill$(' ',length -
LEN(expression$))
1040   ELSE
1050     IF LEN(variable$) > length THEN variable$=variable$(1
TO length)
1060   END IF
1070 END DEFine Lset
```

In order to write an Rset procedure, you would have to work out if the string is to be right justified by adding spaces between words in the string, or merely byadding spaces to the left of the string.

LTRIM$

This function strips any leading spaces from a string whose name is given in the function's paramater, which may be a text constant or a string variable or a single array element.

```
n$ = LTRIM$("      Hello"
```

returns "Hello" (without the quote marks).

n$ = LTRIM$(s$) returns a copy of s$ but without any spaces at the start of the string.

LTRIM$ may be written as a function in QL BASIC like this:

```
1000 DEFine FuNction LTRIM$(s$)
1010   LOCal t$,a
1020   REMark return copy of s$ minus any leading spaces
1030   t$ = "" : REMark in case s$ is all spaces
1040   FOR a = 1 TO LEN(s$)
1050     IF s$(a) <> " " THEN t$ = s$(a TO LEN(s$)) : EXIT a
1060   END FOR a
1070   RETurn t$
1080 END DEFine LTRIM$
```

The corresponding function RTRIM$ (q.v.) strips any spaces from the right hand side of the string.

Stripping spaces from both sides of a string is achieved by combining the two functions.

```
PRINT LTRIM$(RTRIM$(s$))
```

MAX and **MIN**

These functions return the maximum or minimum respectively of the two values supplied as parameters. Some forms of these functions may only handle numeric values, while others may also handle strings.

100 LET value=MAX(2,4) would return the value 4, the higher of the two values given. In the case of equality, such as LET value=MAX(3,3) the value of the first parameter is returned, in this case 3.

MIN works in a similar way for the minimum value, e.g. value = MIN(2,4) returns the value 2.

The paramters may be numbers or numeric variables. Some versions of BASIC allow for expressions too, e.g. LET value=MIN(VAL("1"),VAL("2")).

QL BASIC has no direct equivalents, although it is easy to write similar functions.

```
100 DEFine FuNction MAX(val1,val2)
110   IF val1 > val2 THEN
120     RETurn val1
130   ELSE
140     RETurn val2
150   END IF
160 END DEFine MAX
170 :
180 DEFine FuNction MIN(val1,val2)
190   IF val1 < val2 THEN
200     RETurn val1
210   ELSE
220     RETurn val2
230   END IF
240 END DEFine MIN
```

MERGE

Similar to the MERGE command in QL BASIC, except that the format of the filename parameter may obviously vary in line with the differences between the operating systems.

MID$

MID$(S$,N,M) is a string slicing function which returns a section of the string S$ which is M characters long starting at subscript N. If M is not specified, it defaults to the last character of S$.

Examples:
A$=MID$("HELLO",2,3) returns the three letters ELL
A$=MID$("HELLO",4) returns LO

The best way to convert this is to write a function like this:

```
DEF FN Mid$(S$,N,M)
  RETurn S$(N TO N+M-1)
END DEF Mid$
```

That example doesn't cater for the case where the third parameter is not specified.

MKD$ MKI$ MKS$

These three functions convert numeric values to string values. Any numeric value that is placed ina random access file buffer with LSET or RSET statements must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single precision number to a 4-byte string. MKD$ converts a double precision number to an 8-byte string.

The syntax is:

```
LET I$=MKI$(integer_expression)
```

```
LET S$=MKS$(single_precision_expression)
LET D$=MKD$(double_precision_expression)
```

There is no direct equivalent to these functions in QL BASIC, and QL BASIC doesn't distinguish between single and double precision numeric values, so you will have to try to figre out exactly what the program is doing and write appropriate code to handle the program code concerned.

If it is simply a matter of making strings of fixed length to write to data files, you may be able to write code based on the BPUT and PUT keywords from Toolkit 2 to write QL data values to files in fixe dlength formats (2 bytes for integers, 6 bytes for floating point values on QL).

MOD

MOD gives the integer value that is the remainder of an integer division. Its use is the same as MOD in QL BASIC.

MOTOR

The MOTOR command turns the cassette player motor off (0) or on (non-zero). Not required on the QL.

NAME

Changes the name of a file on disk.

```
NAME <old_filename> AS <new_filename>
```

is broadly equivalent (apart from differences in filename syntax) to

```
RENAME old_filename TO new_filename
```

in QL BASIC. Note that RENAME is a Toolkit 2 extension and not built into original unexpanded Sinclair QL BASIC.

NEW

Like NEW on a QL, this clears out an existing program and resets all variables. Usually, the only difference to the QL version is that the string space allocated by CLEAR N is not reset.

NEXT

NEXT is used as the closing statement in a FOR/NEXT loop, corresponding broadly to the use of END FOR in QL BASIC, although in most cases the QL BASIC NEXT statement may be used as an equivalent too. In some versions of BASIC, in nested FOR/NEXT loops the NEXT keyword may take more than one loop variable name, for example:

```
FOR x=1 TO 10
FOR y=1 TO 2
REM any code here
NEXT y,x
```

In QL BASIC, NEXT y,x should be replaced by NEXT y:NEXT x or (better) END FOR y: END FOR x

In some versions of BASIC, the loop variable name may be omitted, in which case the most recently executed FOR loop counter is incremented. QL SuperBASIC has no equivalent for this - FOR loop names must be specified, although SBASIC does allow such omission of loop names:

```
FOR x=1 TO 10
PRINT x
NEXT
```

is perfectly valid in SBASIC, but not in SuperBASIC.

NOT

The logocal NOT is broadly the same as the use of NOT in QL BASIC. A bitwise NOT such as x=NOT y corresponds to the use of the double tilde in QL BASIC: x= ~~ y
where the two's complement of an integer is the bit complement plus one, or NOT x=-(x+1)

OCT$

v$=OCT$(n) returns a string which represents the octal value of the decimal argument. n is a numeric expresion in the range -32768 to 65535. If n is negative, the two's complement form is used, i.e. OCT$(-n) is the same as OCT$(65536-n).

Here are a pair of functions written in QL BASIC to convert decimal numbers into octal (OCT$) and vice versa (DECIMAL).

```
8000 DEFine FuNction OCT$(dec)
8010   LOCal x,c$
8020   x = dec : IF x < 0 THEN x = 65536-x
8030   c$ = "" : REMark octal character string
8040   REPeat loop
8050     c$ = CHR$(48+x-(INT(x/8)*8)) & c$
8060     x = INT (x/8)
8070     IF x <= 0 THEN EXIT loop
8080   END REPeat loop
8090   RETurn c$
8100 END DEFine OCT$
8110 :
8120 DEFine FuNction DECIMAL(bs$)
8130   LOCal d,l,n,v
8140   d = 0
8150   l = LEN(bs$)
8160   FOR n = 1 TO l
8170     v = CODE(bs$(n))-48
```

```
8180     d = d + 8^(l-n) * v
8190   END FOR n
8200   RETurn d
8210 END DEFine DECIMAL
```

ON

The ON keyword may be used in ON ERROR GOTO <line_number> clauses, or in ON...GOTO and ON...GOSUB selection clauses.

ON ERROR GOTO has no direct equivalent in QL BASIC, although it may be possible to use WHEN ERRor to provide an approximation of the function, but be aware that early QL ROMs do not include support for the WHEN ERRor structure and it is also bugged in some early ROM versions. Basically, when an error occurs (indicated by the value of ERROR becoming non-zero) program flow jumps to an indicated line number to provide an action to handle the error condition.

```
100 ON ERROR GOTO 1000
```

may be converted as:

```
100 WHEN ERRor
101 GOTO 1000
102 END WHEN
1000 PRINT"An error has occurred."
```

In Microsoft-style BASICs an ON ERROR GOTO clause may be turned off by executing something like ON ERROR GOTO 0. In QL BASIC, an empty WHEN ERRor clause is an approximate equivalent for turning off WHEN ERRor error trapping.

ON x GOTO and ON x GOSUB are equivalent to the same structures in QL BASIC. The value of x (starting from 1) decides which line number in the list after GOTO or GOSUB is jumped to. Microsoft BASICs can handle a list withvalues from 1 to 255 for the line number selection and if the value given is out of range (less than 0, greater than 255, or higher than the amount of line numbers given) the program continues from the next executable statement, whereas a QL BASIC program will stop with an error message.

OPEN

Command format:
OPEN <mode>,[i]<file number>,<filename>[,<reclen>]
Purpose: To allow I/O to a disk file.

A disk file must be OPENed before any disk operation can be performed on that file. OPEN allocates a buffer for I/O to the file and determines the mode of access that will be used with the buffer.

<mode> is a string expression whose first character is one of the following:
O Specifies sequential output mode.
I Specifies sequential input mode.
R Specifies random input/output mode.
<file number> is an integer expression whose value is between 1 and 15. The number is then associated with the file for as long as it is OPEN and is used to refer other disk I/O statements to the file.
<filename> is a string expression containing a name that conforms to the operating system's rules for disk filenames.
<reclen> is an integer expression which if included, sets the record length for random files. The default record length is 128 bytes.
Note
Example: A file can be OPENed for sequential input or random access on more than one file number at a time. A file may be OPENed for output, however, on only one file number at a time.

It is difficult to convert this precisely to QL BASIC, although the OPEN command and its variants, for example, the use of OPEN_NEW to open a new file for output, or OPEN_IN to open a file just for input.

OPTION BASE

OPTION BASE n declares the minimum value for array subscripts. n is 0 or 1 and OPTION BASE should be used before any arrays are declared. There is no equivalent in QL BASIC - arrays always start from a 0 subscript.

OR

The use of the logical OR is the same as that in QL BASIC, whereas the bitwise operator OR corresponds to the use of the double bar operator in QL BASIC. In other words, x = y OR z corresponds to x = y || z in QL BASIC.

Order Of Arithmetic Operations

This refers to the order of priority in whch the elements of calculations are carried out. this may vary somewhat between versions of BASIC and some adjustment by hand may be required, e.g. adding parentheses (brackets) to force the order of precedence.

Operations in the innermost level of parentheses are performed first, then evaluation proceeds to the next level out, etc. Operations on the same nesting level are generally performed according to the following hierarchy:

Exponentiation: A ^ B
Negation: -X
*, / (left to right)

+, - (left to right)
<, >, =, <=, >=, <> (left to right)
NOT
AND
OR

OUT

Format: OUT I,J
Where I and J are integer expressions in the range a to 255, OUT I,J sends an integer value J to the output port I.

There is no direct equivalent in QL BASIC, although where memory mapped devices are concerned it may be possible to use POKE I,J.

PAINT

Fills an area of the screen with the selected colour. The command syntax is:

PAINT (x,y) [,paint [,boundary]]

(x,y) are the coordinates of a point within the area to be filled in. The coordinates may be given in absolute or relative form. This point is used as a starting point.

paint is the colour to be painted with. The default is the foreground colour.

boundary is the colour of the edges of the figure to be filled in.

The figure to be filled in is the figure with edges of "boundary" colour. The figure is filled in with the colour "paint"

The starting point of PAINT must be inside the figure to be painted. If the specified point already has the colour "boundary" then PAINT will have no effect.

There is no direct equivalent to PAINT in QL BASIC (the closest being the FILL command which works in a different way), although a few add-on toolkits may have a PAINT or equivalent command.

PEEK

PEEK(a) reads a byte value from the address 'a'. The value read may be from 0 to 255. Although this function is the same as the PEEK function in QL BASIC, the memory locations read from are unlikely to be the same - some knowledge of the system will be needed to work out how best to convertthis, e.g.where colours or characters are read directly from screen memory on different computers.

Some versions of BASIC allow the address value to be specified as a hexadecimal number by the use of &H as a prefix to the hex value, e.g. PEEK(&H5A00)

PEN

This keyword is used for reading a light pen connected to the PC. There is no equivalent in QL BASIC.

PLAY

PLAY "string" plays music as specified by "string". The command implements a concept similar to DRAW by embedding a "tune definition language" into a character string.

string is a string expression consisting of single character music commands

The single character commands in PLAY are:

A to G with optional #, +, or -
        Plays the indicated note in the current octave. A number sign (#) or plus sign (+) afterwards indicates a sharp, a minus sign (-) indicates a flat. The #, +. or - is not allowed unless it corresponds to a black key on a piano. For example, B# is an invalid note.

O n   Octave. Sets the current octave for the following notes. There are 7 octaves, numbered 0 to 6. Each octave goes from C to B. Octave 3 starts with middle C. Octave 4 is the default octave.

N n   Plays note n. n may range from 0 to 84. In the 7 possible octaves, there are 84 notes. n=0 means rest. This is an alternative way of selecting notes besides specifying the octave (O n) and the note name (A-G).

L n   Sets the length of the following notes. The actual note length is 1/n. n may range from 1 to 64. The following table may help explain this:

Length	**Equivalent**
L1	whole not
L2	half note
L3	one of a triplet of three half notes (1/3 of a 4 beat measure)
L4	quarter note
L5	one of a quintuplet (1/5 of a measure)
L6	one of a quarter note triplet
.	
.	
.	
L64	sixty-fourth note

The length may also follow the note when you want to change the length only for the note. For example, A16 is equivalent to L16A

P n Pause (rest). n may range from 1 to 64, and figures the length of the pause in the same way as L (length).

. (dot or period). After a note, causes the note to be played as a dotted note. That is, its length is multiplied by 3/2. More than one dot may appear after the note, and the length is adjusted accordingly. For example, "A.." will play 9/4 as long as L specifies, "A..." will play 27/8 as long, etc. Dots may also appear after a pause (P) to scale the pause length in the same way.

T n Tempo. Sets the number of quarter notes in a minute. n may range from 32 to 255. The default is 120.

MF Music foreground. Music (created by SOUND or PLAY) runs in foreground. That is, each subsequent note or sound will not start until the previous note or sound is finished. You can press Ctrl-Break to exit PLAY. Music foreground is the default state.

MB Music background. Music (created by SOUND or PLAY) runs in background instead of in foreground. That is, each note or soundis placed in a buffer allowing the BASIC program to continue executing while music plays in the background. Up to 32 notes (or rests) may be played in the background at a time.

MN Music normal. Each note plays 7/8 of the time specified by L (length). This is the default setting of MN, ML and MS.

ML Music legato. Each note plays the full period set by L (length).

MS Music staccato. Each note plays 3/4 of the time specified by L.

X variable;

Executes specified string.

In all of these commands the n argument can be a constant like 12 or it can be =variable; where variable is the name of a variable. The semicolon (;) is required when you use a variable in this way, and when you use the X command. Otherwise a semicolon is optional between commands, except a semicolon is not allowed after MF, MB, MN, ML, or MS. Also, any blanks in "string" are ignored.

Variables can also be specified in the form VARPTR$(variable), instead of =variable; which is often useful where a program is to be compiled. For example:

```
One Method       Alternative Method

PLAY "XA$;"      PLAY "X"+VARPTR$(A$)
PLAY "O=I;"      PLAY "O="+VARPTR$(I)
```

X can be used to store a "subtune" in one string and call it repetitively with different tempos or octaves from another string.

As the PLAY command can be difficult to grasp and there is no direct equivalent in QL BASIC without writing a small program to emulate the actions of PLAY "string", here is an example to help set you on the way.

```
10 REM little lamb
20 MARY$="GFE-FGGG"
30 PLAY "MB T100 03 L8;XMARY$;P8 FFF4"
40 PLAY "GB-B-4; XMARY$; GFFGFE-."
```

POINT

Reads a colour value from the specified co-ordinates on the screen. For example, c=POINT(x,y) reads the colour value of the pixel at x across and y down the screen. There is no direct equivalent in QL BASIC, although some BASIC extensions have been written to extract a colour value from the screen, for example in the DIY Toolkit series.

POKE

POKE a,d writes the byte value d to the address a. The value written may be from 0 to 255. Although this function is the same as the POKE command in QL BASIC, the memory locations written to are unlikely to tbe the same - some knowledge of the system will be needed to work out how best to convert this, e.g. where colours or characters are written directly to screen memory on different computers.

Some versions of BASIC allow the address value to be specified as a hexadecimal number by the useof &H as a prefix to the hex value, e.g. POKE &H5A00,d

POS

POS(X) returns the current cursor position. The leftmost position is 1. X is a dummy argument.
Example: IF POS(X)>60 THEN PRINT CHR$(13)

There is no direct equivalent in QL BASIC, although the information required to convert this function is contained within the channel definition blocks for the window channel concerned and may be read using the DIY Toolkit function CHAN_W% for a given channel number. The horizontal cursor position in pixels may be read from offset 34 decimal (hex 22). This starts

from 0 and is in pixel units, so to get a text character position across, divide by the number of pixels per character in the current character size, e.g. 6 for CSIZE 0,0 text. The character spacing for a given window channel may be checked with CHAN_W%(#channel,38), so to read the current text position as characters across the screen, use something like:
LET pos=CHAN_W%(#channel,34) DIV CHAN_W%(#channel,38)

On computers where the POS function starts from 1 for the leftmost position, you should add 1 to this result:

```
LET pos=1+(CHAN_W%(#channel,34) DIV CHAN_W%(#channel,38))
```

or write it as a function in QL BASIC:

```
1000 DEFine FuNction POS(chan)
1010   RETurn 1+(CHAN_W%(#chan,34) DIV CHAN_W%(#chan,38))
1020 END DEFine POS
```

PRINT

In general, works almost exactly the same as the QL PRINT command, the main exception being that in cases where no ambiguity would result, punctuation between items to be printed (; or ,) can be omitted. These need to be included in QL PRINT statements.

Example:

```
PRINT 2"Hello"   Must be converted to PRINT 2;"Hello" on a QL.
```

There may be differences in syntax when PRINT output is routed to a channel or device.

PRINT may take position and format identifiers - see AT, TAB and USING.

When printing numbers, or numeric variables, some BASICs may add a space before and/or after a number, to clearly separate numbers from any surrounding text.

Example:

```
A=3:PRINT "Buy";A;"items."
```
prints the avriable with spaces both sides in this case
Buy 3 items.

So to convert exactly to the QL, you would need to add the extra spaces at the appropriate sides of the variable A in the PRINT statement:

```
A=3:PRINT "Buy";" ";A;" ";"items"
```

A bit of an extreme example, but serves to illustrate what you

need to do.

Some versions of BASIC allow you to use a query symbol as a shorthand way of entering a print command in the BASIC editors. Often this is expanded to a PRINT keyword, but not always, so you may come across listings with lines such as 100 ?"Hello" which is obviously the same as 100 PRINT"Hello"

PRINT @

This is a version of PRINT which allows you to locate the cursor at given text coordinates. The value after the '@' symbol may in some cases be a pair of coordinates (X,Y or Y,X - both forms exist). or a single coordinate which specifies the total number of characters from the top left of the screen - (Y times line_width)+X.

For example, a Tandy TRS-80 can have a 64 column or 32 column screen width. So PRINT @64 could refer to the start of the second line down in 64 column mode (like AT 1,0 on a QL) or to the start of third line down in 32 column mode (like AT 2,0 on a QL).

Some versions of BASIC may use a comma or semi-colon between the @value and the string or variable to be printed. Some may omit the separator completely!

See also LOCATE command.

Example 1:

The single value version is best converted after reference to the number of characters across the screen mode of the computer in question:

PRINT @N,"Hello" convert this as AT (N DIV screen_width), (N MOD screen_width) : PRINT"Hello"

Example 2:

PRINT @X,Y;"Hello" convert as AT Y,X : PRINT"Hello"

Example 3:

PRINT @Y,X"Hello" (note missing separator after X) convert as AT Y,X : PRINT "Hello"

PRINT #

PRINT #filenum, [USING v$;] listofexps writes data sequentially to a file.

filenum is the number used when the file was opened for output (like a channel number in QL BASIC)

v$ is a string expression comprised of formatting characters as described under USING below.

listofexps
is a list of numeric and/or string expressions that will be written to the file.

PRINT # is very similar to PRINT # in QL BASIC, except that filenum and channel numbers may be different, and the punctuation between items ot be printed may be more flexible in QL BASIC - normally, you'd only use semicolons between items in PRINT # statements in Microsoft BASIC to send numeric values or variables ot a file - note that like PRINT a space may be sent between two variables separated by a semicolon.

PSET PRESET

Draws a point at the specified position on the screen. The difference between PSET and PRESET is that if no colour (third parameter) is specified, PSET defaults to the forgeground (ink) colour, while PRESET defaults to the background (paper) colour. Both commands take the same two or three parameters.

```
PSET (x,y) [,colour]
PRESET (x,y) [,colour]
```

x and y are the coordinates of the point to be set, and may be absolute or relative values depending on the system concerned. Relative coordinates are specified by preceding the open bracket with a STEP keyword, e.g. PSET STEP(10,-20)

colour is optional and the colour values depend on the system concerned.

An approximate conversion is to write a short procedure in QL BASIC to use BLOCK to colour a pixel, although you will need some knowledge of the system concerned to know the range of pixel sizes and colour numbers to use.

```
8500 DEFine PROCedure PSET(x,y,colour)
8510   BLOCK 1,1,x,y,colour
8520 END DEFine PSET
8530 :
8540 DEFine PROCedure PRESET(x,y,colour)
8550   BLOCK 1,1,x,y,colour
8560 END DEFine PRESET
```

Depending on which screen window you use to draw the graphics on a QL, you may wish to add a channel number to the BLOCK commands - without one, they default to channel #1.

PUT

PUT [*]<file number>[,<record number>]

Write a record from a random buffer to a random disk file. <file number> is the number under which the file was OPENed. If <record number> is omitted, the record will assume the next available record number (after the last PUT). The largest possible record number is 32,767. The smallest record number is 1.

As the QL filing system is so different to that used on these other computer system, it is hard to give a definitive conversion method. Best to study how the filing system works and devise an individual conversion based on that.

PUT (x,y) , array [, action]

As a graphics command, PUT writes a block of pixels onto the screen, with the colour information and block size specified by 'array', and action may be PSET, PRESET, AND, OR, XOR. The array is likely to have been grabbed into the array originally by the GET command (q.v.)

x,y Coordinates of top left corner of image to be transferred.

array Two dimensional array with colour value for each pixel.

action PSET - draw pixel (see PSET and PRESET above)
PRESET - draw pixel (see PSET and PRESET above)
AND - used when you want to transfer the image only if an image already exists under the transferred image.
OR - is used to superimpose the image onto the existing image (like OVER 1 in QL BASIC)
XOR - a special mode which may be used for animation. XOR causes the points on the screen to be inverted where a point exists in the array image, so if the image is PUT against a complex background twice, the background is restored unchanged, which allows you to move an image around without obliterating the background. A bit like OVER -1 on a QL.
Note that the default is XOR for the fourth parameter.

Here is a simple QL BASIC equivalent, slightly renamed to avoid a name clash with the other use of the PUT command. The bracketing around the parameters has to be different, the word used for action must be in quotes, and there is no equivalent to the AND action.

```
1000 DEFine PROCedure PUT2 (x,y,array,logic$)
1010   w = DIMN(array,1)
1020   h = DIMN(array,2)
1030   IF logic$ == 'OR' THEN OVER 1
1040   IF logic$ == 'XOR' THEN OVER -1
1050   FOR a = 0 to h-1
1060     FOR b = 0 TO w-1
```

```
1070       BLOCK 1,1,b,a,array(b,a)
1080     END FOR b
1090   END FOR a
1100   OVER 0
1110 END DEFine PUT2
```

RANDOMISE

RANDOMISE [<expression>] reseeds the random number generator. If <expression> is omitted, Microsoft BASIC suspends program execution and asks for a value by printing:

```
Random Number Seed (-32768 to 32767)?
```
before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is RUN. To change the sequence of random numbers every time the program is RUN, place a RANDOMIZE statement at the beginning of the program and change the argument with each RUN.

This command is broadly the same as that in QL BASIC, although if <expression> is omitted in QL BASIC, the random number generator is reseeded with a value based on the current clock value.

Note that the sequence for a given seed value is not likely to be the same in QL BASIC as the same seed in other BASICs.

READ

READ <list of variables>

As in QL BASIC, with minor differences this reads values from DATA statements assigns them to variables. The variable used for READ must be of the same data type as the data value contained in the DATA statement, although some BASICs allow unquoted strings to be used in DATA statements. Such strings must be quoted in QL BASIC.

```
1000 DATA HELLO
```

must be altered like this for QL BASIC:

```
1000 DATA "Hello"
```

Some versions of BASIC only allow one data type in a READ statement, or even only single variables, which is why you may come across something like:

```
1000 READ a : READ b : READ c$
```

It is possible to join these together into a single clause in QL BASIC:

```
1000 READ a,b,c$
```

REM

The comment command REM is the same as the command REMark in QL BASIC. Some versions of BASIC allow REMark to be abbreviated to an apostrophe:

```
1000 LET a=b 'assign b to a
```

is the same as

```
1000 LET a=b : REMark assign b to a
```

REPEAT UNTIL

REPEAT and UNTIL form a kind of repeat loop where a test expression to determine the termination of the loop is performed at the REPEAT statement.

```
1000 X=0
1010 REPEAT
1020 X=X+!
1030 UNTIL X=10
```

This can be converted to a REPEAT / END REPEAT loop in QL BASIC, but the test condition will need to be written as an EXIT statement test just before the END REPEAT, and the loop given a name (unless you are using SBASIC, which permits unnamed loops).

```
1000 X=0
1010 REPEAT loop
1020 X=X+1
1030 IF X=10 THEN EXIT loop
1040 END REPEAT loop
```

RESET

The RESET command closes all diskette files and clears the system buffer. There is no direct equivalent in QL BASIC, other than a CLOSE statement for all open file channels.

On some systems, a RESET command is a graphics command which resets the colour of a pixel to the screen background colour - see the SET command for further details.

RESTORE

RESTORE [<line number>]

To allow DATA statements to be read from a specified line. After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the

program. If <line number> is specified, the next READ statement accesses the first item in the specified DATA statement.

Equivalent to the RESTORE command in QL BASIC.

RESUME

RESUME continues program execution after an error recovery procedure has been performed. Up to four formats may be encountered:

RESUME	Execution resumes at the statement which caused the error.
RESUME 0	Execution resumes at the statement which caused the error.
RESUME NEXT	Execution resumes at the statement immediately following the one which caused the error.
RESUME line_number	Execution resumes at line_number.

For RESUME or RESUME 0, use a WHEN ERRor statement with a RETRY command to resume execution at the line which caused the error:

```
100 WHEN ERRor
110 PRINT'Oops'
120 RETRY
130 END WHEN
140 INPUT'Enter a number:';num%
150 REMark reset error processing
160 WHEN ERRor
170 END WHEN
```

For RESUME NEXT, use a similar structure, but use the CONTINUE command in place of the RETRY command in line 120.

For RESUME line_number, use a GOTO line_number in place of the RETRY or CONTINUE commands in line 120.

RETURN

The RETURN statement(s) in a subroutine cause Microsoft BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, should logic dictate a return at different points in the subroutine. The use of the RETURN statement in subroutines in Microsoft-style BASICs is the same as the use of RETURN in QL BASIC.

REVERSE$ or **REVERSE**

REVERSE$("string") is a function which reverses the characters of a string, for example, PRINT REVERSE$("Hello") prints "olleH". In some versions of BASIC the function is called REVERSE (i.e. without the '$' symbol).

QL BASIC has no equivalent function, but it is easy to write a similar function as shown in this example. This version can reverse the digits of a number as well as strings thanks to typeless QL parameters (n in this case, which can be a number or a string).

```
100 DEFine FuNction REVERSE$(n)
110   LOCal a,t1$,t2$
120   t1$ = n : t2$ = n
130   FOR a = 1 TO LEN(t1$)
140     t2$(a) = t1$(LEN(t1$)-a+1)
150   END FOR a
160   RETurn t2$
170 END DEFine REVERSE$
```

RIGHT$

RIGHT$(X$,I) is a function which returns the rightmost I characters of string X$. If I is equal to the number of characters in X$ (LEN(X$)), returns X$. If I=O, the null string (length zero) is returned.

To create an equivalent function in QL BASIC, you can write a function called RIGHT$ like this:

```
3000 DEFine FuNction RIGHT$(x$,i)
3010   IF i < 1 THEN RETurn ""
3020   IF i > LEN(x$) THEN RETurn x$
3030   RETurn x$(LEN(x$)-i+1 TO LEN(x$))
3040 END DEFine RIGHT$
```

RND

RND returns a random number between 0 and 1 (the next number in the pseudo-random sequence of random numbers) when used without a parameter, or with a parameter value of greater than 0, e.g. RND(2). RND(0) repeats the last random number generated. In some versions of BASIC, RND followed by a positive whole number returns a random integer in the range of 1 up to the value given, e.g. RND(9) returns a random number between 1 and 9 (inclusive). In QL BASIC, replace RND(10) with either RND(1 TO 10) or RND(10)+1

RND*10 will generate a random decimal number from 0 to 9 inclusive. Convert that with exactly the same code in QL BASIC. An expression like (RND*10)+1 will generate a random decimal number from 1 to 10 inclusive (that may also be converted with identical code in QL BASIC), while an expression such as INT(RND*10)+1 will generate a random integer from 1 to 10

inclusive. While the latter example may be written using the same code in QL BASIC, you may find it clearer and more brief to use RND(1 TO 10) instead. In QL BASIC, the statement RND(n) generates a random whole number from 0 to n inclusive.

In QL BASIC, RND without a parameter gets a floating point random number in the range 0 to 1. RND(n) gets an integer between 1 and n inclusive. RND(m TO n) gets an integer between m and n inclusive.

So, to convert the version of RND with a positive number parameter, i.e. RND(n) which generates a random number from 1 to n inclusive, use RND(1 TO n) in QL BASIC.

Summary:

PC Basic	QL BASIC	Notes
RND	RND	Random number from 0 to 1
RND(0)	no equivalent	Repeat last random number
RND*n	RND*n	Random decimal number from 0 to n-1
(RND*n)+1	(RND*n)+1	Random decimal number from 1 to n
INT(RND*n)+1	RND(1 TO n) or	Random integer from 1
INT(RND*n)+1	to n	
RND(n)	RND(1 TO n)	Random integer from 1 to n (in some early BASICs may be 0 to n-1)

RSET

see LSET.

RTRIM$

This function returns a copy of a string minus any trailing spaces at the right hand side of the string. It is related to the LTRIM$ function (q.v.) which strips spaces from the left hand side of a string.

An equivalent funtion to RTRIM$ may be written like this in QL BASIC.

```
3350 DEFine FuNction RTRIM$(s$)
3360   LOCal t$,a
3370   REMark return a copy of s$ minus any leading spaces
```

```
3380   t$ = "" : REMark in case s$ is all spaces
3390   FOR a = LEN(s$) TO 1 STEP -1
3400     IF s$(a) <> " " THEN t$ = s$(1 TO a) : EXIT a
3410   END FOR a
3420   RETurn t$
3430 END DEFine RTRIM$
```

RUN

Equivalent to the QL RUN command, except that in most BASICs the RUN command also does the equivalent of a CLEAR command to reset all variables. So you may have to do something like CLEAR:RUN on a QL. RUN can take an optional line number.

RUN "filename"

Equivalent to LRUN "filename" on a QL.

SAME

SAME is a function which tests if two supplied numbers or strings are the same. This sounds very similar to a simple '=' statement and in some cases it is. The main difference is that when comparing strings, the case of text characters does not matter - "hello" is considered to be the same as "HELLO"

The function returns a TRUE value (1 or -1, depending on the system concerned) if the two parameters match, or 0 if not matched.

The easiest way to write a similar function in QL BASIC is to use the PARTYP function of Toolkit 2 to establish if the parameters are numbers or strings, then use either '=' for number comparison or '==' for case insensitive string matching. The QL BASIC functions returns a TRUE value of 1 for a match.

```
3750 DEFine FuNction SAME(x,y)
3760   IF PARTYP(x) > 1 AND PARTYP(y) > 1 THEN
3770     REMark both numbers
3780     RETurn (x=y)
3790   ELSE
3800     REMark not both numbers, compare as strings
3810     RETurn x == y
3820   END IF
3830 END DEFine SAME
```

SAVE

SAVE <filename>[{,A!,P}]

Save a program file on disk.

<filename> is a quoted string that conforms to your operating system's requirements for filenames. (Your operating system may

append a default filename extension if one was not supplied in the SAVE command). If <filename> already exists, the file will be written over.

Use the A option to save the file in ASCII format. Otherwise, Microsoft BASIC saves the file in a compressed binary format. ASCII format takes more space on the disk, but some disk access requires that files be in ASCII format. For instance, the MERGE command requires an ASCII format file, and some operating system commands such as LIST may require an ASCII format file.

Use the P option to protect the file by saving it in an encoded binary format. When a protected file is later RUN (or LOADed), any attempt to list or edit it will fail.

The SAVE command does the same function in QL BASIC, although SAVE always saves in ASCII format, and does not automatically overwrite existing files. Also, there is no QL BASIC equivalent for the P option.

SCREEN

This command sets the screen attributes to be used by subsequent statements. It is system-specific, so often would not be required in QL BASIC conversions, although some features may be relevant.

SCREEN [mode] [,burst] [,page] [,vpage]

mode is a numeric expression resulting in a low integer value which describes the text (0) mode or graphics mode.

burst is a numeric expression resulting in a true or false value. It enables colour. In text mode (mode value = 0) a false (zero) value disables colour (black and white only) and a true (non-zero) value enables colour.

apage (active page) is an integer expression in the range 0-7 for width 40, or 0 to 3 for width 80. It selects the page to be written to by screen output statements and is valid in text modes only.

vpage (visual page) selects which page is to be displayed on the screen, in the same way as apage. The visual page may be different to the active page. vpage is valid in text mode (mode=0) only. If omitted, vpage defaults to apage.

Here are some standard values for "mode", taken from a QBASIC manual.

Mode	Resolution	Colours/Palette	Adapters
0	Text only	16/16 (64 EGA)	All
1	320x200	4/(16 EGA VGA)	CGA EGA VGA
2	640x200	2/(16 EGA VGA)	CGA EGA VGA

3	720x348	2/2	HGA
4	640X400	2/16	Olivetti M24/M28
7	320x200	16/16	EGA VGA
8	640x200	16/16	EGA VGA
9	640x350	16/64	EGA VGA
10	640x350	4/9 (monochrome)	EGA VGA
11	640x480	2/262144	VGA
12	640x480	16/262144	VGA
13	320x200	256/262144	VGA

SELECT CASE

This is a structure very similar to the QL BASIC SELect ON clauses, allowing selection of actions on the basis of individual values or ranges of values. A multi-way conditional branching structure if you like. Broadly speaking, the two structures are similar with only minor variations, although one major difference is that SELECT CASE can handle strings, whereas QL BASIC SELect ON can only handle numbers.

```
SELECT CASE x
  CASE 2
  ...
  CASE 3,6
  ...
  CASE ELSE
  ...
END SELECT
```

In QL BASIC, this corresponds to

```
SELect ON x
  =2
    ...
  =3,6
    ...
  =REMAINDER
    ...
END SELect
```

Note that if the selection variable (x in this example) is an integer such as x% you may find that some QL ROM versions such as AH and JM don't allow the use of an integer variable name in a SELECT ON clause. SBASIC does, many QDOS ROM versions will allow the use of integer variables in compiled BASIC programs but not interpreted BASIC programs. For compatibility purposes allow you to imply that a variable is floating point when interpreted (for compatibility) and treated as integer variables when compiled. The Turbo compiler allows the use of IMPLICIT% x for this purpose, and the QLiberator compiler allows the use of DEF_INTEGER x for this purpose.

The string selection version is a little more complex to

reproduce in QL BASIC. Here is an example of a string SELECT CASE structure:

```
SELECT CASE n$
  CASE "Fred"
  ...
  CASE "Joe"
  ...
  CASE ELSE
  ...
END SELect
```

Probably the easiest way to achieve a similar coding in QL BASIC is the use of multiple IF...THEN statements:

```
IF n$ = "Fred" THEN
  ...
ELSE
  IF n$ = "Joe" THEN
    ...
  ELSE
    REMark CASE ELSE
    ...
  IF
END IF
```

If the version of BASIC concerned allows for case independent string matching, you may prefer to use the '==' equivalence operator in the QL BASIC version to allow upper case lower case to be matched.

SET

SET and RESET are commands used in some versions of BASIC having low resolution graphics. SET x,y sets a pixel x pixels across the screen and y pixels down the screen. A corresponding function POINT interrogates the screen to return the colour value, or on a monochrome screen TRUE or FALSE values for a set or reset pixel respectively.

In general, a SET command may be converted using a BLOCK command, but you will need to know the resolution of the screen to know how to scale the BLOCK size per pixel. As an example, older computers with low resolution screens may have graphics based on 2 pixels across and three down per text character, so if using CSIZE 0,0 on the QL you will need to use a BLOCK 2,3,x,y,colour_value command to convert a SET command, and a BLOCK 2,3,x,y,paper_colour for a RESET command.

SGN

This is a function which returns the sign of a number. It returns -1 if the value is less than 0, 1 if greater than 0 and zero if the value is 0. The QL does not have a SGN function, but

it is easy to write a QL BASIC function to provide this keyword, using a couple of logical tests to compare the range of values.

```
1000 DEFine FuNction SGN(value)
1010   RETurn -1*(value<0)+(value>0)
1020 END DEFine SGN
```

SIN

SIN(angle) returns the sine of the angle specified in radians, as in QL BASIC. If the version of BASIC works in units of degrees, use SIN(RAD(angle)) in QL BASIC. To convert degrees to radians without using the RAD function, multiply the angle value in degrees by PI/180:

```
LET sine=SIN(degrees*PI/180)
```

SLEEP

SLEEP n suspends execution of a program for n seconds. May be simulated with a PAUSE command in QL BASIC (although an extension such as SUSPEND_TASK from Turbo Toolkit may be better). Note that PAUSE works in units of frames, so n seconds should be converted to 50*n on the QL. That is, SLEEP n is converted as PAUSE 50*n.

SOUND

This command generates sound through the speaker.

SOUND freq, duration

freq is the desired frequency in Hertz (cycles per second). It must be a numeric expression in the range 37 to 32767.

duration is the desired duration in clock ticks. The clock ticks occur 18.2 times per second. duration must be a numeric expression in the range 0 to 65535.

QL BEEP duration is measured in units of 72 microseconds compared to 5495 microseconds for SOUND. QL BEEP pitch values are not directly related to musical notes or particular pitch, but here is a table showing approximate relationships between note and the pitch parameter of BEEP duration,pitch in QL BASIC. Outside this range the pitch values become less reliable.

Note	**Pitch**	**Note**	**Pitch**
A	41	A	15
A#	38	A#	14
B	36	B	12
C	33 (middle C)	C	11
C#	31	C#	10
D	28	D	9
D#	26	D#	8
E	24	E	7

F	22	F	6
F#	20	F#	5
G	19	G	4
G#	17	G#	3

When the SOUND statement produces a sound, the program continues to execute until another SOUND statement is reached. If duration of the new SOUND statement is zero, the current SOUND statement that is running is turned off. Otherwise the program waits until the first sound completes before it executes the new statement.

If no SOUND statement is running, SOUND x,0 has no effect.

The tuning note, A, has a frequency of 440. The following table correlates notes with their frequencies for two octaves on either side of middle C.

Note	Frequency	Note	Frequency
C	130.810	C*	523.250
D	146.830	D	587.330
E	164.810	E	659.260
F	174.610	F	698.460
G	196.000	G	783.990
A	220.000	A	880.000
B	246.940	B	987.770
C	261.630	C	1046.500
D	293.660	D	1174.700
E	329.630	E	1318.500
F	349.230	F	1396.900
G	392.000	G	1568.000
A	440.000	A	1760.000
B	493.880	B	1975.500

* middle C. Higher (or lower) notes may be approximated by doubling (or halving) the frequency of the corresponding note in the previous (next) octave.

To create periods of silence, use SOUND 32767,duration.

The duration for one beat can be calculated from beats per minute by dividing the beats per minute into 1092 (the number of clock ticks in a minute)

The next table shows typical tempos in terms of clock ticks:

Tempo		**Beats/Minute**	**Ticks/Beat**
very slow	Larghissimo		
.	Largo	40-60	27.3-18.2
.	Larghetto	60-66	18.2-16.55
.	Grave		
.	Lento		
.	Adagio	66-76	16.55-14.37
slow	Adagietto		
.	Andante	76-108	14.37-10.11

```
medium   Andantino
.        Moderato          108-120          10.11-9.1
fast     Allegretto
.        Vivace
.        Veloce
.        Presto            168-208          6.5-5.25
very fast        Prestissimo
```

SPC

The function SPC(n) skips n spaces in a PRINT statement. n must be in the range 0 to 255. If n is greater than the defined width of the device (e.g. a 60 column screen or 80 column printer), then the value used is n MOD width. SPC may only be used with PRINT, LPRINT and PRINT # statements. If the SPC function is at the end of the list of data items, then BASIC does not add a carriage return, as though the SPC function had an implied semicolon after it. Also, see the SPACE$ function below.

Dependent on whether the version of BASIC concerned uses SPC like a TAB statement or a SPACE$ function, this could be converted to QL BASIC using a PRINT TO n; statement for the former, or a FILL$ function for the latter as described under SPC$ and SPACE$ below.

SPC$ SPACE$

These two functions return a string consisting of n spaces. Both versions are the same, just different names used in different versions of BASIC.

In QL BASIC, the FILL$ function can be used to return n spaces: FILL$(" ",n), or write a function with a similar name to the source BASIC like this, dependent on whether the source BASIC uses SPC$ or SPACE$:

```
1000 DEFine FuNction SPC$(n)
1010   RETurn FILL$(" ",n)
1020 END DEFine
```

SQR

Returns the square root of a given value. Equivalent to the SQRT function in QL BASIC. You could write a function called SQR to provide a function of the same name.

```
2000 DEFine FuNction SQR(value)
2010   RETurn SQRT(value)
2020 END DEFine SQR
```

STEP

Part of the FOR / NEXT loops structure, used the same way as STEP in QL BASIC.

STICK

Returns the x and y co-ordinates of two joysticks. The syntax is v=STICK(n) where

n is a numeric expression in the range 0 to 3 which affects the result as follows
0 - returns the x coordinate of joystick A
1 - returns the y coordinate of joystick A
2 - returns the x coordinate of joystick B
3 - returns the y coordinate of joystick B

Note: STICK(0) retrieves all four values for the coordinates, and returns the value for STICK(0). STICK(1), STICK(2) and STICK(3) do not sample the joystick - they get the values previously retrieved by STICK(0).
The range of values for x and y depends on your particular joysticks.

Since the QL has no specific commands to read such joysticks, a direct conversion is not possible. QL joysticks work by emulating the keyboard - the arrow keys for joystick 1, and the function keys for joystick 2, both of which may be read with INKEY$ to provide joystick control for programs.

STOP

The use of STOP is broadly equivalent to the same command in QL BASIC. Other BASICs tend to have both a STOP and an END command, the main difference between them being that STOP does not close files on return to command level.

STR$

STR$ is a function which returns a string version of a supplied value, e.g. LET x$=STR$(y).

In some versions of BASIC, the string return may contain a leading space, reserved for the sign of the number. A positive value may omit the sign, so that it is returned as a space, whereas the space becomes a minus sign for negative numbers. PRINT STR$(3.5) might return " 3.5" whereas PRINT STR$(-3.5) might return "-3.5".

In QL BASIC, this function is replaced by string coercion whereby numeric and string values are converted implicitly. So the above example would simply be replaced by LET x$=y although you could write a function in QL BASIC to provide a function of the same name for simplicity of conversion:

```
3000 DEFine FuNction STR$(value)
3010   RETurn value
3020 END DEFine STR$
```

The function type is denoted by the '$' symbol at the end of the function name in QL BASIC. If this is not sufficiently explicit, a simple coercion using a local string variable could be used prior to the RETurn statement:

```
3000 DEFine FuNction STR$(value)
3010   LOCal t$
3020   t$ = value
3030   RETurn t$
3040 END DEFine STR$
```

If you wish to emulate the leading space feature, simply do a test for positive or neagtive values and add a leading space if not negative:

```
3000 DEFine FuNction STR$(value)
3010   LOCal t$
3020   if value < 0 then t$ = value : ELSE t$ = " "&value
3030   RETurn t$
3040 END DEFine STR$
```

STRIG

STRIG returns the status of the joystick buttons (triggers).

STRIG(n) ON must be executed to enable trapping by the ON STRIG(n) statement. After STRIG(n) ON every time the program starts a new statement, BASIC checks to see if the specified button has been pressed.

IF STRIG(n) OFF is executed, no testing or trapping takes place. Even if the button is pressed, the event is not remembered. If a STRIG(n) STOP statement is executed, no trapping takes place. However, if the button is pressed, it is remembered so that an immediate trap takes place when STRIG(n) ON is executed.

The values of n for the statements above may be 0, 2, 4, or 6 and indicates the button to be trapped as follows:

0	button A1
2	button B1
4	button A2
6	button B2

Used as a function v=STRIG(n)

n is a numeric expression in the range 0 to 3. It affects the value returned by the function as follows:

0 Returns -1 if button A1 was pressed since the last STRIG(0) function call, returns 0 if not.

1	Returns -1 if button A1 is currently pressed, returns 0 if not.
2	Returns -1 if button B1 was pressed since the last STRIG(2) function call, returns 0 if not.
3	Returns -1 if button B1 is currently pressed, returns 0 if not.

Some systems allow four buttons to be read from the joystick, using the values 4 to 7 for buttons A2 and B2. Same return values.

STRING$

Returns a string of length n whose characters have the ASCII code m or the first character of x$. So you may encounter two versions:

```
v$ = STRING$(n,m) or
v$ = STRING$(n,x$)
```

Either version can be replaced by the FILL$ function in QL BASIC, with parameter type adjusted accordingly. v$ = STRING$(n,m) would become v$ = FILL$(CHR$(m),n) while v$ = STRING$(n,x$) would become v$ = FILL$(x$,n)

If a program uses both forms of the command, you may be able to write a function which tests the type of the second parameter using the Toolkit 2 function PARTYP to test the type of the second parameter and adjust the action of the function accordingly:

```
4000 DEFine FuNction STRING$(n,m)
4010   IF PARTYP(m) > 1 THEN
4020     REMark numeric
4030     RETurn FILL$(CHR$(m),n)
4040   ELSE
4050     REMark null or string
4060     RETurn FILL$(m,n)
4070   END IF
4080 END DEFine STRING$
```

SUB and **END SUB**

The SUB command is used to create a named subroutine, rather like DEF PROC in QL BASIC. END SUB corresponds to END DEF in QL BASIC

```
SUB name
  ...list of instructions
END SUB
```

corresponds to:

```
DEFine PROCedure name
  ...list of instructions
END DEFine name
```

The named subroutine defined by SUB is called either by name alone or by using the CALL command followed by the subroutine name (CALL name using the example above). The call command is optional, but its use makes it clearer that the code is calling a named subroutine.

SWAP

This command swaps the values of two variables, as you might need to do when sorting or reordering data. The syntax of the command is SWAP variable1,variable2.

Any type of variable (integer, string, single precision, double precision, array element) may be swapped, but both must be of the same type or a Type Mismatch error is caused.

A simple procedure can be written in QL BASIC to emulate this command, thanks to typeless procedure parameters this should cover all data types, although you will need to change tmp to tmp$ if swapping strings, although if you have Toolkit 2 you could probably adapt it using PARTYP to check the type of one of the parameters and an IF...THEN to provide separate versions for strings and numbers, but remember that coercion could mean you could just use tmp$ which would happily hold numeric values, albeit a little slower.

```
2000 DEFine PROCedure SWAP(variable1,variable2)
2010   LOCal tmp
2020   tmp = variable1
2030   variable1 = variable2
2040   variable2 = tmp
2050 END DEFine SWAP
```

SYSTEM

Puts the computer into Monitor mode for calling and examining machine code programs. On some computers this enters an operating system "shell" mode. No direct equivalent on the QL without additional software to provide a QDOS or SMSQ shell. QL emulators may have a command to quit the emulator and return to the host operating system, e.g. QPC_EXIT on the QPC2 emulator.

TAB

Used with PRINT, this moves the cursor to the specified position across the current line. TAB is followed by a number from 0 to 255 representing the horizontal psotion to move to. If the value given is greater than the screen width, it is reduced modulo the

screen width.

No punctuation is usually required after TAB, although it may make PRINT TAB commands which include items to be printed easier to read. Some BASICs insist that the TAB value is enclosed in brackets, e.g. TAB(3)

The equivalent modifier on a QL is the TO separator, e.g. PRINT TO 10; is the same as PRINT TAB(10);

Examples:

```
PRINT TAB(10)    Move to column 10 across the screen line.
PRINT TAB(10);   Same.
PRINT TAB 10;    Same.
PRINT TAB(10);"OK"       Move to column 10 across the screen
line and print the letters OK there.
```

TAN

Returns the tangent of x. The angle x is specified in radians normally, although in QL BASIC you could use the RAD function to convert an angle specified in degrees, or multiply the angle in degrees by PI/180. Used in the same way as the function TAN in QL BASIC.

THEN

Part of the IF...THEN structure in most versions of BASIC. In most cases the same as the use of THEN in QL BASIC, although an IF...THEN GOTO ... statement in some versions of BASIC may omit the keyword THEN:
IF...GOTO...
The keyword THEN is always required in this structure in QL BASIC, although in some cases it may be replaced by a colon in single line IF clauses, or omitted in multi-line IF structures. So, IF x=2 GOTO 1000 would be replaced by IF x=2:GOTO 1000

TIME$

TIME$ sets or retrieves the current time.

Its format may vary from system to system, the most common format is a function to return the time as an 8 digit string of the form hh:mm:ss which can be replaced by a slice of the last 8 ditis of the DATE$ function in QL BASIC.

```
5000 DEFine FuNction TIME$
5010   LOCal t$
5020   t$ = DATE$
5030   RETurn t$(13 TO 20)
5040 END DEFine TIME$
```

When used as a command to set the time, the format is TIME$=x$,

where x$ may be given in one of the following forms:

hh Set the hour in the range 0 to 23. Minutes and seconds default to 00.

hh:mm Set the hour and minutes. Minutes must be in the range 0 to 59. Seconds default to 00.

hh:mm:ss Sets the hour, minutes and seconds. Seconds must be in the range 0 to 59.

There is no direct equivalent to this command in QL BASIC, as you can only set the time and date together using the SDATE command with all six parameters specifying date and time.

TO

Part of the FOR/NEXT loop structure, used in the same way in QL BASIC.

TROFF

Turns off the line number Trace system, whereby the currently executing line number is displayed on the screen to help you diagnose program flow.

TRON

Turns on the line number Trace system, whereby the currently executing line number is displayed on the screen to help you diagnose program flow. There is no direct equivalent command on the QL unless you use add-on software.

UBOUND

UBOUND is a QBASIC function to find the upper limit of an array passed to it (the highest subscript of that array). For example, DIM A(100):PRINT UBOUND(A,1) prints the value 100.

The second parameterspecifies which dimension of the array is to be checked.

UBOUND corresponds to DIMN in QL BASIC.

USING

PRINT USING statements allows you to specify a format for printing string and numeric values. It can be used in many applications such as printing report headings, accounting reports, cheques ... or wherever a specific print format is required.

The PRINT USING statement uses the following format:

PRINT USING string; value

String and value may be expressed as variables or constants. This statement will print the expression contained in the string, inserting the numeric value shown to the right of the semicolon as specified by the field specifiers.

The following field specifiers may be used in the string:

This sign specifies the position of each digit located in the numeric value. The number of # signs you use establishes the numeric field. If the numeric field is greater than the number of digits in the numeric value, then the unused field positions to the left of the number will be displayed as spaces and those to the right of the decimal point will be displayed as zeros.

The decimal point can be placed anywhere in the numeric field established by the # sign. Rounding-off will take place when digits to the right of the decimal point are suppressed.

The comma - when placed in any position between the first digit and the decimal point - will display a comma to the left of every third digit as required. The comma establishes an additional position in the field.

** Two asterisks placed at the beginning of the field will cause all unused positions to the left of the decimal to be filled with asterisks. The two asterisks will establish two more positions in the field.

$$ Two dollar signs placed at the beginning of the field will act as a floating dollar sign. That is, it will occupy the first position preceding the number.

**$ If these three signs are used at the beginning of the field, then the vacant positions to the left of the number will be filled by the * sign and the $ sign will again position itself in the first position preceding the number.

^^^^ Causes the number to be printed in exponential (E single-precision or D double-precision) format.

+ When a + sign is placed at the beginning or end of the field, it will be printed as specified as a + for positive numbers or as a - for negative numbers.

- When a - sign is placed at the end of the field, it will cause a negative sign to appear after all negative numbers and will appear as a space for positive numbers.

% spaces % To specify a string field of more than one character, % spaces % is used. The length of the string field will be 2 plus the number of spaces between the percent signs.

! Causes the Computer to use the first string character of the current value.

Any other character that you include in the USING string will be displayed as a string literal.

If you have access to the original BASIC interpreter for which the program was written, a simple three line program can be used to test how the various options work:

```
100 INPUT A$, B
110 PRINT USING A$;B
120 GOTO 100
```

The USING modifier can have subtle differences and sometimes additional facilities on different computers, so some degree of familiarisation with the BASIC in question (or at least access to manuals) will be required.

The nearest QL equivalent command is the PRINT_USING command in Toolkit 2. While there is some considerasble overlap of formatting, there can be differences too and some care will be required to get a close match in functionality. You are advised to read the Toolkit 2 manual section 13 for a full description of the PRINT_USING command to ensure you understand how it works so that you can use the equivalent formatting string in cases where the QL version differs slightly from the other version.

Another way of using the PRINT USING statement is with the string field specifiers "!" and % spaces %.

Examples:

```
PRINT USING "!"; S$
PRINT USING "%    %"; S$
```

The "!" sign will allow only the first letter of the string to be printed. This would be quivalent to PRINT S$(1) or PRINT_USING "#",S$ in SuperBASIC.

The "% spaces %" allows spaces +2 characters to be printed. In other words, 1 character for each of the % symbols plus the number of spaces. Again, the string and specifier can be expressed as string variables. This would be equivalent to PRINT S$(1 TO 1+spaces+1) or PRINT_USING FILL$('#',1+spaces+1),S$

This can get really complicated when multiple values or strings are to be printed using more complex forms. Here is one example, where by using more than one ! symbol, the first letter of each string in the PRINT list will be printed with spaces corresponding to the spaces inserted between the ! symbols in the USING string. In this example, the first character of each string is printed with spaces between them:

```
PRINT USING "! ! !";"ABC","DEF","GHI"
results in
A D G
```

To cope with this, you may need to split multiple parameters to single commands:

```
PRINT_USING "#","ABC"
PRINT_USING "#","DEF"
PRINT_USING "#","GHI"
```

or become inventive with the format string, so that it contains a format pattern or field for each of the following elements of the command:

```
PRINT_USING "# # #","ABC","DEF","GHI"
```

USR

USR [<digit>] (X) is a function which calls the specified user assembler routine numbered from 0 to 9 (0 assumed if not specified), passing the value of X to the routine.

LET A=USR(0,B) calls user assembler routine number 0, and passes the value of B to it.

There is no direct equivalent in QL BASIC - the user assembler code will be processor specific and the nearest equivalent is likely to be writing an extension function in 680x0 assembler code to do the same job.

VAL

VAL(X$) returns the numerical value of string X$. The VAL function also strips leading blanks, tabs and linefeeds from the argument string. String coercion in QL BASIC means this command is not needed - LET A$="-3":LET B=VAL(A$) is simply converted as LET A$="-3":LET B=A$

The opposite function to convert a number to a string is STR$.

In QL BASIC, this function is replaced by string coercion whereby numeric and string values are converted implicitly. So the above example would simply be replaced by LET y=x$ although you could write a function in QL BASIC to provide a function of the same name for simplicity of conversion:

```
3000 DEFine FuNction VAL(value$)
3010   RETurn value$
3020 END DEFine STR$
```

The function type is denoted by the lack of a '$' symbol at the end of the function name in QL BASIC. If this is not sufficiently explicit, a simple coercion using a local variable

could be used prior to the RETurn statement:

```
3000 DEFine FuNction VAL(value$)
3010   LOCal v
3020   v = value$
3030   RETurn v
3040 END DEFine VAL
```

VARPTR

This command may take two forms.

1. LET AD=VARPTR(X)
This function returns the address of the first byte of data associated with the given variable name. There is no direct equivalent in QL BASIC.

2. LET F=VARPTR(#file_number)
This function returns the starting address of the disk I/O buffer assigned to the given file number (or channel), or for random files returns the address of the FIELD buffer assigned to file_number. No direct equivalent in QL BASIC.

VARPTR$

Returns a character form of the address of a variable in memory. It is promarily for use with PLAY and DRAW in programs that were later compiled. The command syntax is LET v$=VARPTR$(variable). All simple variables should have been assigned before calling VARPTR$ for an array element, because addresses of arrays change whenever a new simple variable is assigned. VARPTR$ returns a 3 byte stringin the form:

```
Byte 0   Byte 1           Byte 2

type     low byte of      high byte of
         variable address variable address
```

type indicates the variable type:

2=integer
3=string
4=single precision
5=double precision

The returned value is essentially the same as:

```
CHR$(type)+MKI$(VARPTR(variable))
```

There is no direct equivalent in QL BASIC. The above information shuld help you understand what's going on if you come across VARPTR and VARPTR$ in a program, so that you can write suitable conversion code on a case-by-case basis.

Variable Names

In some early BASICs, only the first few characters of a variable name are significant, for example in TRS80 level 2 BASIC only the first two characters of a name are significant. This means that the variables SUM, SUB, SUPER, and SU are treated as though they are one and the same.

This restriction does not apply in QL SuperBASIC obviously, although poorly written old BASIC programs which use different lengths of the same variable name (e.g. SUPER in one part of the program, abbreviated to SU in another part) can cause havoc when ported to the QL, where the two variations of the same variable name will be treated as different variables. On the QL, LET SU=0 is not the same as LET SUPER=0. Unfortunately, there is no hard and fast rule on this, other than reading the manual for the BASIC in question if you have access to it (many BASIC manuals are available to download from the internet these days).

Some BASICs may have four distinct types of variable names:

(1) **Integer** - whole numbers from -32768 to + 32767. Variable names end with '%' as on the QL

(2) **Single Precision** floating point - usually to 6 significant figures. Names end with the ! symbol. In most cases just use ordinary QL floating point variables, but remove the ! type declaration symbol.

(3) **Double Precision** floating point - usually up to 16 significant figures. Variable names end with the hash # symbol, e.g. ZZ# is such a variable. In most cases you will be able to replace these with ordinary QL floating point values just be removing the # symbol from the variable name.

(4) **String** variables end with a '$' symbol like in QL SuperBASIC. Strings may be limited to just 255 characters in length, and in some cases only about 50 characters may be available by default for a given string.

In some early BASICs, only variable names starting with A or B may be used to hold strings (e.g. TRS80 BASIC level 1).

Variable names such as A or DE which appear to be floating point may in fact be overriden with implicit type definition commands such as DEFINT. See the entries for DEFINT, DEFSTR, DEFSNG and DEFDBL above.

Beware of versions of BASIC which have case sensitive variable and array names. In this case, 'A' and 'a' might be two separate and distinct variables - in other words, A and a may not the same variables! There is no easy way around this in SuperBASIC, where names are case insensitive, other than to use different variable and array names, or doing something like keeping the

upper case names as they are and adding a '_' to the lower case equivalents, e.g. LET A=0:LET _a=0

WAIT

This command suspends program execution while monitoring the status of a machine input port. The command syntax is WAIT port, n [,m]

port is the port number, in the range 0 to 65535

n, m are integer expressions in the range 0 to 255

The WAIT statement auses execution to be suspended until a specified machine port develops a specified bit pattern.

The data read at the port is XORed with the integer expression m and then ANDed with n. If the result is zero, BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If m is omitted, it is assumed to be zero.

There is no direct equivalent in QL BASIC and the different hardware configuration of various computers means that a conversionis unliekly to be meaningful, but if you really wanted to try with a memory mapped peripheral, you could probably write a piece of code like this:

```
1000 DEFine PROCedure WAIT (port,n,m)
1010   LOCal waiting,value
1020   REPeat waiting
1030     value = (PEEK(port) ^^ m) && n
1040     IF value <> 0 THEN EXIT waiting
1050   END REPeat waiting
1060 END DEFine WAIT
```

WHILE / WEND

loop structure whereby a series of statements in a loop are executed as long as the given expression is true. The condition is evaluated at the WHILE statement and if that condition is not true (i.e. it is FALSE, or 0) program execution continues at the statement following the WEND keyword.

```
6000 x=0
6010 WHILE x < 10
6020 x=x+1
6030 WEND
```

This is distantly related to REPeat/END REPeat statements in QL BASIC, although the REPeat and END REPeat commands do not in themselves perform the expression test - you need to write the test as a separate line within the loop and use an EXIT statement to exit the loop if the expression is false.

```
6000 x=0
6010 REPeat loop
6020 IF NOT(x<10) THEN EXIT loop
6030 x=x+1
6040 END REPeat loop
```

Note the inversion of the expression test result using the NOT operator in line 6020.

WIDTH

Sets the output line width in number of characters. Similar to the WIDTH command in QL BASIC, but QL BASIC needs a channel number.

WRITE

This command writes out a list of expressions, much like PRINT, except that WRITE inserts commas between the items as they are displayed and delimits strings with quotation marks. Also, positive numbers are not preceded by blanks. This example shows how WRITE displays numeric and string values.

```
10 A=80: B=90: C$="Hello"
20 WRITE A,B,C$
RUN
80,90,"HELLO"
OK
```

As this is mainly used as a variable value dump to help debug programs, you may not need to convert it to QL, although easily done by writing it like this:

```
PRINT A;",";B;",";'"';C$;'"'
```

WRITE #

WRITE #filenum,list_of_expressions writes data to a sequential file.

filenum is the number under which the file was opened for output, like a channel number in QL BASIC.

list_of_expressions
        is a list of string and/or numeric expressions, separated by commas or semicolons

The difference between WRITE # and PRINT # is that WRITE # inserts commas between the items as they are written and delimits strings with quotation marks. Therefore, it is not necessary for the user to put explicit delimiters in the list. Also, WRITE # does not put a blank in front of a positive number. A carriage return/linefeed sequence is inserted after

the last item in the list is written.

Example:
```
LET A$="CAMERA" : LET B$="93604-1"
WRITE #1,A$,B$
```

writes the following image to the file.
```
"CAMERA","93604-1"
```

A subsequent INPUT # statement, such as:

```
INPUT #1,A$,B$
```

would input "CAMERA" to A$ and "93604-1" to B$.

There is no direct equivalent in QL BASIC, so you would rewrite
```
WRITE #filenum,A$,B$ as
PRINT #filenum,'"';A$;'","';B$;'"'
```

XOR

The XOR operator tests for an Exclusive Or condition, whereby the result is true if either of the conditions tested is true but not the other. Here is the truth table for this operator.

X	Y	X XOR Y
True	True	False
True	False	True
False	True	True
False	False	False

The use of XOR in QL BASIC is the same.