Enhanced 65816 BASIC reference manual

Preface

The manual this was based on was 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.

Most of the content of this manual is the sole intellectual property of the original author Lee Davison. See the copyright notice in the introduction, for what you can and what you cannot do with the source code and with the documentation. Conversion to run on the WE816 platform and the associated disk, graphics, and sound commands was done by Dan Werner.

Content

The original 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 65C816 Port

EhBASIC was ported to the 65816 CPU on the WE816 computer. It is designed to allow a full 64K bank for BASIC code and variables with the BASIC interpreter running in a totally different bank. Program Bank, Data Bank and Stack parameters can be adjusted in the definitions.asm file included in the archive, and zero page usage can be found in the zeropage.asm file.

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.

ABS	AND	ASC	ATN	BIN$	BITCLR
BITSET	BITTST	CALL	CHR$	CLEAR	CLOSE
COLOR	CON	CONT	COS	DATA	DEC
DEEK	DEF	DIM	DIRECTORY	DISKCMD	DISKSTATUS
DO	ELSE	END	EOR	EXP	FN
FOR	FRE	GET#	GET	GOSUB	GOTO
HEX$	IECINPUT	IECOUTPUT	IECST	IF	INC
INPUT	INT	LCASE$	LEFT$	LEN	LET
LIST	LOAD	LOCATE	LOG	LOOP	MAX
MID$	MIN	MONITOR	NEW	NEXT	NOISE
NOT	NULL	OFF	ON	OPEN	OR
PATTERN	PEEK	PI	PLOT	POKE	POS
PRINT	PUT#	READ	REM	RESTORE	RETURN
RIGHT$	RND	RUN	SADD	SAVE	SCNCLR
SCREEN	SGN	SIN	SOUND	SPC	SQR
STEP	STOP	STR$	SWAP	SYS	TAB
TAN	THEN	TO	TONE	UCASE$	UNTIL
USR	VAL	VARPTR	VOICE	VOLUME	WAIT
WHILE	WIDTH	*	+	-	/
<<	=	>>	<	>	

Manual Conventions

- 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.

CLOSE <Logical file b>

Close IEC logical File <b>. See Working with Files for more information.

COLOR <background b>,<foreground b>

Sets the background and foreground colors on the text screen.

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

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.

DIRECTORY <IEC DEVICE b>

Prints the disk directory from device <b>.

DISKCMD <command$>,<IEC DEVICE b>

Sends the disk command to device b. See the disk device documentation for a list of commands and syntax.

Example: **DISKCMD "S0:FILENAME",8**

DO

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

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 an 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# <Logical File b>,<var[$]>

Gets a character, if there is one, from the input channel <b>. This command must be preceded by OPEN and IECINPUT commands. See Working with Files for more information.

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.

IECINPUT <Logical File b>

Designate IEC channel <b> as input. See Working with Files for more information.

IECOUTPUT <Logical File b>

Designate IEC channel <b> as output. See Working with Files for more information.

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.

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 var[$] ,<b>

Load a program named var[$] from IEC channel <b>.

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.

MONITOR

Exit to the system monitor. See WE816 system documentation for more information on how to use the system monitor.

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*.

NOISE <channel b>,<noise period b>

Selects a given channel (0-2) to generate NOISE, rather than TONE, and configures the noise period.

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.

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.

OPEN <Logical File b>,<IEC Device Number b>,<Secondary Address b>,<file name var[$]>

Open a file to an IEC Device. See Working with Files for more information.

PATTERN <character number b>, <b>,<b>,<b>,<b>, <b>,<b>,<b>,<b>

Redefine a character set pattern.

PLOT <b>,<b>,<b>

Plot a pixel to the screen X,Y,Color.

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 ?

PUT# <Logical File b>,<var[$]>

Outputs the char value to IEC channel <b>. This must be preceded with Open and IECOUTPUT commands. See Working with Files for more information.

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.

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 <var[$]>,<b>

Save the program in memory to filename <$> on IEC Device <b>.

SCNCLR

Clear the Text Screen.

SCREEN <b,b> [,b]

Set the graphics screen mode. See Working with Graphics for more information.

0,0	Text Mode (40x24) 16 color
0,1	Text Mode (80x24) 16 color
1,0,0	Single Lores (40,48) 16 color
1,0,1	Single Lores (40,40) 16 color, with 4 line text window
1,1,0	Double Lores (80,48) 16 color
1,1,1	Double Lores (80,40) 16 color, with 4 line text window
2,0,0	Hires (140x192) 16 color
2,0,1	Hires (140x160) 16 color, with 4 line text window
2,1,0	Double Hires (280x192) 16 color
2,1,1	Double Hires (280x160) 16 color, with 4 line text window
2,2,0	Quad Hires (560x192) 2 color
2,2,1	Quad Hires (560x160) 2 color, with 4 line text window
2,3,0	Mono Hires (280x192) 2 color
2,3,1	Mono Hires (280x160) 2 color, with 4 line text window

SOUND <channel b>, <frequency var>

Play a sound on channel (0-2), at the frequency var. See Appendix for musical note to frequency values. Note that “frequency” does not equate to actual frequency of the note provided.

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.

TONE <channel b>

Sets the channel (0-2) to generate TONEs rather than NOISE.

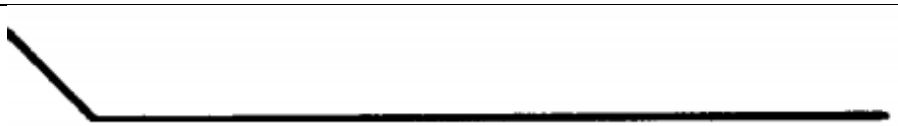
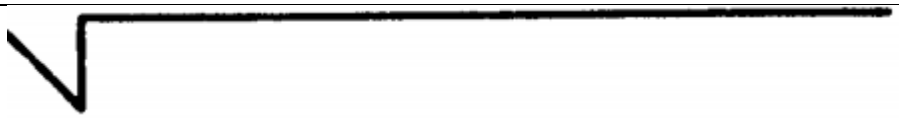
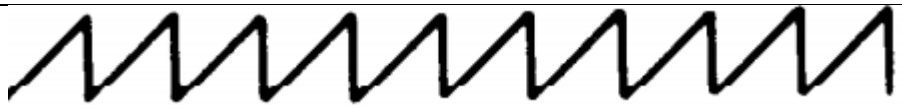
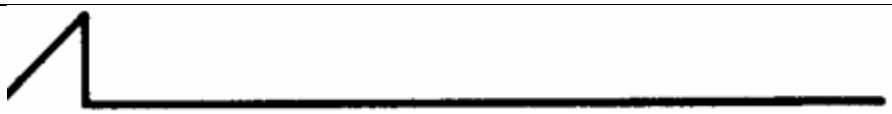
UNTIL

See DO and LOOP.

VOICE <shape b>,<period var>

Set the envelope generator on the chip to a given shape, and period (duration of one cycle).

Shapes:

0	
4	
8	
9	
10	
11	
12	
13	
14	
15	

VOLUME <channel b>,<volume b>

Set the volume of a given channel (0-2) to <B>.

For solid tones, use values 0-15. To use the envelope generator on the chip, use values 16-31.

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.

CON(<b>)

Takes game controller number as one parameter (0,1) and returns the state of that game controller. The status of the controller is encoded like this:

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
		BUTTON B	BUTTON A	RIGHT	LEFT	DOWN	UP

COS(<expression>)

Returns the cosine of the angle <expression> radians.

DISKSTATUS(<b>)

Returns the string value of the disk status of device <b>

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.

IECST(<b>)

Returns the IEC status of channel <b>.

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.

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

Working with Files.

The following is an example BASIC program that writes and reads back a sequential file to the Commodore 1541 Disk Drive. For more advanced drive functions, please see the reference guide for the IEC device you are communicating with.

```
10 OPEN 3,8,4,"TFI2,SEQ,W"
20 IECOUTPUT 3
30 PUT#3,"OUT"+CHR$(13)
31 PUT#3,"OUT1"+CHR$(13)+CHR$(5)
32 PUT#3,"OUT2"+CHR$(13)+CHR$(10)+CHR$(5)
40 CLOSE 3
70 OPEN 4,8,8,"TFI2,SEQ,R"
80 IECINPUT 4
90 GET#4,A
100 PRINT A, IECST
101 IF IECST=0 THEN GOTO 90
110 CLOSE 4
```

Working with Sound.

The AY-3-8910 sound chip is a simple (by today's standards) sound generation chip. It contains 3 voices that can be configured as either tone or noise, and has a simple envelope generator. More specific information on the AY-3-8910 can be found online or in the AY-3-8910 datasheet.
Here are a few sample BASIC programs to generate a series of tones from the WE816.

Program 1:

```
5 REM SIMPLE SERIES OF TONES ON VOICE 0
10 VOLUME 0,15
20 TONE 0
30 I=100
30 FOR I = 100 TO 4000 STEP 100
40 SOUND 0,I
50 FOR X = 0 TO 500: NEXT X
60 NEXT I
70 SOUND 0,0
```

Program 2:

```
4 REM MORE COMPLEX SOUNDS USING THE ENVELOPE
6 REM GENERATOR AND A SAWTOOTH WAVEFORM
10 VOLUME 0,31
20 TONE 0
25 VOICE 8,2000
30 FOR I = 100 TO 4000 STEP 100
40 SOUND 0,I
50 FOR X = 0 TO 500: NEXT X
60 NEXT I
70 SOUND 0,0
```

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.

WE816 Hardware Reference

Using Supermon

The WE816 computer utilizes BCS Technology's (http://sbc.bcstechnology.net/) Supermon 816 monitor program for its machine language monitor. To access Supermon, type 'MONITOR' from BASIC. You will be greeted by Supermon's welcome screen.

```
   __ _____  ___  __   __
  / /| ____|/ _ \/_ | / /
 / /_| |__ | (_) || |/ /_
|  _ \___ \ > _ < | |  _ \
| (_) |__) | (_) || | (_) |
 \___/____/ \___/ |_|\___/

65c816 BIOS (NATIVE MODE)
v0.55 4/9/2021 - D.WERNER
-------------------------------------


PB  PC   NVmxDIZC  .C   .X   .Y   SP
F2 4F58  00110000 0000 0000 0000 7FFF
 DP  DB
0000 00
.
```

From the '.' Prompt, you can enter any of Supermon's commands.

- A Assemble code
- C Compare memory regions
- D Disassemble code
- F Fill memory region (cannot span banks)
- G Execute code (stops at BRK)
- H Search (hunt) memory
- J Execute code as a subroutine (stops at BRK or RTS)
- L Load Motorola S28-records
- M Dump & display memory range
- R Dump & display 65C816 registers
- T Copy (transfer) memory region
- > Modify up to 32 bytes of memory
- ; Modify 65C816 registers

Supermon 816 accepts binary (%), octal (@), decimal (+) and hexadecimal ($) as input for numeric parameters. Additionally, the H and > operations accept an ASCII string in place of numeric values by preceding the string with ', e.g.:

```
h 042000 042FFF 'BCS Technology Limited
```

If no radix symbol is entered hex is assumed.

Numeric conversion is also available. For example, typing:

```
+1234567 [CR]
```

at the monitor's prompt will display:

```
$12D687
+1234567
@04553207
%10010110101101000011 1
```

In the above example, [CR] means the console keyboard's return or enter key.

All numeric values are internally processed as 32 bit unsigned integers. Addresses may be entered as 8, 16 or 24 bit values. During instruction assembly immediate mode operands may be forced to 16 bits by preceding the operand with an exclamation point if the instruction can accept a 16 bit operand, e.g.:

```
a 1f2000 lda !#4
```

The above will assemble as:

```
A 1F2000  A9 04 00     LDA #$0004
```

Entering:

```
a 1f2000 ldx !#+157
```

will assemble as:

```
A 1F2000  A2 9D 00     LDX #$009D
```

Absent the ! in the operand field, the above would have been assembled as:

```
A 1F2000  A2 9D        LDX #$9D
```

If an immediate mode operand is greater than $FF, assembly of a 16 bit operand is implied.

Redirecting Console to the Serial Port

The WE816 computer defaults console input/output to it's built in keyboard and display, however it is possible to change the console display from internal to the serial port. Location $0341 in memory tells the computer which to use. If $341 contains a $01, the system will use the internal console, and value $00 designates the serial port.

To change the console using Supermon 816, enter:

```
>0341 00[ENTER]
```

And the system will change to the serial console. Entering:

```
>0341 01[ENTER]
```

Will change back to the internal video and keyboard.

Returning to BASIC

To return to BASIC from Supermon 816, just execute the “G” command with the address of the BASIC startup code.

```
G FF1000[ENTER]
```

Screen Font

The following is the screen font for the WE816.

Musical Note to "Frequencies"

value	Produced freq	note	Note freq	variance	out of tune
14	7990.313	B8	7902.13	-88.1825	1%
15	7457.625	A8	7040	-417.625	6%
16	6991.523	G#8/Ab8	6644.88	-346.643	5%
17	6580.257	G8	6271.93	-308.327	5%
18	6214.688	F#8/Gb8	5919.91	-294.778	5%
20	5593.219	F8	5587.65	-5.56875	0%
21	5326.875	E8	5274.04	-52.835	1%
22	5084.744	D#8/Eb8	4978.03	-106.714	2%
23	4863.668	D8	4698.63	-165.038	3%
25	4474.575	C#8/Db8	4434.92	-39.655	1%
26	4302.476	C8	4186.01	-116.466	3%
28	3995.156	B7	3951.07	-44.0862	1%
29	3857.392	A#7/Bb7	3729.31	-128.082	3%
31	3608.528	A7	3520	-88.5282	2%
33	3389.83	G#7/Ab7	3322.44	-67.3895	2%
37	3023.361	F#7/Gb7	2959.96	-63.4015	2%
42	2663.438	E7	2637.02	-26.4175	1%
44	2542.372	D#7/Eb7	2489.02	-53.3522	2%
47	2380.093	D7	2349.32	-30.7731	1%
50	2237.288	C#7/Db7	2217.46	-19.8275	1%
53	2110.649	C7	2093	-17.6486	1%
56	1997.578	B6	1975.53	-22.0481	1%
59	1896.006	A#6/Bb6	1864.66	-31.3464	2%
63	1775.625	A6	1760	-15.625	1%

67	1669.618	G#6/Ab6	1661.22	-8.39754	1%
71	1575.555	G6	1567.98	-7.57458	0%
75	1491.525	F#6/Gb6	1479.98	-11.545	1%
84	1331.719	E6	1318.51	-13.2088	1%
89	1256.903	D#6/Eb6	1244.51	-12.3931	1%
95	1177.52	D6	1174.66	-2.85974	0%
100	1118.644	C#6/Db6	1108.73	-9.91375	1%
106	1055.324	C6	1046.5	-8.82429	1%
113	989.9502	B5	987.77	-2.18022	0%
119	940.0368	A#5/Bb5	932.33	-7.70676	1%
127	880.8219	A5	880	-0.82185	0%
134	834.8088	G#5/Ab5	830.61	-4.19877	1%
142	787.7773	G5	783.99	-3.78729	0%
151	740.8237	F#5/Gb5	739.99	-0.83368	0%
160	699.1523	F5	698.46	-0.69234	0%
169	661.9194	E5	659.25	-2.66938	0%
179	624.9406	D#5/Eb5	622.25	-2.69064	0%
190	588.7599	D5	587.33	-1.42987	0%
201	556.5392	C#5/Db5	554.37	-2.16918	0%
213	525.1849	C5	523.25	-1.93486	0%
226	494.9751	B4	493.88	-1.09511	0%
239	468.0518	A#4/Bb4	466.16	-1.89178	0%
254	440.4109	A4	440	-0.41093	0%
269	415.8527	G#4/Ab4	415.3	-0.5527	0%
285	392.5066	G4	392	-0.50658	0%
302	370.4118	F#4/Gb4	369.99	-0.42184	0%
339	329.9834	E4	329.63	-0.35341	0%

359	311.5999	D#4/Eb4	311.13	-0.46993	0%
380	294.3799	D4	293.66	-0.71993	0%
403	277.5791	C#4/Db4	277.18	-0.39909	0%
427	261.9775	C4	261.63	-0.34746	0%
453	246.9412	B3	246.94	-0.00123	0%
479	233.5373	A#3/Bb3	233.08	-0.45732	0%
508	220.2055	A3	220	-0.20546	0%
570	196.2533	G3	196	-0.25329	0%
604	185.2059	F#3/Gb3	185	-0.20592	0%
640	174.7881	F3	174.61	-0.17809	0%
678	164.9917	E3	164.81	-0.1817	0%
719	155.5833	D#3/Eb3	155.56	-0.02328	0%
761	146.9966	D3	146.83	-0.16655	0%
807	138.6176	C#3/Db3	138.59	-0.02757	0%
855	130.8355	C3	130.81	-0.02553	0%
906	123.4706	B2	123.47	-0.00061	0%
959	116.6469	A#2/Bb2	116.54	-0.1069	0%
1016	110.1027	A2	110	-0.10273	0%
1077	103.8666	G#2/Ab2	103.83	-0.03664	0%
1141	98.04064	G2	98	-0.04064	0%
1209	92.52636	F#2/Gb2	92.5	-0.02636	0%
1281	87.32582	F2	87.31	-0.01582	0%
1357	82.43506	E2	82.41	-0.02506	0%
1438	77.79164	D#2/Eb2	77.78	-0.01164	0%
1523	73.45002	D2	73.42	-0.03002	0%
1614	69.30878	C#2/Db2	69.3	-0.00878	0%
1710	65.41776	C2	65.41	-0.00776	0%

1811	61.7694	B1	61.74	-0.0294	0%
1919	58.29306	A#1/Bb1	58.27	-0.02306	0%
2033	55.02429	A1	55	-0.02429	0%
2154	51.93332	G#1/Ab1	51.91	-0.02332	0%
2282	49.02032	G1	49	-0.02032	0%
2418	46.26318	F#1/Gb1	46.25	-0.01318	0%
2562	43.66291	F1	43.65	-0.01291	0%
2715	41.20235	E1	41.2	-0.00235	0%
2876	38.89582	D#1/Eb1	38.89	-0.00582	0%
3047	36.71296	D1	36.71	-0.00296	0%
3228	34.65439	C#1/Db1	34.65	-0.00439	0%
3420	32.70888	C1	32.7	-0.00888	0%
3623	30.87617	B0	30.87	-0.00617	0%
3838	29.14653	A#0/Bb0	29.14	-0.00653	0%
4067	27.50538	A0	27.5	-0.00538	0%

Memory Map

Address (Decimal)	Address (Hex)	Label	Comment
0	00:0000	LAB_WARM	BASIC warm start entry point
1	00:0001	Wrmjpl	BASIC warm start vector jump low byte
2	00:0002	Wrmjph	BASIC warm start vector jump high byte
4	00:0004	TMPFLG	BASIC Temp area
6	00:0006	VIDEOMODE	BASIC Current Video Mode
7	00:0007	LOCALWORK	word (2 bytes)
10	00:000A	Usrjmp	USR function JMP address
11	00:000B	Usrjpl	USR function JMP vector low byte
12	00:000C	Usrjph	USR function JMP vector high byte
13	00:000D	Nullct	nulls output after each line
14	00:000E	TPos	BASIC terminal position byte
15	00:000F	TWidth	BASIC terminal width byte
16	00:0010	Iclim	input column limit
17	00:0011	Itempl	temporary integer low byte
18	00:0012	Itemph	temporary integer high byte
17	00:0011	nums_1	number to bin/hex string convert MSB
18	00:0012	nums_2	number to bin/hex string convert
19	00:0013	nums_3	number to bin/hex string convert LSB
91	00:005B	Srchc	search character
91	00:005B	Temp3	temp byte used in number routines
92	00:005C	Scnquo	scan-between-quotes flag
92	00:005C	Asrch	alt search character
91	00:005B	XOAw_l	eXclusive OR, OR and AND word low byte
92	00:005C	XOAw_h	eXclusive OR, OR and AND word high byte
93	00:005D	Ibptr	input buffer pointer
93	00:005D	Dimcnt	# of dimensions
93	00:005D	Tindx	token index
94	00:005E	Defdim	default DIM flag
95	00:005F	Dtypef	data type flag, $FF=string, $00=numeric
96	00:0060	Oquote	open quote flag (b7) (Flag: DATA scan
96	00:0060	Gclctd	garbage collected flag
97	00:0061	Sufnxf	subscript/FNX flag, 1xxx xxx = FN(0xxx xxx)
98	00:0062	Imode	input mode flag, $00=INPUT, $80=READ
99	00:0063	Cflag	comparison evaluation flag
100	00:0064	TabSiz	TAB step size (was input flag)
101	00:0065	next_s	next descriptor stack address these two bytes form a word pointer to the item currently on top of the descriptor stack
102	00:0066	last_sl	last descriptor stack address low byte
103	00:0067	last_sh	last descriptor stack address high byte (always $00)
104	00:0068	des_sk	descriptor stack start address (temp strings)
112	00:0070		End of descriptor stack
113	00:0071	ut1_pl	utility pointer 1 low byte

114	00:0072	ut1_ph	utility pointer 1 high byte
115	00:0073	ut2_pl	utility pointer 2 low byte
116	00:0074	ut2_ph	utility pointer 2 high byte
113	00:0071	Temp_2	temp byte for block move
117	00:0075	FACt_1	FAC temp mantissa1
118	00:0076	FACt_2	FAC temp mantissa2
119	00:0077	FACt_3	FAC temp mantissa3
118	00:0076	dims_l	array dimension size low byte
119	00:0077	dims_h	array dimension size high byte
120	00:0078	TempB	temp page 0 byte
121	00:0079	Smeml	start of mem low byte (Start-of-Basic)
122	00:007A	Smemh	start of mem high byte (Start-of-Basic)
123	00:007B	Svarl	start of vars low byte (Start-of-Variables)
124	00:007C	Svarh	start of vars high byte (Start-of-Variables)
125	00:007D	Sarryl	var mem end low byte (Start-of-Arrays)
126	00:007E	Sarryh	var mem end high byte (Start-of-Arrays)
127	00:007F	Earryl	array mem end low byte (End-of-Arrays)
128	00:0080	Earryh	array mem end high byte (End-of-Arrays)
129	00:0081	Sstorl	string storage low byte (String storage (moving down))
130	00:0082	Sstorh	string storage high byte (String storage (moving down))
131	00:0083	Sutill	string utility ptr low byte
132	00:0084	Sutilh	string utility ptr high byte
133	00:0085	Ememl	end of mem low byte (Limit-of-memory)
134	00:0086	Ememh	end of mem high byte (Limit-of-memory)
135	00:0087	Clinel	current line low byte (Basic line number)
136	00:0088	Clineh	current line high byte (Basic line number)
137	00:0089	Blinel	break line low byte (Previous Basic line number)
138	00:008A	Blineh	break line high byte (Previous Basic line number)
139	00:008B	Cpntrl	continue pointer low byte
140	00:008C	Cpntrh	continue pointer high byte
141	00:008D	Dlinel	current DATA line low byte
142	00:008E	Dlineh	current DATA line high byte
143	00:008F	Dptrl	DATA pointer low byte
144	00:0090	Dptrh	DATA pointer high byte
145	00:0091	Rdptrl	read pointer low byte
146	00:0092	Rdptrh	read pointer high byte
147	00:0093	Varnm1	current var name 1st byte
148	00:0094	Varnm2	current var name 2nd byte
149	00:0095	Cvaral	current var address low byte
150	00:0096	Cvarah	current var address high byte
151	00:0097	Frnxtl	var pointer for FOR/NEXT low byte
151	00:0097	Tidx1	temp line index
151	00:0097	Lvarpl	let var pointer low byte
152	00:0098	Frnxth	var pointer for FOR/NEXT high byte
152	00:0098	Lvarph	let var pointer high byte
153	00:0099	prstk	precedence stacked flag
155	00:009B	comp_f	compare function flag, bits 0,1 and 2 used

			bit 2 set if >
			bit 1 set if =
			bit 0 set if <
156	00:009C	func_l	function pointer low byte
157	00:009D	func_h	function pointer high byte
156	00:009C	garb_l	garbage collection working pointer low byte
157	00:009D	garb_h	garbage collection working pointer high byte
158	00:009E	des_2l	string descriptor_2 pointer low byte
159	00:009F	des_2h	string descriptor_2 pointer high byte
160	00:00A0	g_step	garbage collect step size
161	00:00A1	Fnxjmp	jump vector for functions
162	00:00A2	Fnxjpl	functions jump vector low byte
162	00:00A2	g_indx	garbage collect temp index
163	00:00A3	Fnxjph	functions jump vector high byte
163	00:00A3	FAC2_r	FAC2 rounding byte
164	00:00A4	Adatal	array data pointer low byte
164	00:00A4	Nbendl	new block end pointer low byte
165	00:00A5	Adatah	array data pointer high byte
165	00:00A5	Nbendh	new block end pointer high byte
166	00:00A6	Obendl	old block end pointer low byte
167	00:00A7	Obendh	old block end pointer high byte
168	00:00A8	numexp	string to float number exponent count
168	00:00A8	numbit	bit count for array element calculations
169	00:00A9	expcnt	string to float exponent count
170	00:00AA	numdpf	string to float decimal point flag
170	00:00AA	Astrtl	array start pointer low byte
171	00:00AB	expneg	string to float eval exponent -ve flag
171	00:00AB	Astrth	array start pointer high byte
170	00:00AA	Histrl	highest string low byte
171	00:00AB	Histrh	highest string high byte
170	00:00AA	Baslnl	BASIC search line pointer low byte
171	00:00AB	Baslnh	BASIC search line pointer high byte
170	00:00AA	Fvar_l	find/found variable pointer low byte
171	00:00AB	Fvar_h	find/found variable pointer high byte
170	00:00AA	Ostrtl	old block start pointer low byte
171	00:00AB	Ostrth	old block start pointer high byte
170	00:00AA	Vrschl	variable search pointer low byte
171	00:00AB	Vrschh	variable search pointer high byte
172	00:00AC	FAC1_e	FAC1 exponent
173	00:00AD	FAC1_1	FAC1 mantissa1
174	00:00AE	FAC1_2	FAC1 mantissa2
175	00:00AF	FAC1_3	FAC1 mantissa3
176	00:00B0	FAC1_s	FAC1 sign (b7)
172	00:00AC	str_ln	string length
173	00:00AD	str_pl	string pointer low byte
174	00:00AE	str_ph	string pointer high byte
174	00:00AE	des_pl	string descriptor pointer low byte

175	00:00AF	des_ph	string descriptor pointer high byte
175	00:00AF	mids_l	MID$ string temp length byte
177	00:00B1	negnum	string to float eval -ve flag
177	00:00B1	numcon	series evaluation constant count
178	00:00B2	FAC1_o	FAC1 overflow byte
179	00:00B3	FAC2_e	FAC2 exponent
180	00:00B4	FAC2_1	FAC2 mantissa1
181	00:00B5	FAC2_2	FAC2 mantissa2
182	00:00B6	FAC2_3	FAC2 mantissa3
183	00:00B7	FAC2_s	FAC2 sign (b7)
184	00:00B8	FAC_sc	FAC sign comparison, Acc#1 vs #2
185	00:00B9	FAC1_r	FAC1 rounding byte
184	00:00B8	ssptr_l	string start pointer low byte
185	00:00B9	ssptr_h	string start pointer high byte
184	00:00B8	sdescr	string descriptor pointer
186	00:00BA	csidx	line crunch save index
186	00:00BA	Asptl	array size/pointer low byte
187	00:00BB	Aspth	array size/pointer high byte
186	00:00BA	Btmpl	BASIC pointer temp low byte
187	00:00BB	Btmph	BASIC pointer temp low byte
186	00:00BA	Cptrl	BASIC pointer temp low byte
187	00:00BB	Cptrh	BASIC pointer temp low byte
186	00:00BA	Sendl	BASIC pointer temp low byte
187	00:00BB	Sendh	BASIC pointer temp low byte
188	00:00BC	LAB_IGBY	get next BASIC byte subroutine
194	00:00C2	LAB_GBYT	get current BASIC byte subroutine
195	00:00C3	Bpntrl	BASIC execute (get byte) pointer low byte
196	00:00C4	Bpntrh	BASIC execute (get byte) pointer high byte
197	00:00C5	Bpntrp	BASIC execute (get byte) pointer PAGE byte
224	00:00E0		end of get BASIC char subroutine
225	00:00E1	Rbyte4	extra PRNG byte
226	00:00E2	Rbyte1	most significant PRNG byte
227	00:00E3	Rbyte2	middle PRNG byte
228	00:00E4	Rbyte3	least significant PRNG byte
229	00:00E5	NmiBase	NMI handler enabled/setup/triggered flags bit function === ======== 7 interrupt enabled 6 interrupt setup 5 interrupt happened
230	00:00E6		NMI handler addr low byte
231	00:00E7		NMI handler addr high byte
232	00:00E8	IrqBase	IRQ handler enabled/setup/triggered flags
233	00:00E9		IRQ handler addr low byte
234	00:00EA		IRQ handler addr high byte
235	00:00EB	FCBPTR	POINTER TO FCB FOR FILE OPS
239	00:00EF	Decss	number to decimal string start

240	00:00F0	Decssp1	number to decimal string start
253	00:00FD	TEMPW	
255	00:00FF		decimal string end
256-511	00:0100-00:01FF	6502Stack	Left Free for 6502 Emulation Mode Stack
512-767	00:0200-00:02FF	KEYBUFF	256 BYTE KEYBOARD BUFFER
			NATIVE VECTORS
768	00:0300	ICOPVECTOR	COP handler indirect vector...
770	00:0302	IBRKVECTOR	BRK handler indirect vector...
772	00:0304	IABTVECTOR	ABT handler indirect vector...
774	00:0306	INMIVECTOR	NMI handler indirect vector...
776	00:0308	IIRQVECTOR	IRQ handler indirect vector...
			6502 Emulation Vectors
778	00:030A	IECOPVECTOR	ECOP handler indirect vector...
780	00:030C	IEABTVECTOR	EABT handler indirect vector...
782	00:030E	IENMIVECTOR	ENMI handler indirect vector...
784	00:0310	IEINTVECTOR	EINT handler indirect vector...
			IEC Driver work Area
786	00:0312	IECDCF	Serial output: deferred char flag
787	00:0313	IECDC	Serial deferred character
788	00:0314	IECBCI	Serial bit count/EOI flag
789	00:0315	IECBTC	Countdown, bit count
790	00:0316	IECCYC	Cycle count
791	00:0317	IECSTW	Status word
792	00:0318	IECFNLN	File Name Length
793	00:0319	IECSECAD	IEC Secondary Address
794	00:031A	IECBUFFL	low byte IEC buffer Pointer
795	00:031B	IECBUFFH	High byte IEC buffer Pointer
796	00:031C	IECDEVN	IEC Device Number
797	00:031D	IECSTRTL	low byte IEC Start Address Pointer
798	00:031E	IECSTRTH	High byte IEC Start Address Pointer
799	00:031F	IECMSGM	message mode flag, $C0 = both control and kernal messages, $80 = control messages only, $40 = kernal messages only, $00 = neither control or kernal messages
800	00:0320	IECFNPL	File Name Pointer Low,
801	00:0321	IECFNPH	File Name Pointer High,
802	00:0322	LOADBUFL	low byte IEC buffer Pointer
803	00:0323	LOADBUFH	High byte IEC buffer Pointer
804	00:0324	LOADBANK	BANK buffer Pointer
805	00:0325	IECOPENF	OPEN FILE COUNT
806	00:0326	IECLFN	IEC LOGICAL FILE NUMBER
807	00:0327	IECIDN	input device number
808	00:0328	IECODN	output device number
			VIDEO/KEYBOARD PARAMETER WORK AREA

816	00:0330	CSRX	CURRENT X POSITION
817	00:0331	CSRY	CURRENT Y POSITION
818	00:0332	LEDS	Current Keyboard LED status
819	00:0333	KeyLock	Current Keylock status
820	00:0334	ScannedKey	Current Scanned Key
821	00:0335	ScrollCount	Current Scroll Count
822	00:0336	TEMP	TEMP AREA
833	00:0341	ConsoleDevice	Current Console Device
			$00 Serial, $01 On-Board vga/KB
834	00:0342	CSRCHAR	Character under the Cursor
835	00:0343	VIDEOWIDTH	SCREEN WIDTH -- 32 or 40 (80 in the future)
848	00:0344	Default Color	
849	00:0345-6	Temp storage	Used for calculating video offsets
851	00:0347-8	Temp storage	Used for calculating video offsets
944	00:03B0	PTRLFT	.. to $03B9 logical file table
954	00:03BA	PTRDNT	.. to $03C3 device number table
964	00:03C4	PTRSAT	.. to $03CD secondary address table
1001-4095	00:03F0-00:0FFF		OPEN
4096-8191	00:1000-00:19FF		Text Page 1
8192-24575	00:2000-00:5FFF		Text Page 2, Graphics Page 1
24576-40959	00:6000-00:9FFF		Graphics Page 2
45055	00:BFFF	STACK	TOP OF 65816 STACK
45056-65535	00:C000-00:FFFF	ROM	
			IO Area
65024	00:FE00	UART0:	DATA IN/OUT
65025	00:FE01	UART1:	CHECK RX
65026	00:FE02	UART2:	INTERRUPTS
65027	00:FE03	UART3:	LINE CONTROL
65028	00:FE04	UART4:	MODEM CONTROL
65029	00:FE05	UART5:	LINE STATUS
65030	00:FE06	UART6:	MODEM STATUS
65032	00:FE08		RTC Address Register
65033	00:FE09		RTC Date Register
65040	00:FE10	via1regb	Register (Sound Data)
65041	00:FE11	via1rega	Register (IEC,RTC, & Sound)
			B0 - Serial Clock in
			B1- Serial Data in
			B2- Sound BDIR
			B3- Sound BC1

			B4- Sound BC2
			B5- RTC DQ
			B6- RTC RST
			B7 - Serial ATN Out
65042	00:FE12	via1ddrb	Register
65043	00:FE13	via1ddra	Register
65044	00:FE14	via1t1cl	Register
65045	00:FE15	via1t1ch	Register
65046	00:FE16	via1t1ll	Register
65047	00:FE17	via1t1lh	Register
65048	00:FE18	via1t2cl	Register
65049	00:FE19	via1t2ch	Register
65050	00:FE1A	via1sr	Register
65051	00:FE1B	via1acr	Register
65052	00:FE1C	via1pcr	Register
65053	00:FE1D	via1ifr	Register
65054	00:FE1E	via1ier	Register
65055	00:FE1F	via1ora	Register
65056	00:FE20	via2regb	Register (KB ROW & LED control)
65057	00:FE21	via2rega	Register (KB Column)
65058	00:FE22	via2ddrb	Register
65059	00:FE23	via2ddra	Register
65060	00:FE24	via2t1cl	Register
65061	00:FE25	via2t1ch	Register
65062	00:FE26	via2t1ll	Register
65063	00:FE27	via2t1lh	Register
65064	00:FE28	via2t2cl	Register
65065	00:FE29	via2t2ch	Register
65066	00:FE2A	via2sr	Register
65067	00:FE2B	via2acr	Register
65068	00:FE2C	via2pcr	Register
65069	00:FE2D	via2ifr	Register
65070	00:FE2E	via2ier	Register
65071	00:FE2F	via2ora	Register
65072	00:FE30	VideoScanlines	01=ON, 02=OFF
65073	00:FE31	VideoDisplayPage	01=Page 1, 2=Page2
65074	00:FE32	VideoCharGenOffset	Character Generator write offset (character)
65075	00:FE33	VideoCharGenData	Character Generator write Data
65076	00:FE34	VideoTextMode	01=ON, 02=OFF
65077	00:FE35	VideoCommandReg	
65078	00:FE36	VideoLoresMode	01=ON, 02=OFF
65079	00:FE37	VideoDoubleLores	01=ON, 02=OFF
65070	00:FE38	VideoHiresMode	01=ON, 02=OFF
65071	00:FE39	VideoDoubleHires	01=ON, 02=OFF
65072	00:FE3A	Video80col	01=80 col, 02=40 col

65073	00:FE3B	VideoMixedMode	01=ON, 02=OFF
65074	00:FE3C	VideoQuadHires	01=ON, 02=OFF
65075	00:FE3D	VideoMonoHires	01=ON, 02=OFF
	01:0000-07:FFFF	RAM	Basic uses 02:0000-02:FFFF for storage
	FF:0000-FF:FFFF	BASIC ROM	
			BIOS Jump Table (Native Long JSR)
64768	00:FD00	LPRINTVEC	Print a character to the active console
64772	00:FD04	LINPVEC	Get a character from the active console (no wait)
64776	00:FD08	LINPWVEC	Get a character from the active console (wait)
64780	00:FD0C	LSetXYVEC	Set the Cursor position on the screen
64784	00:FD10		
64788	00:FD14	LSrlUpVEC	Scroll up the Screen
64792	00:FD18	LSetColorVEC	Set the Color for the Screen
64796	00:FD1C	LCURSORVEC	Turn on the Cursor
64800	00:FD20	LUNCURSORVEC	Turn Off the Cursor
64804	00:FD24	LWRITERTC	Write a RTC register
64808	00:FD28	LREADRTC	Read a RTC register
64812	00:FD2C	LIECIN	Read byte from serial bus. (Must call TALK and TALKSA beforeh
64816	00:FD30	LIECOUT	Write byte to serial bus. (Must call LISTEN and LSTNSA beforeh
64820	00:FD34	LUNTALK	Send UNTALK command to serial bus.
64824	00:FD38	LUNLSTN	Send UNLISTEN command to serial bus.
64828	00:FD3C	LLISTEN	Send LISTEN command to serial bus.
64832	00:FD40	LTALK	Send TALK command to serial bus.
64836	00:FD44	LSETLFS	Set file parameters.
64840	00:FD48	LSETNAM	Set file name parameters.
64844	00:FD4C	LLOAD	Load or verify file. (Must call SETLFS and SETNAM beforehand.
64848	00:FD50	LSAVE	Save file. (Must call SETLFS and SETNAM beforehand.)
64852	00:FD54	LIECINIT	INIT IEC
64856	00:FD58	LIECCLCH	close input and output channels
64860	00:FD5C	LIECOUTC	open a channel for output
64864	00:FD60	LIECINPC	open a channel for input
64868	00:FD64	LIECOPNLF	open a logical file
64872	00:FD68	LIECCLSLF	close a specified logical file
64876	00:FD6C	LClearScrVec	clear the Screen
64880	00:FD70		
			BIOS Jump Table (Emulation, short JSR)
65393	00:FF71	PRINTVEC	Print a character to the active console
65396	00:FF74	INPVEC	Get a character from the active console (no wait)
65399	00:FF77	INPWVEC	Get a character from the active console (wait)
65402	00:FF7A	SetXYVEC	Set the Cursor position on the screen

65405	00:FF7D		
65408	00:FF80	SrlUpVEC	Scroll up the Screen
65411	00:FF83	SetColorVEC	Set the Color for the Screen
65414	00:FF86	CURSORVEC	Turn on the Cursor
65417	00:FF89	UNCURSORVEC	Turn Off theCursor
65420	00:FF8C	WRITERTC	Write a RTC register
65423	00:FF8F	READRTC	Read a RTC register
65426	00:FF92	IECIN	Read byte from serial bus. (Must call TALK and TALKSA beforeh
65429	00:FF95	IECOUT	Write byte to serial bus. (Must call LISTEN and LSTNSA beforeh
65432	00:FF98	UNTALK	Send UNTALK command to serial bus.
65435	00:FF9B	UNLSTN	Send UNLISTEN command to serial bus.
65438	00:FF9E	LISTEN	Send LISTEN command to serial bus.
65441	00:FFA1	TALK	Send TALK command to serial bus.
65444	00:FFA4	SETLFS	Set file parameters.
65447	00:FFA7	SETNAM	Set file name parameters.
65450	00:FFAA	LOAD	Load or verify file. (Must call SETLFS and SETNAM beforehand.
65453	00:FFAD	SAVE	Save file. (Must call SETLFS and SETNAM beforehand.)
65456	00:FFB0	IECINIT	INIT IEC
65459	00:FFB3	IECCLCH	close input and output channels
65462	00:FFB6	IECOUTC	open a channel for output
65465	00:FFB9	IECINPC	open a channel for input
65468	00:FFBC	IECOPNLF	open a logical file
65471	00:FFBF	IECCLSLF	close a specified logical file
65474	00:FFC2	ClearScrVec	clear the Screen
65477	00:FFC5		