BASICODE TRANSLATOR v2.0

FOR THE SINCLAIR QL

User's Manual

(preliminary version)

Original Dutch release: March 1988
English translation and update: December 2017

0. Introduction

This package allows you to read and write BASICODE programs on the Sinclair QL. BASICODE is a software standard developed by Dutch computer enthusiasts in the 1980's, allowing BASIC programs to be broadcast on the radio and run on several brands of home computers like the Acorn, Commodore, MSX and Sinclair ZX Spectrum. On these machines, programs were usually stored on cassette tapes so it was very easy to record them from the radio and load them into your computer.

From the early '80s until 1992, many BASICODE programs were written by computer enthusiasts, mainly in Holland but also in Germany and the UK. During this period, the Dutch broadcasters involved with BASICODE also distributed CDs and cassettes containing BASICODE programs which have been aired over the years.

You might wonder how I managed to load BASIC programs, recorded from the radio, into the QL since the machine never had a cassette interface. Back in 1987, I solved this problem by using the network port, adding some simple electronics, and putting this together with the translator software on an EPROM, which was necessary because the loading and saving routines needed exact timing which on a standard QL could only be achieved in ROM!

So fast forward to 2017. If you're lucky to still have a QL, you can re-build the cassette interface and load your BASICODE programs from tape, if you have them. But for many QL users this is not an option. First of all, the original QL hardware might not be available (you are using alternative hardware or an emulator), and chances are big that you have dumped all your old cassettes when you moved homes or did not even have had BASICODE programs at all!

Fortunately, there are still ways to get BASICODE programs into the QL, and it's even easier and faster than before. A GitHub repository has been created which contains hundreds of BASICODE programs which have been aired and published on cassettes and CDs. These programs are available in ASCII form and can be readily loaded into S*BASIC and run.

This new release of QL BASICODE comes 30 years after the original release on EPROM. A lot of things have changed since then; original QL hardware is becoming scarce and many QL enthusiasts who are still active or returning to the scene (including myself) are now using emulators such as QPC2, which run at speeds several hundred times that of the QL's 68008 processor and use SMSQ/E, which is a QDOS-compatible operating system offering graphic capabilities superior to the original QL hardware. The original code has been updated and made compatible with these systems as much as possible. The BASICODE-3C standard, which supports colour, has also been implemented, even though only a few application program support colour graphics. And, in the true spirit of BASICODE, the source code of the entire project has now been made available so if you are not satisfied with a particular feature, you can now build your own version of QL BASICODE!
All source code and documentation can be found at https://github.com/SinclairQL/Basicode (which is supposed to be stable). A more recent version, which might be unstable, is available at https://github.com/janbredenbeek/QL-Basicode.

Jan Bredenbeek, September 2017.

e-mail: jan@bredenbeek.net

1. QuickStart Guide

WHAT IS BASICODE?

In short: BASICODE was a kind of 'BASIC Esperanto' for home computers, developed by Dutch computer enthusiasts in the early 1980's. It enabled BASIC programs to be run on several brands of home computers by eliminating machine-specific BASIC commands and replacing them by subroutines which perform a specific task. These machine-specific subroutines are placed at the beginning of the program and are merged with the application program when loaded into the computer, forming a complete BASIC program.

In the 1980's, BASIC programs were stored on cassette using audible signals. BASICODE also provided for a standard modulation format so that programs written in BASICODE could be transmitted over the radio. From 1981 to 1992, BASICODE programs were aired weekly by Dutch broadcasters NOS and TROS. Outside the Netherlands, the BBC and German broadcasters WDR and Radio DDR have also contributed to the distribution of BASICODE.

Rob Hagemans has created an excellent repository of BASICODE programs from radio recordings and 'Best of BASICODE' cassettes published over the years. These programs are now available in ASCII form so you don't need to bother about loading programs from cassette (although this QL implementation still supports it!). It's available at https://github.com/janbredenbeek/basicode. This repository also contains a detailed description of the BASIC commands and standard subroutines used in BASICODE.

IMPLEMENTATION ON THE SINCLAIR QL

BASICODE on the QL is available in two flavours:

1. a RAM-version which can be loaded as resident extension using LRESPR;
2. a ROM-version which can be burnt on EPROM or loaded as extension ROM.

The ROM-version has support for a cassette interface, which can be built by connecting the QL's network port with some simple electronics. It has extra commands to facilitate this from SuperBASIC as well as a special CAS device for loading and storing files. Of course, you will need native QL hardware for this to work!

If you're using an emulator, or only want to run BASICODE programs stored in ASCII-form, the RAM-version will be sufficient. It lacks the extra commands for cassette support and the CAS device, but is compatible with most QL emulators available today, including QPC2 which runs SMSQ/E and SBASIC.

NOTE: SMSQ/E allows you to have several instances of SBASIC active at the same time, each running their own BASIC program. You should however load the QL BASICODE extensions only once, in the main instance of SBASIC. Do not try to load QL BASICODE in a ‘cloned’ SBASIC job since the BASICODE extensions will disappear as soon as you do a NEW or LOAD command!

GETTING STARTED

To get started, you need the following:

- A QDOS or SMSQ/E environment such as a native QL or emulator

- The BASICODE extension binary BCRAM_BIN or BCROM_BIN - read the above section on how to choose the right one.
  NOTE: If you want to run BASICODE-programs stored on cassette, you MUST use the ROM version and burn it on EPROM. You can't just load it into RAM and activate it using CALL since the cassette routines need exact timing, which can't be achieved in RAM due to the QL's video hardware which constantly accesses RAM, causing it to periodically halt the CPU.
  For the same reason, you can't use the cassette interface with extensions like the Gold Card which run significantly faster than native QLs!
- and of course BASICODE programs! You can get them from Rob Hagemans's repository, (https://github.com/janbredenbeek/basicode). A sample program written by yours truly (skymap.bc3) has also been included in the QL BASICODE distribution repository.

Start by loading the BCRAM_BIN file as a system extension, i.e. LRESPR BCRAM_BIN. When using the ROM version, you should see the BASICODE ROM message printed on startup.

You should now LOAD the standard subroutines by entering the following command:

BCBOOT

This clears any existing BASIC program and loads the BASICODE standard subroutines, needed for any BASICODE program to run. You need to do this before loading any BASICODE program (see note on the BCLOAD command below).

LOADING BASICODE PROGRAMS FROM TAPE:

(You can of course skip this if you only load BASICODE programs as ASCII file)

First of all, you need to calibrate the cassette interface. Enter the command

REPeat loop: AT 0,0: PRINT PEEK(98336)

This prints a number at the top left corner of the window. Now adjust the trim potentiometer on the cassette interface so that it's just around the point where the number flips from an even to an odd number.

A BASICODE program can be loaded from tape by entering the command:

BCLOAD_T

and then start the tape. After the program has been read in, it can be saved as a ASCII file. Then, continue to the next step...

LOADING BASICODE PROGRAMS FROM A FILE:

There are two commands for this: BCLOAD and BCMERGE, both followed by a file name. They both load the file, which should be in plain ASCII, into S*BASIC doing the necessary conversions as they read it in. The difference between the two commands is that BCLOAD first clears out any BASIC program above line 1000 and BCMERGE does not, so BCMERGE may be used to MERGE two BASICODE programs into one (if the line numbers clash, the newer lines will overwrite the older).

NOTE: After testing BCLOAD on SMSQ/E, I discovered that the vectored routine that should delete all BASIC lines above 1000 didn't work on SBASIC as it did on the original QL ROMs and Minerva. Unfortunately, I haven't yet found a way to duplicate this 'DLINE 1000 TO' functionality without

crashing SBASIC. So for the moment, a BCLOAD command executed on SMSQ/E will return an error message when there are still lines above 1000 present in SBASIC. The easiest way to work around this is to execute a **BCBOOT** command, which will clear out the whole SBASIC program and reload the standard BASICODE subroutines. Alternatively, if you want to preserve the standard subroutines, you can enter **'DLINE 1000 TO'**.

The program can now be run by simply typing RUN. If you notice any errors or other anomalies, please refer to Section 2 and 3 for more information. But it's worth noting that the BASIC section from line 20 contains various commands to initialise the BASICODE environment and some of them may need adjustments. This is especially true for the screen handling commands, as there are now many S*BASIC environments around with graphics capabilities which differ greatly from the original QL (see below)

THE TEXT AND GRAPHICS DISPLAYS OF BASICODE

Developed in the early 1980's, BASICODE was originally designed for text displays of 40 columns wide with 24 or 25 screen lines. With BASICODE-3, graphics capabilities were introduced as well as a simple way to determine the size of the text display so that programs could adapt their output. In order to simplify graphics programming for a wide number of platforms, the following conventions were adopted:

- The display can be either in text mode (initiated by a GOSUB 100) or graphics mode (initiated by GOSUB 600). In graphics mode, the commands PRINT and INPUT are NOT allowed. However there is a GOSUB 650 routine to print text on the graphics screen;
- The graphics screen is assumed to have a 4:3 aspect ratio with square pixels. The x and y coordinates (HO and VE in subroutines 620, 630 and 650) are floating point variables ranging from 0,0 (top left corner) to 1,1 (just past the bottom right corner).
- Subroutine 20 should set HO and VE according to the text mode resolution (i.e. number of columns and lines) and HG and VG to the graphics resolution (this means that a GOSUB 620 with HO=1-1/HG and VE=1-1/VG plots a point in the bottom right corner of the graphics screen).
- In 1991, BASICODE-3C was introduced with colour capabilities, supporting 8 colours in text and graphics mode. Since this was very late in BASICODE's lifecycle, there are not many programs which make use of it.

While the graphics capabilities of even the bare QL are more than adequate to accommodate the requirements of BASICODE, the fact that there are so many graphics resolutions available now makes things a little tricky. First of all, using the 'lowest common denominator' of native QL screen resolution (256x256 with 8 colours) will get you a relatively small window with huge text characters on a modern PC running SMSQ/E. While (almost) all BASICODE programs should run fine using this minimal resolution, there are also many programs which will benefit from the higher resolutions being offered today. For this reason, I have chosen not to set the size of the default output window (#1) in the initialisation routine at line 20 as earlier versions did. If you need a particular window or character size (e.g. for running old programs which assume a 40x24 character screen), feel free to put your own WINDOW statement there.

The variables HO, VE, HG and VG are dynamically calculated by functions called from the initialisation code according to the window size.

NOTE: For most programs to function correctly, a window size of at least 480 x 240 pixels is recommended. On native QL hardware, this yields a 40x24 text screen (at 8 colours). Since most

BASICODE programs don't use colour, you can also safely use MODE 4 which gives you an 80x24 text screen.

A second problem is caused by the fact that the native QL screen has non-square pixels. In MODE 8, using 256x256 pixels, a circle drawn on a BASICODE graphic screen would be displayed as a flattened ellipse since BASICODE assumes an aspect ratio of 4:3 (as was common on TVs in the '80s).

…to be finished…

2. Compatibility between BASICODE and S*BASIC

BASICODE was originally developed as a 'lowest common denominator' between various computer platforms which existed around 1980. Most of these platforms used Microsoft Basic as programming environment so the set of programming rules in BASICODE has been largely based on versions of Microsoft BASIC as it existed in the late 1970's. This means that structured programming techniques such as loops (except FOR-NEXT), procedures and functions are not available and we have to stick to dirty old GOTOs and GOSUBs as means to control program flow. Variable names are restricted to just two characters and the length of a BASIC line must not exceed 60 characters. On the other hand, you don't have to waste valuable space by typing spaces around keywords as is required by more modern BASICs – a line like 'IF A=1 THEN GO TO 1200' can be typed in BASICODE as 'IFA=1THEN1200'!
Thus, there are some conversions needed in order to be able to run BASICODE on S*BASIC. The translator program will take care of most of them automatically, but there are some odds and ends left that sometimes require manual intervention. These will now be discussed below.

2.1. Non-S*BASIC functions

Some functions in BASICODE either do not exist in S*BASIC, or exist under another name. Examples are LEFT$, MID$, RIGHT$, VAL and SGN (which have no equivalent S*BASIC function) and ASC, LOG and ATN (for which the S*BASIC equivalents are CODE, LN and ATAN respectively).
QL BASICODE will take care of these functions by defining them as machinecode extensions. Of course, this requires that QL BASICODE must always be loaded first before attempting to run a BASICODE program.

2.2. String concatenation

In BASICODE, string concatenation is performed using the '+' operator (e.g. when a$="john" and b$="doe" then a$+b$ yields "johndoe"). In S*BASIC, the '&' operator must be used since '+' will try to add the numerical values of a$ and b$ (just try it for fun). QL BASICODE will automatically convert this.

2.3. Array names

Like ordinary variables, array names in BASICODE are restricted to a maximum of two characters. However, they do have a separate namespace from ordinary variables. Thus, a variable A and an array A(10) can co-exist in BASICODE without confusion. Not so in S*BASIC: When DIM-ing an array A(), any use of A will refer to the array A() – a feature that by itself can be very powerful but unfortunately will confuse any BASICODE program that tries to use A and A() at the same time.
To prevent this confusion, QL BASICODE adds an extra underscore character to the name of **all** arrays. Hence, A(10) will be converted to A_(10) and AB$(X) will be converted to AB_$(X).

2.4. FOR-NEXT loops

These are a bit more complicated. In BASICODE, a FOR-NEXT loop comprises **all** statements between a FOR statement and the corresponding NEXT statement. There is no such thing as an 'inline' FOR loop (which comprises only the statements after a FOR on the same line) as in S*BASIC.
To avoid confusion, QL BASICODE splits lines containing a FOR statement if there are any statements on the same line after the FOR. Thus, a line like

```
2000 FOR I=1 TO 10: GOSUB 5000:NEXT I
```

will translate to:

```
2000 FOR I=1 TO 10
2001 GOSUB 5000: NEXT I
```

However, this will go wrong when there is an IF statement before the FOR. For instance:

```
2000 IF A=1 THEN FOR I=1 TO 10: GOSUB 5000: NEXT I
```

gets translated to:

```
2000 IF A=1 THEN FOR I=1 TO 10
2001 GOSUB 5000: NEXT I
```

This works correctly when A equals 1, but in other cases the FOR statement will not be executed. On encountering the NEXT in line 2001, S*BASIC won't find the corresponding FOR and stops with an error message.

There are two ways to correct this. One is to undo the split and place the GOSUB 5000: NEXT I back on line 2000. However, you may then run into a bug which is present in the original QL ROMs: if a FOR loop contains a GOSUB, SuperBASIC will treat the GOSUB as an end-of-line and execute it only once when I=10!
To avoid this issue, if you don't have a Minerva ROM or SMSQ/E (which don't suffer from this bug), you can change the code to something like this:

```
2000 IF A<>1 THEN GOTO 2010
2001 FOR I=1 TO 10
2002 GOSUB 5000: NEXT I
2010 …
```

2.5. String arrays

In BASICODE, string arrays have a dynamic length just like ordinary string variables. S*BASIC is a bit pickier about this: when DIMing a string array, you must specify the maximum length of a string within the array as an *extra* dimension. For instance, a statement:

```
DIM A$(10)
```

will give you an array of 11 (A$(0) to A$(10)) strings with dynamic length in BASICODE.
However, in S*BASIC, to achieve more or less the same, you have to do:

```
DIM A$(10, maxlength)
```

(where *maxlength* is the length of the longest string you expect.)

You can of course determine *maxlength* by trial and error. However, a fortunate circumstance is the fact that the length of a string in BASICODE is limited to only 255 characters, and even on a bare QL you will have plenty of memory available. Thus, you can safely set *maxlength* to 255, which is exactly what QL BASICODE does when translating BASICODE to S*BASIC! So, in most cases you don't have to change any DIM statements.
In the unlikely event that you still run out of memory when DIMing a string array, you can always experiment with a lower length value.

2.6. DEF FN

This command defines a user-defined function, like the S*BASIC DEFine FuNction command. However, the syntax of DEFFN in BASICODE is much different (and more restrictive) than in S*BASIC. It takes the form DEFFN *name*(*parameter*)=*expression*, where *name* follows the same rules for numeric variables

(thus string functions are not allowed!), *parameter* is a numeric or string variable (which is local to the function) and *expression* a numeric expression.
The function is defined by *executing* the DEFFN statement (just putting a DEFFN somewhere in the program is not enough), and may be called within an expression by 'FN *name*(*parameter*)'. This example defines a function SH(X) which returns the sinus hyperbolicus (sinh) of X:

DEFFN SH(X)=(EXP(X)-EXP(-X))/2

Then, the command PRINT FN SH(1) will print the value of sinh(1).

When translating this to S*BASIC, two problems occur: the syntax of DEFFN used in BASICODE will be rejected as invalid, and functions cannot have the same name as an array or ordinary variable.
QL BASICODE will *not* automatically convert the BASICODE syntax to S*BASIC. Hence, a DEFFN will cause a syntax error, which must be corrected by manually editing the offending line. However, to avoid name conflicts with variables, QL BASICODE will *both* prepend and append an underscore to the function's name. In expressions which call the function, QL BASICODE will replace the 'FN' keyword with an underscore so the function will be called with correct syntax. Thus, it is *not* necessary to manually edit function calls!

An example using the above SH function:

Initial DEFFN statement with invalid S*BASIC syntax:

1500 MISTake DEFine FuNction _SH_(X)=(EXP(X)-EXP(-X))/2

The same line after manual correction:

1500 DEFine FuNction _SH_(X):RETurn (EXP(X)-EXP(-X))/2:END DEFine

Then, the command:

PRINT "The value of sinh(1) is: "; FN SH(1)

will automatically be translated to:

PRINT "The value of sinh(1) is: "; _SH_(1)

NOTE: The DEFFN command was introduced in the second edition of the BASICODE-3 book (1988) and appears to be rarely used in programs. In fact, I could only find a single instance of it in the BASICODE archive!

2.7. Other conversions

The BCLOAD command will also convert the following syntax:

- 'IF x THEN line number' to 'IF x THEN GOTO line number'
- 'PRINT TAB(X)' to 'PRINT TO(X)'.

3. Error messages during runtime

Now that you have loaded your program using BCLOAD, you can RUN it. Most programs will run fine under S*BASIC, however in some cases you may encounter errors. The text below will list some, but not all, possible causes of a specific error message.

"Not found"
Usually caused by a NEXT without corresponding FOR statement. See 2.4 for more information.

"Bad name" or **"Error in expression"**
Usually caused by using a variable which has not been assigned a value yet. Try PRINTing all variables which occur in the line in question. If you get an asterisk printed, you have found the culprit. Then, add a line at the start of the program (1015 is usually best) which assigns a zero (or empty string) to this variable.
If you encounter an array, try putting a DIM statement near the start of the program with size 10 (e.g. DIM X(10) or X$(10,255).

NOTE: on SMSQ/E and SBASIC, variables will have a default value of zero or null string. Hence, using an unset variable in an expression won't cause an error. However, SBASIC also has many extra Toolkit II commands which have two-letter names (such as EX and EW) which may clash with variable names used in BASICODE programs. There is nothing short of manually changing these names that can be done to cure this problem.
On native QL environments where Toolkit II causes such clashes, try avoiding the TK2_EXT command in your BOOT file.
Finally, this error will occur when you try to RUN a BASICODE program without loading the extensions first.

"Bad line" or "incorrect syntax"
Usually caused by a line with incorrect syntax, possibly caused by transmission errors but also by sloppy programming which other BASICs would let you get away with. A common example is 'PRINT "A has the value:"A' (missing ';' before A). This can be easily corrected, but don't forget to remove the MISTake keyword at the beginning of the line or you will again get this message.
Note that all uses of DEF FN will give this error and have to be manually adapted (see 2.6 above).

Program suddenly stops without error message
Probably caused by a FOR loop where the limit value has already been reached at the beginning. Example: FOR I=1 TO N where N is zero. In such a case, most other BASICs will execute the FOR loop once (which is strictly speaking wrong!), but S*BASIC will try to find the next matching ENDFOR statement and continue from there. Since there is no ENDFOR in BASICODE, S*BASIC will run off the end of the program.
Errors of this type are not easy to find and you probably have to find the offending FOR statement by examining the program flow and output of the program. You can then put an IF statement just before the FOR which skips the FOR loop when it is going to be executed zero times, or replace the NEXT at the end by END FOR. Note that QL BASICODE does not automatically replace NEXT by END FOR since NEXT may occur more times within the FOR loop, unlike END FOR.

BREAK in GOSUB 210
If you press BREAK in GOSUB 210 (which waits for a key to be pressed), then the cursor in window #1 will remain active, preventing command input in window #0. To cure this, you can press CTRL-C to switch to window #0 and then enter the command CURS_OFF to disable the cursor in window #1.

4. Command Reference

This section describes the extension commands added by QL BASICODE.

4.1. Procedures

BUFFER *n*
Only implemented in the ROM version. Sets the size of the tape buffer to be used for BCLOAD_T and BCSAVE to *n* Kbytes.

CLOAD *filename*
Only implemented in the ROM version. Loads a file from cassette and saves it as a QL file. Details to be described yet.

CSAVE *filename*
Only implemented in the ROM version. Saves a QL file onto cassette. Details to be described yet.

BCLOAD_T
Only implemented in the ROM version. Loads a BASICODE program from cassette and SAVEs it as an ASCII file.

BCLOAD *filename1[,filename2]*
When only *filename1* is given, loads a BASICODE program in ASCII format into S*BASIC. When two filenames are given, translates the BASICODE program in *filename1* to S*BASIC and writes it to *filename2.*
When loading into S*BASIC, all existing program lines above 1000 are deleted first. On SMSQ/E systems this is currently not possible and an error message is given if there are lines above 1000 in the BASIC program. You should delete these lines manually before issuing this command.

BCMERGE *filename1[,filename2]*
Like **BCLOAD** above, but does not delete existing program lines above 1000 before reading in the BASICODE program.

BCSAVE *[#channel;] line number*
Only implemented in the ROM version. Converts an existing S*BASIC program to BASICODE from *line number* onwards and saves it to tape in BASICODE format. Error messages will be sent to the specified channel or #1 by default. Details to be described yet.

BCBOOT
Loads the standard BASICODE subroutines into S*BASIC. Equivalent to LOAD BCBOOT.

CURS_ON *[#channel]*
Turns the cursor on on the specified window, or #1 by default. Used by GOSUB 210

CURS_OFF *[#channel]*
Turns the cursor off on the specified window, or #1 by default. Used by GOSUB 210.

BREAK_ON
Enables the BREAK key (it is enabled by default). Used by GOSUB 280.

BREAK_OFF
Disables the BREAK key. **Note:** on Minerva and SMSQ/E systems, which support multiple instances of S*BASIC running simultaneously, this command currently affects only the first instance (job 0).

BCSOUND *duration,pitch*
Sounds a tone through the QL speaker for *duration* frames (1 frame = 20 ms), where *pitch* is defined by the value of the SP parameter in subroutine 400 of the BASICODE-3 protocol. A pitch of 60 corresponds with the middle C, and an increase or decrease of 12 means a pitch exactly one octave higher or lower respectively. In practice, *pitch* can have values 29 to 88 since these are the limits imposed by the QL's sound generator. Also, since the internal pitch steps of this generator do not exactly correspond with the pitch steps used by BASICODE, the tones generated may sound slightly out of tune. This command is used by GOSUB 250 and 400.

GETLINE *#channel,string*
Reads a line of text from *channel* terminated by EOL and assigns it to *string*. Any I/O errors are trapped and may be read by using the IOSTATUS function. Used by GOSUB 540.

PUTLINE *#channel,string*
Sends text in *string* to *channel*, any I/O errors may be read by using the IOSTATUS function. Used by GOSUB 560.

PX_ASPRT *ratio*
On SMSQ/E systems version 3.00 and above, changes the pixel aspect ratio for graphics routines. *Ratio* is the pixel height/width ratio for the screen resolution used. On a standard QL system, which has a 512x256 resolution and a 4:3 aspect ratio, this value is 1.355. On a SMSQ/E system running under QPC2, you may want to set this to 1 since most PC resolutions have square pixels. This allows the graphics routines to produce graphics with correct aspect ratio. Note that BASICODE-3 assumes a 4:3 aspect ratio for the graphics screen and you may have to adjust the initialising commands from line 20 onwards to suit your own needs.
This command should only be used on SMSQ/E version 3.00 and above, as this feature is not implemented at system level on earlier versions (including QDOS and Minerva).

4.2. Functions

WWIDTH *[#channel]*
WHEIGHT *[#channel]*
Returns the defined width and height of the specified window (or #1 by default) in pixels. Used by the initialisation routine at line 20.

HSIZE *[#channel]*
VSIZE *[#channel]*
Returns the number of text columns (HSIZE) and rows (VSIZE) of the specified window (or #1 by default). Used by the initialisation routine at line 20.

HPOS *[#channel]*
VPOS *[#channel]*
Returns the current cursor position in characters of the specified window (or #1 by default). Used by GOSUB 120.

ASC, ATN, LOG, SGN, SQR, VAL, LEFT$, MID$, RIGHT$
These functions are implemented as per the BASICODE definition.

USING$ *(length,decimals,number)*
Returns a string containing the decimal representation of *number* of *length* digits, using *decimals* number of decimals. If *number* wouldn't fit into the specified *length,* a string containing asterisks ('***') is returned. If *length* is greater than the number of digits required (including any decimal point), the string is padded to the left with spaces. Used by GOSUB 310.

BSC_CODE *(string$)*
Returns, like ASC, the ASCII code of the first character of *string$* but takes into account special codes for cursor and function keys (see below). Used by GOSUB 200,210,450.

KEY	CODE
ENTER	13
LEFT	28
RIGHT	29
DOWN	30
UP	31
DELETE	127
F1 TO F5	-1 to -5
F6 TO F10 **(F1 TO F5 SHIFTED)**	-6 to -10

SHIFT$ *(string$)*
Returns *string$* with all lowercase letters converted to uppercase. Used by GOSUB 330.

SCREEN$ *([#channel;]x,y)*
Returns the character in the specified window (default #1) at cursor position x,y (in the text coordinate system). **Note**: This currently only works on QL screen modes (512x256x4 or 256x256x8), and the current STRIP and INK colours must be the same when the character was printed in order for it to be recognised. Used by GOSUB 220.

FOPEN_IN *(#channel,filename)*
FOPEN_NEW *(#channel,filename)*
Opens file *filename* (which may be a string or name) to *channel* for reading (FOPEN_IN) or writing (FOPEN_NEW). If the open succeeds the functions return 0, if it fails for any reason it returns -1. Used by GOSUB 500.

IOSTATUS
Returns the error status of the last GETLINE or PUTLINE command (0 means OK, <0 means error). Used by GOSUB 540 and 560.