Enhanced 6502 BASIC reference manual

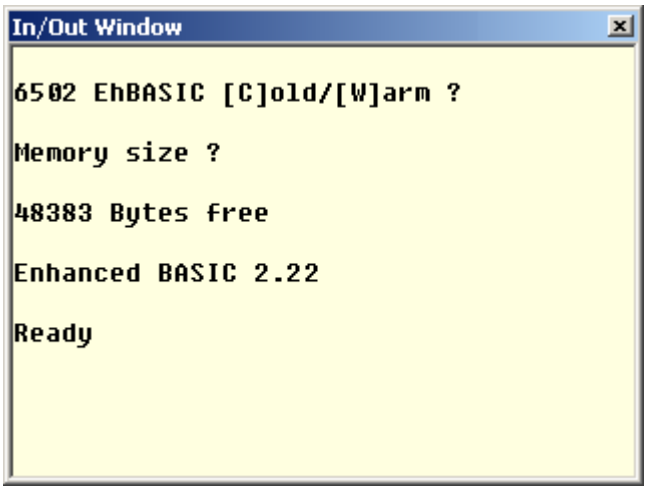
Preface

This manual has been compiled in October 2013 from a snapshot of Lee Davison's website http://mycorner.no-ip.org/6502/ehbasic/index.html after it had been off air for quite a while. In an effort to keep reference material available for and to preserve usability of EhBASIC, I have undertaken the task to collect all available information in a PDF format manual. Please contact me, if you find errors in the manual: K@2m5.de

Content

Enhanced 6502 BASIC By Lee Davison

Introduction



Enhanced BASIC is a BASIC interpreter for the 6502 and compatible microprocessors. It is constructed to be quick and powerful and easily ported to most 6502 systems. It requires few resources to run and includes instructions to facilitate easy low level handling of hardware devices. It also retains most of the powerful high level instructions from similar BASICs.

EhBASIC represents hundreds of hours work over nearly three years, lots of frustration, lots of joy and the occasional twinge from RSI induced tendonitis.

EhBASIC is free but not copyright free. For non commercial use there is only one restriction, any derivative work should include, in any binary image distributed, the string "Derived from EhBASIC" and in any distribution that includes human readable files a file that includes the above string in a human readable form e.g. not as a comment in an HTML file.

For commercial use please e-mail Lee for conditions.

Enhanced BASIC requirements

Minimum requirements

- 6502 processor.
- 10k ROM or RAM for the interpreter code.
- 1k of RAM from $0000.
- RS232 I/O.

Preferred requirements

- 6502 or better processor (65c02, CCU3000, M38xx).
- 10k ROM or RAM for the interpreter code.
- RAM from $0000 to $BFFF (more with changes).
- Any character based I/O (e.g. RS232, LCD/keyboard etc).

Enhanced BASIC on your system

Hardware.

EhBASIC can be made to work on nearly any 6502 system, it requires very little. The system it was developed on is a combination of my SBC and 6551 projects.

Memory.

EhBASIC makes extensive use of page zero and some use of page 2. Some areas may be re-used as long as care is taken. Program and variable space is from $0300 up to whatever is available, the more the better. The interpreter can be ROM or RAM based and can be assembled to reside almost anywhere in memory, only minor changes need to be made.

Software.

For minimal functionality the interpreter needs only two external routines, a character get routine and a character send routine.

For full functionality two other external routines, load and save, along with two interrupt service routines are needed.

Minimal set-up is required, most of the set-up is performed by the interpreter cold start routine.

How to.

The interpreter calls the system routines via RAM based vectors and, as long as the requirements for each routine are met, these can be changed on the fly if needs be.

All the routines exit via an RTS.

The routines are ..

Input	This is a non halting scan of the input device. If a character is ready it should be placed in A and the carry flag set, if there is no character then A, and the carry flag, should be cleared.
Output	The character to be sent is in A and should not be changed by the routine. Also on return, the N and Z flags should reflect the character in A.
Load	This is entirely system dependant.
Save	This is entirely system dependant.

Also if you wish to use the ON {IRQ|NMI} commands ..

Irq	If no other valid interrupt has happened then this routine should, after checking that the interrupt is set-up, set the IRQ interrupt happened flag.
Nmi	If no other valid interrupt has happened then this routine should, after checking that the interrupt is set-up, set the NMI interrupt happened flag.

Example code.

Example code for all the above is provided if the file **min_mon.asm** that is included in the main source code archive.

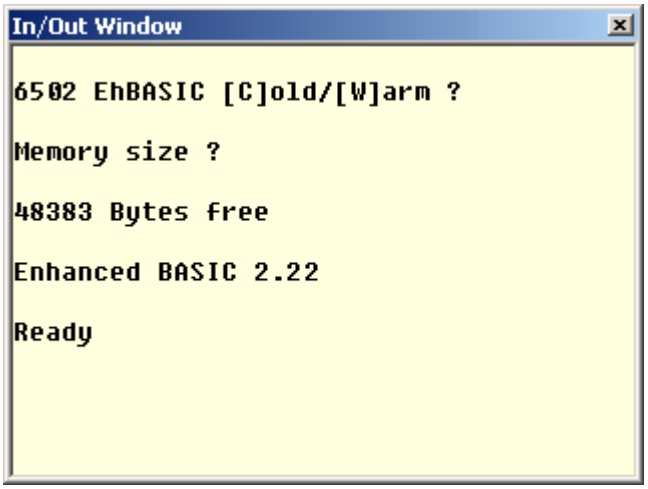
Starting Enhanced BASIC

Starting EhBASIC

Starting EhBASIC will mostly depend on how you set up your system to start it. The following assumes you are trying to run EhBASIC using the example monitor and Michal Kowalski's 6502 simulator, though this should not differ too much from the startup of EhBASIC on a real system.

- Unpack the .zip source to a directory and run the 6502 simulator.
- Open min_mon.asm from the directory where you unzipped it.
- Select assemble **[F7]**.
- Run the debugger **[F6]**.
- Make sure the I/O window is open.
- Press **[CTRL]**, **[SHIFT]** and **R** to reset the simulated processor.

You should then be presented with the **[C]old/[W]arm** prompt as seen here. As the simulator has just been started you should now press **C** for a cold start.



This should present you with the **Memory size ?** prompt. Now type either carriage return, in which case EhBASIC calculates available memory space automatically, or enter the total size of the memory in either decimal, hex or binary followed by a carriage return.

E.g. to set the physical memory size to 8k bytes.

In decimal ..

```
Memory size ? 8192
```

.. or in hex ..

```
Memory size ? $2000
```

.. or in binary.

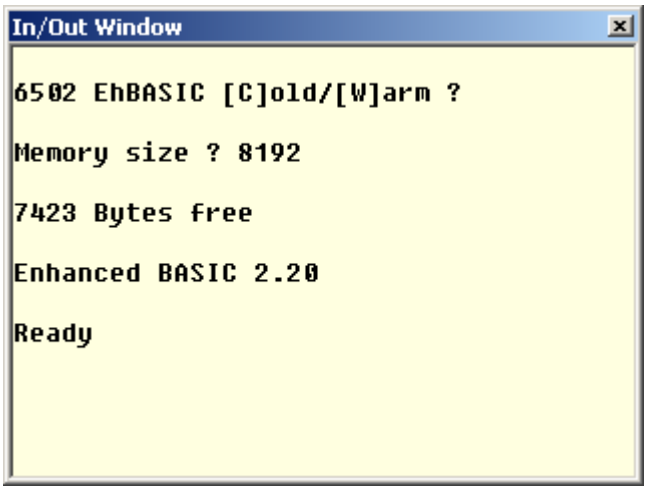
```
Memory size ? %10000000000000
```

EhBASICs program memory is then allocated from Ram_base, which is usually $0300, up to the limit specified. Any remaining RAM, or any RAM not continuous from EhBASICs memory, may be used to contain user subroutines or data.

If you did not enter a number greater than the minimum required to run EhBASIC, or there is not the minimum memory present, then EhBASIC will return to the **Memory size ?** prompt.

Do not type a number larger than the physical memory present. EhBASIC assumes you know what you are doing and does not check the specified memory size. Trying to use non existent RAM will, at best, corrupt your string variables. *This check can easily be implemented, the code is already in place but is commented out. See the source for more details.*

There is no Terminal width ? prompt as with some BASICs, the default is for no terminal width limit to be set. However if you wish to set a terminal width, and a TAB step size, there is a **WIDTH** command available, *see WIDTH in the EhBASIC language reference.*



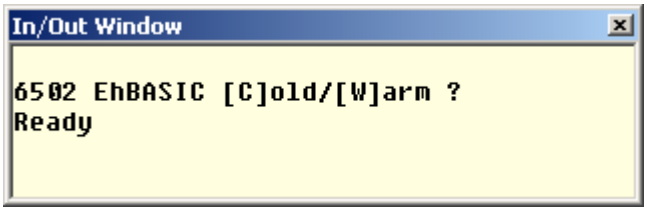
If the memory sizing was successful then EhBASIC will respond with the total number of bytes available for both programmes and variables and then the Ready prompt.

The display should look something like the image on the right.

You are now ready to start using EhBASIC.

Restarting EhBASIC

To restart EhBASIC After a reset, assuming you have at some time performed a cold start, if you have set up a Cold/Warm start request just press **W**.



If all is well, and sometimes if not, EhBASIC will respond with the **Ready** prompt like that shown here.

After a warn start, if the reset was not caused by a program running amok, the program and all the variables used, will be unchanged. You will not though be able to use **CONT** to continue program execution.

So you are now ready to program in EhBASIC, check the language reference for details.

Enhanced BASIC language reference

Numbers

Numbers may range from zero to plus or minus 1.70141173x10^38 and will have an accuracy of just under 1 part in 1.68 x 10^7.

Numbers can be preceded by a sign, + or -, and are written as a string of numeric digits with or without a decimal point and can also have a positive or negative exponent as a power of 10 multiplier e.g.

-142 96.3 0.25 -136.42E-3 -1.3E7 1

.. are all valid numbers.

Integer numbers, i.e. with no decimal fraction or exponent, can also be in either hexadecimal or binary. Hexadecimal numbers should be preceded by **$** and binary numbers preceded by **%**, e.g.

%101010 -$FFE0 $A0127BD -%10011001 %00001010 $0A

.. again are all valid numbers.

Strings

Strings are any string of printable characters enclosed in a pair of quotation marks. Non printing characters may be converted to single character strings using the CHR$() functions.

"Hello world" "-136.42E-3" "+----+----+" "[Y/n]" "Y"

Are all valid strings.

Variables

Variables of both numeric and string type are available. String variables are distinguished by the **$** suffix. As well as simple variables arrays are also available and these may be either numeric or string and are distinguished by their bracketed indices after the variable name.

Variable names may be any length but only the first two name characters are significant so BL and BLANK will refer to the same variable. The first character must be one of "A" to "Z" or "a" to "z", following characters may also include numbers. E.g.

A A$ NAME$ x2LIM y colour s1 s2

Variable names are case sensitive so AB, Ab, aB and ab are all separate variables.

Variable names may not contain BASIC keywords. Keywords are only valid in upper case so 'PRINTER' is not allowed (it would be interpreted as PRINT ER) but 'printer' is.

Note that spaces in variable names are ignored so 'print e r', 'print er' and 'pri nter' will all be interpreted the same way.

BASIC Keywords

Here is a list of BASIC keywords. They are only valid when entered in upper case as shown and spaces may not be included in them. So GOTO is a valid BASIC keyword but GO TO is not. Click ▲ to return to keyword index.

ABS	AND	ASC	ATN	BIN$	BITCLR	BITSET
BITTST	CALL	CHR$	CLEAR	CONT	COS	DATA
DEC	DEEK	DEF	DIM	DO	DOKE	ELSE
END	EOR	EXP	FN	FOR	FRE	GET
GOSUB	GOTO	HEX$	IF	INC	INPUT	INT
IRQ	LCASE$	LEFT$	LEN	LET	LIST	LOAD
LOG	LOOP	MAX	MID$	MIN	NEW	NEXT
NMI	NOT	NULL	OFF	ON	OR	PEEK
PI	POKE	POS	PRINT	READ	REM	RESTORE
RETIRQ	RETNMI	RETURN	RIGHT$	RND	RUN	SADD
SAVE	SGN	SIN	SPC(	SQR	STEP	STOP
STR$	SWAP	TAB(	TAN	THEN	TO	TWOPI
UCASE$	UNTIL	USR	VAL	VARPTR	WAIT	WHILE
WIDTH	+	-	*	/	^	<<
>>	>	=	<			

- Anything in upper case is part of the command/function structure and must be present
- Anything in lower case enclosed in < > is to be supplied by the user
- Anything enclosed in [ ] is optional
- Anything enclosed in { } and separated by | characters are multi choice options
- Any items followed by an ellipsis, ... , may be repeated any number of times
- Any punctuation and symbols, except those above, are part of the structure and must be included

var	is a valid variable name
var$	is a valid string variable name
var()	is a valid array name
var$()	is a valid string array name

expression	is any expression returning a result
expression$	is any expression returning a string result

addr	is an integer in the range +/- 16777215 that will be wrapped to the range 0 to 65535
b	is a byte value 0 to 255
n	is an integer in the range 0 to 63999
w	is an integer in the range -32768 to 32767
i	is a positive integer value

r	is real number
+r	is a positive value real number (0 is considered positive)
$	is a string literal

BASIC Commands

BITCLR <addr>,<b>

Clears bit b of address addr. Valid bit numbers are 0, the least significant bit, to 7, the most significant bit. Values outside this range will cause a function call error. ▲

BITSET <addr>,<b>

Sets bit b of address addr. Valid bit numbers are 0, the least significant bit, to 7, the most significant bit. Values outside this range will cause a function call error. ▲

CALL <addr>

CALLs a user subroutine at address addr. No values are passed or returned and so this is much faster than using USR(). See extending CALL for details. ▲

CLEAR

Erases all variables and functions and resets FOR .. NEXT, GOSUB .. RETURN and DO ..LOOP states. ▲

CONT

Continues program execution after CTRL-C has been typed, a STOP has been encountered during program execution or a null input was given to an INPUT request. ▲

DATA [{r|$}[,{r|$}]...]

Defines a constant or series of constants. Real constants are held as strings in program memory and can be read as numeric values or string values. String constants may contain spaces but if they need to contain commas then they must be enclosed in quotes. ▲

DEC <var>[,var]...

Decrement variables. The variables listed will have their values decremented by one. Trying to decrement a string variable will give a type mismatch error. DEC A is much faster than doing A=A-1 and DEC A,A is slightly faster than doing A=A-2. ▲

DEF FN <name>(<var>) = <statement>

Defines <statement> as function <name>. <name> can be any valid numeric variable name of one or more characters. <var> must be a simple variable and is used to pass a numeric argument into the function.

Note that the value of <var> will be unchanged if it is used in the function so <var> should be considered to be a local variable name. ▲

DIM <var[$](i1[,i2[,in]...])>[,var[$](i1[,i2[,in]...])]...

Dimension arrays. Creates arrays of either string or numeric variables. The arrays can have one or more dimensions. The lower limit of any dimension is always zero and the upper limit is i. If you do not explicitly dimension an array then it's number of dimensions will be set when you first access it and the upper bound will be set to 10 for each dimension. ▲

DO

Marks the beginning of a DO .. LOOP loop (See LOOP). No parameters. This command can be nested like FOR .. NEXT or GOSUB .. RETURN. ▲

DOKE <addr,w>

Writes the word value w into the addresses addr and addr+1, the lower byte of w is in addr. Note if addr = 65535 ($FFFF) then the high byte will be written to address zero. ▲

ELSE

See IF. ▲

END

Terminates program execution and returns control to the command line (direct mode). END may be placed anywhere in a program, it does not have to be on the last line, and there may be any number, including none, of ENDs in total.

Note. CONT may be used after and END to resume execution from the next statement. ▲

FN<name>(<expression>)

See DEF. ▲

FOR <var> = <expression> TO <expression> [STEP expression]

Assigns a variable to a loop counter and optionally sets the start value, the end value and the step size. If STEP expression is omitted then a default step size of +1 will be assumed. ▲

GET <var[$]>

Gets a key, if there is one, from the input device. If there is no key waiting then var will be set to 0 and var$ will return a null string "". GET does not halt and execution will continue. ▲

GOSUB <n>

Call a subroutine at line n. Program execution is diverted to line n but the calling point is remembered. Upon encountering a RETURN statement program execution will continue with the next statement (line) after the GOSUB. ▲

GOTO <n>

Continue execution from line number n. ▲

IF <expression> {GOTO<n>|THEN<{n|statement}>}[ELSE<{n|statement}>]

Evaluates expression. If the result of expression is non zero then the GOTO or the statement after the THEN is executed. If the result of expression is zero then execution continues with the next line.

If the result of expression is zero and the optional ELSE clause is included then the statement after the ELSE is executed.

IF .. THEN .. ELSE .. behaves as a single statement so in the line ..

```
IF <expression> THEN <statement one> ELSE <statement two> :
<statement three>
```

.. statement three will always be executed regardless of the outcome of the IF as long as the executed statement was not a GOTO. ▲

INC <var>[,var]...

Increment variables. The variables listed will have their values incremented by one. Trying to increment a string variable will give a type mismatch error. INC A is much faster than doing A=A+1 and INC A,A is slightly faster than doing A=A+2. ▲

INPUT ["$";] <var>[,var]...

Get a variable, or list of variables from the input stream. A question mark, "?", is always output, after the string if there is one, and if further input is required, i.e. there are more variables in the list than the user entered values, then a double question mark, "??", will be output until enough values have been entered.

There are two possible messages that may appear during the execution of an input statement:

Extra ignored

The user has attempted to enter more values than are required. Program execution will continue but the extraneous data entered has been discarded.

Redo from start

The user has attempted to enter a string where a number was expected. The reverse never causes an error as numbers are also valid strings. ▲

IRQ {ON|OFF|CLEAR}

Enables or disables the IRQ handling subroutine. Note that turning the handler off does not suppress the interrupt detection and if an interrupt occurs while handling is off it will be

actioned as soon as handling is turned back on. Using CLEAR clears the interrupt assignment and it can only be restarted with an ON IRQ command. ▲

LET <var> = <expression>

Assign the value of expression to var. Both var and expression bust be of the same type. The LET command word is optional and just <var> = <expression> will give exactly the same result. It is only maintained for historical reasons. ▲

LIST [n1][-n2]

Lists the entire program held in memory. If n1 is specified then the listing will start from line n1 and run to the end of the program. If -n2 is specified then the listing will terminate after line n2 has been listed. If n1 and -n2 are specified then all the lines from n1 to n2 inclusive will be listed.

Note. If n1 does not exist then the list will start from the next line numbered after n1. If n2 does not exist then the listing will stop with the last line numbered before n2.

Also note. LIST can be executed from within a program, first a [CR][LF] is printed and then the specified lines, if any, each terminated with another [CR][LF]. Program execution then continues as normal. ▲

LOAD

Does nothing in this version but does it via a vector in RAM so is easily patched. ▲

LOOP [{UNTIL|WHILE} expression]

Marks the end of a DO .. LOOP loop. There are three possible variations on the LOOP command ..

> **LOOP**
>
> This loop repeats forever. With just this command control is passed back to the next command after the corresponding DO.
>
> **LOOP UNTIL expression**
>
> This loop will repeat until the value of expression is non zero. Once that occurs execution will continue with the next command after the LOOP UNTIL.
>
> **LOOP WHILE expression**
>
> This loop will repeat while the value of expression is non zero. When the value of expression is zero execution will continue with the next command after the LOOP WHILE. ▲

NEW

Deletes the current program and all variables from memory. ▲

NEXT [var[,var]...]

Increments or decrements a loop variable and checks for the terminating condition. If the terminating condition has been reached then execution continues with the next command, else execution continues with the command after the FOR assignment. *See FOR*. ▲

NMI {ON|OFF|CLEAR}

Enables or disables the NMI handling subroutine. Note that turning the handler off does not suppress the interrupt detection and if an interrupt occurs while handling is off it will be actioned as soon as handling is turned back on. Using CLEAR clears the interrupt assignment and it can only be restarted with an ON NMI command. ▲

NOT <expression>

Generates the bitwise NOT of then signed integer value of <expression>. ▲

NULL <n>

Sets the number of null characters printed by BASIC after every carriage return. n may be specified in the range 0 to 255. ▲

OFF

See IRQ or NMI. ▲

ON <expression> {GOTO|GOSUB} <n>[,n]...

The integer value of expression is calculated and then the nth number after the GOTO or GOSUB is taken (where n is the result of expression). Note that valid results for expression range only from zero to 255. Any result outside this range will cause a Function call error. ▲

ON {IRQ|NMI} <n>

Set up the IRQ or NMI routine pointers. This sets up the effective GOSUB line that is taken when an interrupt happens. When the effective GOSUB is taken the interrupt, IRQ or NMI, is turned off. This can be turned back on with the interrupt on command or by using the matching special return. The normal program flow is resumed by any of RETIRQ, RETNMI or RETURN. ▲

POKE <addr,b>

Writes the byte value b into the address addr. ▲

PRINT [expression][{;|,}expression]...[{;|,}]

Outputs the value of each expressions. If the list of expressions to be output does not end with a comma or a semi-colon, then a carriage return and linefeed is output after the values.

Expressions on the line can be separated with either a semi-colon, causing the next expression to follow immediately, or a comma which will advance the output to the next tab stop before continuing to print. If there are no expressions and no comma or semi-colon after the PRINT statement then a carriage return and linefeed is output.

When entering a program line, or immediate statement, PRINT can be abbreviated to ? ▲

READ <var>[,var]...

Reads values from DATA statements and assigns them to variables. Trying to read a string literal into a numeric variable will cause a syntax error. ▲

REM

Everything following this statement on this program line will be ignored, even colons. ▲

RESTORE [n]

Reset the DATA pointer. If n is specified then the pointer will be reset to the beginning of line n else it will be reset to the start of the program. If n is specified but doesn't exist an error will be generated. ▲

RETIRQ

Returns program execution to the next statement after an interrupt, automatically restores the IRQ enabled flag. See ON IRQ. ▲

RETNMI

Returns program execution to the next statement after an interrupt, automatically restores the NMI enabled flag. See ON NMI. ▲

RETURN

Returns program execution to the next statement (line) after the last GOSUB encountered. See GOSUB. Also returns program execution to the next statement after an interrupt but does not restore the enabled flags. ▲

RUN [n]

Begins execution of the program currently in memory at the lowest numbered line. RUN erases all variables and functions, resets FOR .. NEXT, GOSUB .. RETURN and DO ..LOOP states and sets the data pointer to the program start.

If n is specified then programme execution will start at the specified line number. ▲

SAVE

Does nothing in this version but does it via a vector in RAM so is easily patched. ▲

SPC(<expression>)

Prints <expression> spaces. This command is only valid in a PRINT statement. ▲

STEP

Sets the step size in a FOR .. NEXT loop. See FOR. ▲

STOP

Halts program execution and generates a "Break in line n" message where n is the line in which the STOP was encountered. ▲

SWAP <var[$]>,<var[$]>

Swap two variables. The variables listed will have their values exchanged. Both must be of the same type, numeric or string, and either, or both, may be array elements. Trying to swap a numeric and string variable will give a type mismatch error. ▲

TAB(<expression>)

Sets the cursor position to <expression>. If the cursor is already beyond that point then the cursor will be left where it is. This command is only valid in a PRINT statement. ▲

THEN

See IF. ▲

TO

Sets the range in a FOR .. NEXT loop. See FOR. ▲

UNTIL

See DO and LOOP. ▲

WAIT <addr,b1>[,b2]

Program execution will wait at this point until the value of the location addr exclusive ORed with b2 then ANDed with b1 is non zero. If b2 is not defined then it is assumed to be zero. Note b1 and b2 must both be byte values. ▲

WHILE

See DO and LOOP. ▲

WIDTH {b1|,b2|b1,b2}

Sets the terminal width and TAB spacing. b1 is the terminal width and b2 is the tab spacing, default is 80 and 14. Width can be zero, for "infinite" terminal width, or from 16 to 255. The tab size is from 2 to width-1 or 127, whichever is smaller. ▲

BASIC Operators

Operators perform mathematical or logical operations on values and return the result. The operation is usually preceded by a variable name and equality sign or is part of an IF .. THEN statement.

+	Add. c = a + b will assign the sum of a and b to c.
-	Subtract. c = a - b will assign the result of a minus b to c.
*	Multiply. c = a * b will assign the product of a and b to c.
/	Divide. c = a / b will assign the result of a divided by b to c.
^	Raise to the power of. c = a ^ b will assign the result of a raised to the power of b to c.
AND	Logical AND. c = a AND b will assign the logical AND of a and b to c
EOR	Logical Exclusive OR. c = a EOR b will assign the logical exclusive OR of a and b to c.
OR	Logical OR. c = a OR b will assign the logical inclusive OR of a and b to c.
<<	Shift left. c = a << b will assign the result of a shifted left by b bits to c.
>>	Shift right. c = a >> b will assign the result of a shifted right by b bits to c.
=	Equals. c = a = b will assign the result of the comparison a = b to c.
>	Greater than. c = a < b will assign the result of the comparison a > b to c.
<	Less than. c = a < b will assign the result of the comparison of a < b to c.

The three comparison operators can be mixed to provide further operators ..

>= or =>	Greater than or equal to.
<= or =<	Less than or equal to.
<> or ><	Not equal to (greater than or less than).
<=> any order	Always true (greater than or equal to or less than). ▲

BASIC Functions

Functions always return a value, be it numeric or string, so are used on the right hand side of the = sign in assignments, on either side of operators and in commands requiring an expression e.g. after PRINT, within expressions, or in other functions.

ABS(<expression>)

Returns the absolute value of <expression>. ▲

ASC(<expression$>)

Returns the ASCII value of the first character of <expression$>. ▲

ATN(<expression>)

Returns, in radians, the arctangent of <expression>. ▲

BIN$(<expression>[,b])

Returns <expression> as a binary string. If b is omitted, or if b = 0, then the string is returned with all leading zeroes removed and is of variable length. If b is set, permissible values range from 1 to 24, then a string of length b will be returned. The result is always unsigned and calling this function with expression > 2^24-1 or b > 24 will cause a function call error. ▲

BITTST(<addr>,<b>)

Tests bit b of address addr. Valid bit numbers are 0, the least significant bit, to 7, the most significant bit. Values outside this range will cause a function call error. Returns zero if the bit was zero, returns -1 if the bit was 1. ▲

COS(<expression>)

Returns the cosine of the angle <expression> radians. ▲

DEEK(<addr>)

Returns the word value of <addr> and addr+1 as an integer in the range -32768 to 32767. Addr holds the word low byte. ▲

EXP(<expression>)

Returns e^<expression>. Natural antilog. ▲

FRE(<expression>)

Returns the amount of free program memory. The value of expression is ignored and can be numeric or string. ▲

HEX$(<expression>[,b])

Returns <expression> as a hex string. If b is omitted, or if b = 0, then the string is returned with all leading zeroes removed and is of variable length. If b is set, permissible values range from 1 to 6, then a string of length b will be returned. The result is always unsigned and calling this function with expression > 2^24-1 or b > 6 will cause a function call error. ▲

INT(<expression>)

Returns the integer of <expression>. ▲

LCASE$(<expression$>)

Returns <expression$> with all the alpha characters in lower case. ▲

LEFT$(<expression$,b>)

Returns the leftmost b characters of <expression$>. ▲

LEN(<expression$>)

Returns the length of <expression$>. ▲

LOG(<expression>)

Returns the natural logarithm (base e) of <expression>. ▲

MAX(<expression>[,<expression>]...)

Returns the maximum value from a list of numeric expressions. There must be at least one expression but the upper limit is dictated by the line length. Each expression is evaluated in turn and the largest of them returned. ▲

MID$(<expression$,b1>[,b2])

Returns the substring string from character b1 of expression$ of length b2. The characters of expression$ are numbered from 1 starting with the leftmost. If b2 is omitted then all the characters from b1 to the end of the string are returned. ▲

MIN(<expression>[,<expression>]...)

Returns the minimum value from a list of numeric expressions. There must be at least one expression but the upper limit is dictated by the line length. Each expression is evaluated in turn and the smallest of them returned. ▲

PEEK(<addr>)

Returns the byte value of <addr>. ▲

PI

Returns the value of pi as 3.14159274 (closest floating value). ▲

POS(<expression>)

Returns the POSition of the cursor on the terminal line. The value of expression is ignored. ▲

RIGHT$(<expression$,b>)

Returns the rightmost b characters of <expression$>. ▲

RND(<expression>)

Returns a random number in the range 0 to 1. If the value of <expression> is non zero then it will be used as the seed for the returned pseudo random number otherwise the next number in the sequence will be returned. ▲

SADD(<{var$|var$()}>)

Returns the address of var$ or var$(). This returns a pointer to the actual string in memory not the descriptor. If you want the pointer to the descriptor use VARPTR instead. ▲

SGN(<expression>)

Returns the sign of <expression>. If the value is positive SGN returns +1, if the value is negative then SGN returns -1. If expression=0 then SGN returns 0. ▲

SIN(<expression>)

Returns the sine of the angle <expression> radians. ▲

SQR(<expression>)

Returns the square root of <expression>. ▲

STR$(<expression>)

Returns the result of <expression> as a string. ▲

TAN(<expression>)

Returns the tangent of the angle <expression> radians. ▲

TWOPI

Returns the value of 2*pi as 6.28318548 (closest floating value). ▲

UCASE$(<expression$>)

Returns <expression$> with all the alpha characters in upper case. ▲

CHR$(b)

Returns single character string of character <b>. ▲

USR(<expression>)

Takes the value of <expression> and places it in FAC1 and then calls the USeR routine pointed to by the vector at $0B,$0C. What the routine does with this value is entirely up to the user, it can even be safely ignored if it isn't needed. The routine, after the user code has done an RTS, takes whatever is in FAC1 and returns that. Note it can be either a numeric or string value. See using USR() for details.

If no value needs to be passed or returned then CALL is a better option. ▲

VAL(<expression$>)

Returns the value of <expression$>. ▲

VARPTR(<var[$]>)

Returns a pointer to the variable memory space. If the variable is numeric, or a numeric array element, then VARPTR returns the pointer to the packed value of that variable in memory. If the variable is a string, or a string array element, then VARPTR returns a pointer to the descriptor for that string. If you want the pointer to the string itself use SADD instead. ▲

BASIC Error Messages

These all occur from time to time and, if the error occurred while executing a program, will be followed by "in line " where is the number of the line in which the error occurred.

Array bounds Error

An attempt was made to access an element of an array that was outside it's bounding dimensions.

Can't continue Error

Execution can't be continued because either the program execution ended because an error occurred, NEW or CLEAR have been executed since the program was interrupted or the program has been edited.

Divide by zero Error

The right hand side of an A/B expression was zero.

Double dimension Error

An attempt has been made to dimension an already dimensioned array. This could be because the array was accessed previously causing it to be dimensioned by default.

Function call Error

Some parameter of a function was outside it's limits. E.g. Trying to POKE a value of less than 0 or greater than 255.

Illegal direct Error

An attempt was made to execute a command or function in direct mode which is disallowed in that mode e.g. INPUT or DEF.

LOOP without DO Error

LOOP has been encountered and no matching DO could be found.

NEXT without FOR Error

NEXT has been encountered and no matching FOR could be found.

Out of DATA Error

A READ has tried to read data beyond the last item. Usually because you either mistyped the DATA lines, miscounted the DATA, RESTOREd to the wrong place or just plain forgot to restore.

Overflow Error

The result of a calculation has exceeded the numerical range of BASIC. This is plus or minus 1.7014117+E38

Out of memory Error

Anything that uses memory can cause this but mostly it's writing and running programmes that does it.

RETURN without GOSUB Error

RETURN has been encountered and no matching GOSUB could be found.

String too complex Error

A string expression caused an overflow on the descriptor stack. Try splitting the expression into smaller pieces.

String too long Error

String lengths can be from zero to 255 characters, more than that and you will see this.

Syntax Error

Just generally wrong. 8^)=

Type mismatch Error

An attempt was made to assign a numeric value to a string variable, a string value to a numeric variable or a value of one type was returned when a value of the other type was expected or an attempt at a relational operation between a string and a number was made.

Undefined function Error

FN <var> was called but not found.

Undefined statement Error

Either a GOTO, GOSUB, RUN or RESTORE was attempted to a line that doesn't exist or the line referred to in an ON <expression> {GOTO|GOSUB} or ON {IRQ|NMI} doesn't exist.

Enhanced BASIC, advanced examples

Creating buffer space.

Sometimes there is a need for a byte oriented buffer space. This can be achieved by lowering the top of BASIC memory and using the "protected" space created thus. The main problem with this is that there may not be the same RAM configuration in all the systems this code is to run on.

One way round this is to allocate the space from BASIC's array memory by dimensioning an array big enough to hold your data. As arrays always start from zero then to work out the array size needed you do ..

Array dimension = (bytes needed/4)-1.

E.g.

```
10 DIM b1(19) : REM need 80 bytes for input buffer
20 DIM b2($100) : REM need $0400 bytes for screen buffer
```

So you've allocated the buffer but where is it? This is one use of the VARPTR function, it is used in this case to return the start of the array's data space.

E.g.

```
100 a1 = VARPTR(b1(0)) : REM get the address of the buffer space
```

But wait, there is another problem here. Because variables are created when they are first assigned a value any new variable created after the array is dimensioned will move the array in memory. So the following will not work..

```
10 DIM b1(19) : REM 80 bytes for buffer
20 a1 = VARPTR(b1(0)) : REM get the address of the buffer space
40 FOR x = 0 to 79
50 POKE a1+x,ASC(" ")
60 NEXT
.
.
```

When we get to line 40, a1, the pointer to the array data space, is wrong because the variable x has been created and moved all the arrays up by six bytes. The way round this is to ensure that all variables that you will use have been created prior to getting the pointer. This also means you start with known values in all your variables.

```
10 DIM b1(19) : REM 80 bytes for buffer
20 x = 0 : REM loop counter
30 a1 = VARPTR(b1(0)) : REM get the address of the buffer space
40 FOR x = 0 to 79
50 POKE a1+x,ASC(" ")
60 NEXT
.
.
```

Another way is to get the pointer every time you use it. This has the advantage of always being correct but is somewhat slower.

```
10 DIM b1(19) : REM 80 bytes for buffer
40 FOR x = 0 to 79
50 POKE VARPTR(b1(0))+x,ASC(" ")
60 NEXT
.
.
```

One thing to remember, never try to use a string array as a buffer. Everything will seem to work until you run out of string space and the garbage collection routine is called. Once this happens it's likely that your buffer will get trashed and you may even find that the program freezes because the garbage collection routine now thinks that there are more string bytes than there are memory.

Creating short code space.

While the techniques explained above can also be used to create space for machine code routines there is a simpler way for position independent routines up to 255 bytes long to be held in memory.

Assemble the code and use the hex output from your assembler to create a set of BASIC data statements.

E.g.

```
1000 DATA $A5,$11,$C9,$3A,$B0,$08,$38,$E9
1010 DATA $30,$38,$E9,$D0,$90,$0D,$09,$20
1020 DATA $38,$E9,$61,$90,$0B,$C9,$06,$B0
1030 DATA $07,$69,$3A,$E9,$2F,$85,$11,$60
1040 DATA $18,$60
1050 DATA -1
```

Now we just use a loop like this to load this hex code into a string.

```
10 RESTORE 1000
20 READ by : REM assume at least one byte
30 DO
40 co$ = co$+CHR$(by)
50 READ by
60 LOOP UNTIL by=-1
```

The code can now be called by doing ..

```
140 CALL(SADD(co$))
```

Note that you must always use the SADD() function to get the address for the CALL as the garbage collection routine may move the string in memory and this is the best way to ensure that the address is always correct.

Coding for speed

Spaces

Remove spaces from your code. Spaces, while they don't affect the program flow, do take a finite time to skip over. The only space you don't need to worry about is the one between the line number and the code as this is stripped during input parsing and the apparent space is generated by the LIST command output.

E.g. the following ..

```
10 REM line 10
20   REM line 20
30     REM line 30
```

.. reads as follows when LISTed

```
10 REM line 10
20 REM line 20
30 REM line 30
```

Removing REM.

Remove remarks from your code. Remarks like spaces don't do anything, program wise, but take time to skip. Removing remarks, especially from time critical code, can make a big difference.

Variables.

Use variables. One place where time is wasted, especially in loops, is repeatedly interpreting numeric values or unchanging functions.

E.g.

```
.
140 FOR x = 0 to 79
150 POKE $F400+x,ASC(" ")
160 NEXT
.
```

This loop can be improved in a number of ways. First assign a variable the value $F400 and use that. Doing this is faster after only three uses.

E.g.

```
10 a1 = $F400
.
140 FOR x = 0 to 79
150 POKE a1+x,ASC(" ")
160 NEXT
.
```

The other way to make this loop faster is to assign the value of the (unchanging) function to a variable, then move the function outside the loop.

E.g.

```
10 a1 = $F400
.
130 sp = ASC(" ")
140 FOR x = 0 to 79
150 POKE a1+x,sp
160 NEXT
.
```

Now the ASC(" ") is only evaluated once and the loop is executed faster.

GOTO and GOSUB

When EhBASIC encounters a GOTO or GOSUB it has to search through memory for the target line. If the target line follows the command then it searches from the next line, if the target line precedes the command then the search starts from the beginning of program memory. So keeping this distance, in lines, as short as possible will make the program run faster.

One place that this is difficult is in a conditional loop. In calculating points in the Mandelbrot set, for example, code like this is used ..

```
.
230 INC it
235 tp = mx*mx-my*my+x
240 my = 2*mx*my+y
245 mx = tp
250 co = (mx*mx + my*my)
255 IF (it<128) AND (co<4.0) THEN 230
.
```

Each time the condition in line 255 is met the interpreter has to search from the start of memory for line 230. While this may not take long if the program is short it can slow longer programs considerably.

This can easily be resolved though by using a DO .. LOOP instead. So our example code becomes..

```
.
220 DO
230 INC it
235 tp = mx*mx-my*my+x
240 my = 2*mx*my+y
245 mx = tp
250 co = (mx*mx + my*my)
255 LOOP WHILE (it<128) AND (co<4.0)
.
```

This is quicker because the location of the start of the loop, the DO, is placed on the stack and the interpreter doesn't have to search for it.

Packing them in.

Another way to speed up time critical code is to place as many commands as possible on each line, this can make a noticeable speed gain.

E.g.

```
10 a1 = $F400
.
130 sp = ASC(" ")
140 FOR x = 0 to 79 : POKE a1+x,sp : NEXT
.
```

INC and DEC.

INCrement and DECrement are quick and clear ways of altering a numeric value by plus or minus one and are faster than using add or subtract.

E.g.

```
100 INC a
```

.. is quicker than ..

```
100 a = a+1
```

.. and ..

```
100 INC a,a
```

.. is still quicker than ..

```
100 a = a+2
```

Also combine increments or decrements if you can.

E.g.

```
100 INC so,d
```

.. is quicker than ..

```
100 INC so : INC de
```

>> and <<

Using >> and << can be quicker than using / or * where integer math and a power of two is involved.

E.g. you want to find the byte that holds the pixel at x,y in a 256 x 32 display

```
100 ad = y*32 + INT(x/8) : REM pixel address
```

.. is done quicker with.

```
100 ad = y<<5 + x>>3 : REM pixel address
```

Coding for space

Most of the techniques used to improve the speed of a program can also reduce the number of bytes used by that program.

Spaces.

Remove spaces from your code. The only space you don't need to worry about is the one between the line number and the code as this is stripped during input parsing and the apparent space is generated by the LIST command output.

Removing REM.

Remove remarks from your code. Remarks like spaces don't do anything, removing remarks, can save a lot of space.

Variables.

Use variables. Often you will find yourself using the same numeric value again and again. If this value has many digits, such as the value for e (2.718282), then assigning that value at the beginning of the program can start to save space with the third use.

Re-use variables. Every time you assign a new variable a value it takes up six more bytes of the available memory. If you have a variable that is only used as a loop counter then try to use it for temporary values or GET values elsewhere in the program.

Constants.

There are two constants defined in EhBASIC, PI and TWOPI. They are the closest floating values to pi and 2*pi and will save you seven bytes each time you can use them.

Packing them in.

Another way to save space is to place as many commands as possible on each line, this will save you five bytes every time you put another command on an existing line compared to using a new line.

INC and DEC.

INCrement and DECrement also save space. Either will save you three bytes for each variable INCremented or DECremented.

Derived functions

The following functions, while not part of BASIC, can be calculated using the existing BASIC functions.

Secant	SEC(X)=1/COS(X)
Cosecant	CSC(X)=1/SIN(X)
Cotangent	COT(X)=1/TAN(X)
Inverse sine	ARCSIN(X)=ATN(X/SQR(X*X+1))
Inverse cosine	ARCCOS(X)=-ATN(X/SQR(X*X+1))+PI/2
Inverse secant	ARCSEC(X)=ATN(SQR(X*X-1))+(SGN(X)-1)*PI/2
Inverse cosecant	ARCCSC(X)=ATN(1/SQR(X*X-1))+(SGN(X)-1)*PI/2
Inverse cotangent	ARCCOT(X)=-ATN(X)+PI/2
Hyperbolic sine	SINH(X)=(EXP(X)-EXP(-X))/2
Hyperbolic cosine	COSH(X)=(EXP(X)+EXP(-X))/2
Hyperbolic tangent	TANH(X)=-EXP(-X)/(EXP(X)+EXP(-X))*2+1
Hyperbolic secant	SECH(X)=2/(EXP(X)+EXP(-X))
Hyperbolic cosecant	CSCH(X)=2/(EXP(X)-EXP(-X))
Hyperbolic cotangent	COTH(X)=EXP(-X)/(EXP(X)-EXP(-X))*2+1
Inverse hyperbolic sine	ARCSINH(X)=LOG(X+SQR(X*X+1))
Inverse hyperbolic cosine	ARCCOSH(X)=LOG(X+SQR(X*X-1))
Inverse hyperbolic tangent	ARCTANH(X)=LOG((1+X)/(X))/2
Inverse hyperbolic secant	ARCSECH(X)=LOG((SQR(X*X+1)+1)/X)
Inverse hyperbolic cosecant	ARCCSCH(X)=LOG((SGN(X)*SQR(X*X+1)+1)/X)
Inverse hyperbolic cotangent	ARCCOTH(X)=LOG((X+1)/(X-1))/2

Enhanced BASIC, extending CALL

Introduction.

CALL <address> calls a machine code routine at location address. While this in itself is useful it can be extended by adding parameters to the CALL and parsing them from within the routine.
This technique can also be used to pass extra parameters to the USR() function.

How to.

First you need to define the parameters for your CALL. This example is for an imaginary bitmapped graphic device.

CALL PLOT,x,y Set the pixel at x,y

PLOT	routine address
x	x axis value, range 0 to 255
y	y axis value, range 0 to 64

This will then be the form that the call will always take.

Now you need to write the code.

```
.include BASIC.DIS      ; include the BASIC labels file. this allows you
                        ; easy access to the internal routines you need
                        ; to parse the command stream and access some of
                        ; the internals of BASIC. It is usually output
                        ; by the assembler as part of the listing or as a
                        ; separate, optional, file.

; for now we'll put this in the spare RAM @ $F400

        *=      $F400

PLOT
        JSR     LAB_SCGB        ; scan for "," and get byte
        STX     PLOT_XBYT       ; save plot x
        JSR     LAB_SCGB        ; scan for "," and get byte
        CPX     #$40            ; compare with max+1
        BCS     PLOT_FCER       ; if 64d or greater do function call error

        STX     PLOT_YBYT       ; save plot y

; now would be your code to perform the plot command
;.
;.
;.
;.
;.

        RTS                     ; return to BASIC

; does BASIC function call error

PLOT_FCER
        JMP     LAB_FCER        ; do function call error, then warm start

; now we just need the variable storage

PLOT_XBYT
        .byte   $00             ; set default

PLOT_YBYT
        .byte   $00             ; set default

        END
```

Finally you need to set the value of PLOT in your BASIC program and use that to call it.

E.g.

```
.
10 PLOT = $F400
.
.
145 CALL PLOT,25,14 : REM set pixel
.
```

Enhanced BASIC, using USR()

Introduction.

USR(<expression[$]>) calls the machine code function pointed to by the user jump vector after evaluating <expression[$]> and placing the result in the first floating accumulator. Once the user function exits, via an RTS, the value in the floating accumulator is passed back to EhBASIC.

Either a numeric value or a string can be passed, and either type can be returned depending on the setting of the data type flag at the end of the user code and the return point (see code examples for details).

It can also be extended by adding parameters to USR() and parsing them from within the routine in the same way that CALL can be extended, just remember to get the value from FAC1 first.

How to - numeric source, numeric result.

First you need to write the code.

```
; this code demonstrates the use of USR() to quickly calculate the square of a
; byte value. Compare this with doing SQ=A*A or even SQ=A^2.

        .include BASIC.DIS      ; include the BASIC labels file. this allows
                                ; you easy access to the internal routines you
                                ; need to parse the command stream and access
                                ; some of the internals of BASIC. It is usually
                                ; output by the assembler as part of the listing
                                ; or as a separate, optional, file.

; for now we'll put this in the spare RAM @ $F400

        *=      $F400

Square
        JSR     LAB_EVBY        ; evaluate byte expression, result in X and FAC1_3
        LDA     #$00            ; clear A
        STA     FAC1_2          ; clear square low byte (use FAC1 as the workspace)
                                ; (no need to clear the high byte, it gets shifted out)
        TXA                     ; copy byte to A
        LDX     #$08            ; set bit count
Nextr2bit
        ASL     FAC1_2          ; low byte *2
        ROL     FAC1_1          ; high byte *2+carry from low
```

```
        ASL     A               ; shift byte
        BCC     NoSqadd         ; don't do add if C = 0

        TAY                     ; save A
        CLC                     ; clear carry for add
        LDA     FAC1_3          ; get number
        ADC     FAC1_2          ; add number^2 low byte
        STA     FAC1_2          ; save number^2 low byte
        LDA     #$00            ; clear A
        ADC     FAC1_1          ; add number^2 high byte
        STA     FAC1_1          ; save number^2 high byte
        TYA                     ; get A back
NoSqadd
        DEX                     ; decrement bit count
        BNE     Nextr2bit       ; go do next bit

        LDX     #$90            ; set exponent=2^16 (integer)
        SEC                     ; set carry for positive result
        JMP     LAB_STFA        ; set exp=X, clearFAC1 mantissa3, normalise & return
```

Now you need to set up the address for your function. This is done by DOKEing an address into the USR() vector e.g.

```
DOKE $0B,$F400          ; set the user function address to addr
                        ; $0B   - user function vector address
                        ; $F400 - routine address
```

Finally you need to set the vector in your BASIC program and use that to call the function

```
.
10 DOKE $0B,$F400
.
.
.
145 SQ=USR(A)
.
```

How to - numeric source, string result.

AS before, first you need to write the code.

```
; this code demonstrates the use of USR() to generate a string of # characters.
; the length of the required string is the parameter passed.

        .include BASIC.DIS      ; include the BASIC labels file. this allows
                                ; you easy access to the internal routines you
                                ; need to parse the command stream and access
                                ; some of the internals of BASIC. It is usually
                                ; output by the assembler as part of the listing
                                ; or as a separate, optional, file.

; for now we'll put this in the spare RAM @ $F400

        *=      $F400

STRING
        JSR     LAB_EVBY        ; evaluate byte expression, result in X and FAC1_3

        TXA                     ; string is byte length
```

```
        BEQ     NUL_STRN        ; branch if null string

        JSR     LAB_MSSP        ; make string space A bytes long A=$AC=length,
                                ; X=$AD=Sutill=ptr low byte,
                                ; Y=$AE=Sutilh=ptr high byte
        LDA     #"#"            ; set character
        LDY     FAC1_3          ; get length
SAV_HASH
        DEY                     ; decrement bytes to do
        STA     (str_pl),Y      ; save byte in string
        BNE     SAV_HASH        ; loop if not all done

NUL_STRN
        PLA                     ; dump return address (return via get value
        PLA                     ; from line, this skips the type checking and
                                ; so allows a string result to be returned)
        JMP     LAB_RTST        ; check for space on descriptor stack then put
                                ; string address and length on descriptor stack
                                ; & update stack pointers
```

Now you need to set up the address for your function. This is done by DOKEing an address into the USR() vector e.g.

```
DOKE $0B,$F400          ; set the user function address to addr
                        ; $0B   - user function vector address
                        ; $F400 - routine address
```

Finally you need to set the vector in your BASIC program and use that to call the function

```
.
10 DOKE $0B,$F400
.
.
.
145 HA$=USR(A)
.
```

How to - string source, numeric result.

AS before, first you need to write the code.

```
; this code demonstrates the use of USR() to test a string of characters.
; if all the string is alpha -1 is returned, else 0 is returned.

        .include BASIC.DIS      ; include the BASIC labels file. this allows
                                ; you easy access to the internal routines you
                                ; need to parse the command stream and access
                                ; some of the internals of BASIC. It is usually
                                ; output by the assembler as part of the listing
                                ; or as a separate, optional, file.

; for now we'll put this in the spare RAM @ $F400

        *=      $F400

ALPHA
        JSR     LAB_EVST        ; evaluate string
```

```
        TAX                     ; copy length to X
        BEQ     NOT_ALPH        ; branch if null string

        LDY     #$00            ; clear index
ALP_LOOP
        LDA     (ut1_pl),Y      ; get byte from string
        JSR     LAB_CASC        ; is character "a" to "z" (or "A" to "Z")
        BCC     NOT_ALPH        ; branch if not alpha

        INY                     ; increment index
        DEX                     ; decrement count
        BNE     ALP_LOOP        ; loop if not all done

        LDA     #$FF            ; set for -1
        BNE     IS_ALPHA        ; branch always

NOT_ALPH
        LDA     #$00            ; set for 0
IS_ALPHA
        TAY                     ; copy byte
        LDX     #$00            ; clear byte
        STX     Dtypef          ; clear data type flag, $00=numeric
        JMP     LAB_AYFC        ; save & convert integer AY to FAC1 & return
```

Now you need to set up the address for your function. This is done by DOKEing an address into the USR() vector e.g.

```
DOKE $0B,$F400          ; set the user function address to addr
                        ; $0B   - user function vector address
                        ; $F400 - routine address
```

Finally you need to set the vector in your BASIC program and use that to call the function

```
.
10 DOKE $0B,$F400
.
.
.
145 AL=USR(A$)
.
```

How to - string source, string result.

AS before, first you need to write the code.

```
; this code demonstrates the use of USR() invert the case of a string of
; characters. only alpha characters will be affected.

        .include BASIC.DIS      ; include the BASIC labels file. this allows
                                ; you easy access to the internal routines you
                                ; need to parse the command stream and access
                                ; some of the internals of BASIC. It is usually
                                ; output by the assembler as part of the listing
                                ; or as a separate, optional, file.

; for now we'll put this in the spare RAM @ $F400

        *=      $F400
```

```
ALPHA
        JSR     LAB_EVST        ; evaluate string
        STA     str_ln          ; set string length
        STX     str_pl          ; set string pointer low byte
        STY     str_ph          ; set string pointer high byte
        TAX                     ; copy length to X
        BEQ     NO_STRNG        ; branch if null string

        LDY     #$00            ; clear index
ALP_LOOP
        LDA     (ut1_pl),Y      ; get byte from string
        JSR     LAB_CASC        ; is character "a" to "z" (or "A" to "Z")
        BCC     NOT_ALPH        ; branch if not alpha

        EOR     #$20            ; toggle case
        STA     (ut1_pl),Y      ; save byte back to string
NOT_ALPH
        INY                     ; increment index
        DEX                     ; decrement count
        BNE     ALP_LOOP        ; loop if not all done

NO_STRNG
        PLA                     ; dump return address (return via get value
        PLA                     ; from line, this skips the type checking and
                                ; so allows a string result to be returned)
        JMP     LAB_RTST        ; check for space on descriptor stack then put
                                ; string address and length on descriptor stack
                                ; & update stack pointers
```

Now you need to set up the address for your function. This is done by DOKEing an address into the USR() vector e.g.

```
DOKE $0B,$F400          ; set the user function address to addr
                        ; $0B   - user function vector address
                        ; $F400 - routine address
```

Finally you need to set the vector in your BASIC program and use that to call the function

```
.
10 DOKE $0B,$F400
.
.
.
145 A$=USR(A$)
.
```

Enhanced BASIC internals

Floating point numbers.

Floating point numbers are stored in memory in four bytes. The format of the numbers is as follows.

Exponent	S	Mantissa 1	Mantissa 2	Mantissa 3

Exponent

This is the power of two to which the mantissa is to be raised. This number is biased to +$80 i.e. 2^0 is represented by $80, 2^1 by $81 etc. Zero is a special case and is used to represent the value zero for the whole of the number.

S

Sign bit. This bit (b7 of mantissa 1) is one if the number is negative.

Mantissa 1/2/3

This is the 24 bit mantissa of the number and is normalised to make the highest bit (b7 of mantissa 1) always one. So the absolute value of the mantissa varies between 0.5 and 0.9999999403954 . As we know that the highest bit is always one it is replaced by the sign bit in memory.

Example.

```
$82,$49,$0F,$DB = +3.14159274  nearest floating equivalent to pi
 |   ||  |   |
 |   |\--+---+- = 0.785398185  absolute value of mantissa
 |   |
 |   \--------- = +            b7 of mantissa 1 is zero
 |
 \------------- = x 2^2 = 4    mantissa to be multiplied by 4
```

Values represented in this way range between + and - 1.70141173x10^38

BASIC program memory use.

A BASIC program is stored in memory from Ram_base upwards. It's format is ..

$00 Start of program marker byte

.. then each BASIC program line which is stored as ..

start of next line pointer low byte
start of next line pointer high byte
line number low byte
line number high byte
code byte(s)
$00 End of line marker byte

.. and finally ..

$00 End of program marker byte 1
$00 End of program marker byte 2

If there is no program in memory only the start and end marker bytes are present.

BASIC variables memory use.

After the program come the variables and function references, all six bytes long, which are stored as ..

1st character of variable or function name (+$80 if FN name)
2nd character of variable or function name (+$80 if string)

.. then for each type ..

Numeric	String	Function
Exponent	String length	BASIC execute pointer low byte
Sign (bit 7) + mantissa 1	String pointer low byte	BASIC execute pointer high byte
Mantissa 2	String pointer high byte	Function variable name 1st character
Mantissa 3	$00	Function variable name 2nd character

After the variables come the arrays, which are stored as ..

1st character of variable name
2nd character of variable name (+$80 if string)
array size in bytes low byte (size includes this header)
array size in bytes high byte
number of dimensions
[dimension 3 size high byte] (lowest element is zero)
[dimension 3 size low byte]
[dimension 2 size high byte] (lowest element is zero)
[dimension 2 size low byte]
dimension 1 size high byte (lowest element is zero)
dimension 1 size low byte

.. and then each element ..

Numeric	String
Exponent	String length
Sign (bit 7) + mantissa 1	String pointer low byte
Mantissa 2	String pointer high byte
Mantissa 3	$00

The elements of every array are stored in the order ..

index1 [0-n], index2 [0-n], index3 [0-n]

i.e. element (1,2,3) in an array of (3,4,5) would be the ..

1 + 1 + 2*(3+1) + 3*(3+1)*(4+1) = 70th element

(As array dimensions range from 0 to n element n will always be the (n+1)th element in memory.)

String placement in memory.

Strings are generally stored from the top of available RAM, Ram_top, working down, however if the interpreter encounters a line such as ..

```
100 A$ = "This is a string"
```

.. then the high/low pointer in the A$ descriptor will point to the string in program memory and will not make a copy of the string in the string memory.

String descriptors in BASIC.

A string descriptor is a three byte table that describes a string, it is of the format ..

base = string length
base+1 = string pointer low byte
base+2 = string pointer high byte

Stack use in BASIC.

GOSUB and DO both push on the stack ..

BASIC execute pointer high byte
BASIC execute pointer low byte
current line high byte
current line low byte
command token (TK_GOSUB or TK_DO)

FOR pushes on the stack ..

BASIC execute pointer low byte
BASIC execute pointer high byte
FOR line high byte
FOR line low byte
TO value mantissa3
TO value mantissa2
TO value mantissa1
TO value exponent
STEP sign
STEP value mantissa3
STEP value mantissa2
STEP value mantissa1
STEP value exponent
var pointer for FOR/NEXT high byte
var pointer for FOR/NEXT low byte
token for FOR (TK_FOR)

Enhanced BASIC, useful routines

Introduction.

There are many subroutines within BASIC that can be useful if you wish to use your own assembly routines with it. Here are some of them with a brief description of their function. For full details see the source code.

Note that most, if not all, of these routines need EhBASIC to be initialised before they will work properly and can not be used in isolation from EhBASIC.

The routines.

LAB_IGBY
BASIC increment and get byte routine. gets the next byte from the BASIC command stream. If the byte is a numeric character then the carry flag will be set, if the byte is a termination byte, either null or a statement separator, then the zero flag will be set. Spaces in the command stream will automatically be ignored.

LAB_GBYT
BASIC get byte routine. Gets the current byte from the BASIC command stream but does not change the pointer. Otherwise the same as above.

LAB_COLD
Performs a cold start. BASIC is reset and all BASIC memory is cleared.

LAB_WARM
Performs a warm start. Execution is stopped and BASIC returns to immediate mode.

LAB_OMER
Do "Out of memory" error, then warm start. The same as error $0C below.

LAB_XERR
With X set, do error #X, then warm start.

X	Error	X	Error
$00	NEXT without FOR	$02	syntax
$04	RETURN without GOSUB	$06	out of data
$08	function call	$0A	overflow
$0C	out of memory	$0E	undefined statement
$10	array bounds	$12	double dimension array
$14	divide by 0	$16	illegal direct
$18	type mismatch	$1A	long string
$1C	string too complex	$1E	continue error
$20	undefined function	$22	LOOP without DO

LAB_INLN
Print "? " and get BASIC input. Returns XY (low/high) as a pointer to the start of the input line. The input is null terminated.

LAB_SSLN
Search Basic for a line, the line number required is held in the temporary integer, from start of program memory. Returns carry set and a pointer to the line in Baslnl/Baslnh if found, if not it returns carry and a pointer to the next numbered line in Baslnl/Baslnh.

LAB_SHLN
Search Basic for temporary integer line number from AX. Same as above but starts the search from AX (low/high).

LAB_SNBS
Scan for next BASIC statement (: or [EOL]). Returns Y as index to : or [EOL] from (Bpntrl).

LAB_SNBL
Scan for next BASIC line. Same as above but only returns on [EOL].

LAB_REM
Perform REM, skip (rest of) line.

LAB_GFPN
Get fixed-point number into temporary integer.

LAB_CRLF
Print [CR]/[LF] to output device.

LAB_PRNA
Print character in A to output device.

LAB_GVAR
Get variable address. Returns a pointer to the variable in Lvarpl/h and sets the data type flag, $FF=string, $00=numeric.

LAB_EVNM
Evaluates an expression and checks the result is numeric, if not it does a type mismatch. The result of the expression is returned in FAC1.

LAB_CTNM
Check if source is numeric, else do type mismatch.

LAB_CTST
Check if source is string, else do type mismatch.

LAB_CKTM
Type match check, set carry for string, clear carry for numeric.

LAB_EVEX
Evaluate expression.

LAB_GVAL
Get numeric value from line. Returns the result in FAC1.
LAB_SCCA
Scan for the byte in A as the next byte. If so return here, else do syntax error then warm start.

LAB_SNER
Do syntax error, then warm start.

LAB_CASC
Check byte is alpha ("A" to "Z" or "a" to "z"), return carry clear if so.

LAB_EVIN
Evaluate integer expression. Return integer in FAC1_3/FAC1_2 (low/high).

LAB_EVPI
Evaluate positive integer expression.

LAB_EVIR
Evaluate integer expression, check is in range -32786 to 32767

LAB_FCER
Do function call error, then warm start.

LAB_CKRN
Check that the interpreter is not in immediate mode. If not then return, if so do illegal direct error.

LAB_GARB
Perform garbage collection routine.

LAB_EVST
Evaluate string.

LAB_ESGL
Evaluate string, return string length in Y.

LAB_SGBY
Scan and get byte parameter, return the byte in X.

LAB_GTBY
Get byte parameter and ensure numeric type, else do type mismatch error. Return the byte in X.

LAB_EVBY
Evaluate byte expression, return the byte in X.

LAB_GADB
Get two parameters as in POKE or WAIT. Return the byte (second parameter) in X and the integer (first parameter) in the temporary integer pair, Itempl/Itemph.

LAB_SCGB
Scan for "," and get byte, else do Syntax error then warm start. Return the byte in X.

LAB_F2FX
New convert float to fixed routine. accepts any value that fits into 24 bits, positive or negative and converts it into a right truncated integer in the temporary integer pair, Itempl/Itemph.

LAB_UFAC
Unpack the four bytes starting (AY) into FAC1 as a floating point number.

LAB_PFAC
Pack the floating point number in FAC1 into the current variable (Lvarpl).

LAB_STFA
Stores a 16 bit number in FAC1. Set X to the exponent required (usually $90) and the carry set for positive numbers and clear for negative numbers. The routine will clear FAC1 mantissa3 and then normalise it.

LAB_AYFC
Save integer AY (A = high byte, Y = low byte) in FAC1 and convert to float. The result will be -32768 to +32767.

LAB_MSSP
Make string space A bytes long. This returns the following. str_ln = A = string length str_pl = Sutill = string pointer low byte str_ph = Sutilh = string pointer high byte

LAB_RTST
Return string. Takes the string described instr_ln, str_pl and str_ph and puts it on the string stack. This is how you return a string to BASIC.