FASTBASIC v4.0

for Atari 8-Bit Computers

by Daniel Serpell

Native Atari Version

Contents

1 Introduction

`FastBasic` is a fast interpreter for the BASIC language on the Atari 8-bit computers.

One big difference from other BASIC interpreters in the old 8-bit computers is the lack of line numbers and the integrated full-screen editor. This is similar to newer programming environments, giving the programmer a lot of flexibility.

Another big difference is that default variables and operations are done using integer numbers, this is one of the reasons that the programs run so fast. The other reason is that the program is parsed on run, generating optimized code for very fast execution.

Currently, FastBasic support:

- Integer and floating point variables, including all standard arithmetic operators.
- All graphic, sound and color commands from Atari Basic, plus some extensions from Turbo Basic.
- All control flow structures from Atari Basic and Turbo Basic.
- Automatic string variables of up to 255 characters.
- Arrays of “word”, “byte” and strings.
- User defined procedures.
- Compilation to binary loadable files.
- Available as a full version `FB.COM`, as a smaller integer-only `FBI.COM` and as a command-line compiler `FBC.COM`.

2 First Steps

To run FastBasic from the included disk image, simply type `FB` at the DOS prompt. This will load the IDE and present you with a little help text:

```
--D:HELP.TXT-------------------0--
'  FastBasic v4.0 -  (c) 2019 dmsc
'
' Editor Help
' -----------
'  Ctrl-A : Move to beg of line
'  Ctrl-E : Move to end of line
'  Ctrl-U : Page up
'  Ctrl-V : Page down
'  Ctrl-Z : Undo (only curr line)
'  Ctrl-Q : Exit to DOS
'  Ctrl-S : Save file
'  Ctrl-L : Load file
'  Ctrl-N : New file
'  Ctrl-R : Parse and run program
'  Ctrl-W : Compile to binary file
'
'- Press CONTROL-N to begin -
```

You are now in the integrated editor. In the first line of the screen the name of the currently edited file is shown, and at the right the line of the cursor. Note that lines that show an arrow pointing to the top-left are empty lines beyond the last line of the current file.

In this example, the cursor is in the first column of the first line of the file being edited.

As the help text says, just press the `CONTROL` key and the letter `N` (at the same time) to begin editing a new file. If the text was changed, the editor asks if you want to save the current file to disk, to skip saving simply type `CONTROL-C`, to cancel the New File command type `ESC`.

Now, you can start writing your BASIC program. Try the following example, pressing `RETURN` after each line to advance to the next:

```
1 INPUT "WHAT IS YOUR NAME?";NAME$
2 ?
3 ? "HELLO", NAME$
```

If you made any mistake, simply move back using the cursor keys, this is `CONTROL` and `-`, `=`, `+` or `*`, then press `BACKSPACE` key to delete the character before the cursor or press `DELETE` (`CONTROL` and `BACKSPACE`) to delete the character bellow the cursor. To join two lines, go to the end of the first line and press `DELETE`.

After typing the last line, you can run the program by pressing `CONTROL` and `R`.

If you correctly typed your program, the computer will ask you for your name, type anything you want and press `RETURN`

After this, the IDE waits for any key press to return to the editor, so you have a chance to see all your program output.

If you made a mistake in the program, instead of running the cursor will be moved to the line and column of the error so you can correct it and retry.

Remember to save often, pressing the `CONTROL` and `S` keys, you need to supply a filename. Type the name, and press `ENTER` to save. As with any prompt, you can press `ESC` to cancel the save operation. Use the `BACKSPACE` over the proposed file name if you want to change it.

3 Compiling the program to disk

Once you are satisfied with your program, you can compile to a disk file, producing a program that can be run directly from DOS.

Press the `CONTROL` and `W` key, and type a name for your compiled program. It is common practice to name your compiled programs with an extension of ".COM" or ".XEX", also with ".COM" files you don't need to type the extension in some DOS.

Compiled programs include the full runtime, so you can distribute them alone without the IDE.

You can also compile a program directly from DOS by using the included command line compiler `FBC.COM`. The compiler prompts for the input file name, loads the BASIC source, compiles it and prompts for a executable output file name to write the compiled program.

4 About the syntax

The syntax of FastBasic language is similar to many BASIC dialects, with the following main rules:

1. A program line must be of 4 types:
    - a comment line, starting with a dot `.` or an apostrophe `'`,
    - a statement followed by its parameters,
    - a variable assignment, this is a name followed by `=` and a the expression for the new value. For string variables, there is also a concatenation operator, `=+`.
    - an empty line.
2. All statements and variable names can be lower or uppercase, the language is case insensitive.
3. Statements can be abbreviated to reduce typing, each statement have a different abbreviation.
4. Multiple statements can be put on the same line by placing a colon `:` between statements.
5. After any statement a comment can be included by starting it with an apostrophe `'`.
6. No line numbers are allowed.
7. Spaces after statements and between operators are optional and ignored.

In the following chapters, whenever a value can take any numeric expression, it is written as "*value*", and whenever you can use a string it is written as "*text*".

5 Expressions

Expressions are used to perform calculations in the language.

There are numeric expressions (integer and floating point), boolean expressions and string expressions.

In FastBasic, standard numeric expressions are evaluated as integers from -32768 to 32767, this is called 16 bit signed integer values.

Floating point expressions are used *only* if numbers have a decimal point. Floating point numbers are stored with standard Atari BCD representation, with a range from 1E-98 to 1E+98.

Boolean expressions are “true” or “false”, represented as the numbers 1 and 0.

String expressions contain arbitrary text, in FastBasic strings can have up to 255 characters of length.

5.1 Numeric Values

Basic values can be written as decimal numbers (like `123`, `1000`, etc.), as hexadecimal numbers with a $ sign before (like `$1C0`, `$A00`, etc.) or by using the name of a variable.

Floating point values are written with a decimal dot and an optional exponent (like `1.0E+10`, or `-3.2`)

5.2 Numeric Variables

Variable names must begin with a letter or the symbol `_`, and can contain any letter, number or the symbol `_`. Examples of valid variable names are `COUNTER`, `My_Var`, `num1`.

Floating point variables have an `%` as last character in the name. Examples of valid names are `MyNum%`, `x1%`.

5.3 Numeric Operators

There are various “operators” the perform calculation in expressions, the operators with higher precedence always execute first. These are the *integer* operators in order of precedence:

- `+` `-` : addition, subtraction, from left to right.
- `*` `/` `MOD` : multiplication, division, modulus, from left to right.
- `&` `!` `EXOR` : binary AND, OR and EXOR, from left to right.
- `+` `-` : positive / negative.

For example, an expression like `1 + 2 * 3 - 4 * -5` is evaluated in the following order:

- First, the unary `-` before the `5`, giving the number `-5`.
- The first multiplication, giving `2*3` = `6`.
- The second multiplication, giving `4*-5` = `-20`.
- The addition, giving `1+6` = `7`.
- The subtraction, giving `7 - -20` = `27`.

So, in this example the result is 27.

If there is need to alter the precedence, you can put the expression between parenthesis.

When using floating point expressions, the operators are:

- `+` `-` : addition, subtraction, from left to right.
- `*` `/` : multiplication, division, from left to right.
- `^` : exponentiation, from left to right.
- `+` `-` : positive / negative.

Note that integer expressions are automatically converted to floating point if needed, this allows mixing integers and floating point in some calculations, but you must have care to force floating point calculations to avoid integer overflows.

Example: the expression

```
a% = 1000 * 1000 + 1.2
```

gives correct result as 1000 is converted to floating point before calculation, but:

```
x=1000: a% = x * x + 1.2
```

gives incorrect results as the multiplication result is bigger tan 32767.

Note that after any floating point errors (division by 0 and overflow), `ERR()` returns 3.

5.4 Boolean Operators

Boolean operators return a “true” or “false” value instead of a numeric value, useful as conditions in loops and `IF` statements.

Note that any boolean value can also be used as a numeric value, in this case, “true” is converted to 1 and “false” converted to 0.

The supported boolean operators, in order of precedence, are:

- `OR` : Logical OR, true if one or both operands are true.
- `AND` : Logical AND, true only if both operands are true.
- `NOT` : Logical NOT, true only if operand is false.
- `<=` `>=` `<>` `<` `>` `=` Integer or floating point comparison, compare the two numbers and return true or false. Note that `<>` is *not equal*. You can only compare two values of the same type, so an expression like `x = 1.2` is invalid, but `1.2 = x` is valid as the second operand is converted to floating point before comparison.

5.5 Arrays

Arrays hold many ordered values (called elements), the elements can be accessed by an index.

In FastBasic, arrays must be dimensioned before use (see `DIM` statement bellow), the index of the element is written between parenthesis and goes from 0 to the number of elements less 1.

You can use an array position (the variable name followed by the index) in any location where a standard numeric variable or value is expected.

Arrays can be of three types:

- `WORD` arrays (the default if no type is given) use two bytes of memory for each element, and works like normal numeric variables.
- `BYTE` arrays use only one byte for each element, but the numeric range is reduced from 0 to 255.
- String arrays store an string in each element. String arrays use two byte of memory for each element that is not yet assigned (containing empty strings), and 258 bytes for each element with a string assigned.

5.6 String Values

String values are written as a text surrounded by double quotes (`"`). If you need to include a double quote character in a string, you must write two double quotes together.

Example:

```
1 PRINT "Hello ""world"""
```

Will print:

```
1 Hello "world"
```

The bracket operator `[ ]` allows creating a string from a portion of another, and accepts two forms:

- [*num*] This form selects all characters from *num* up to the end of the string, counting from 1. So, `A$[1]` selects all the string and `A$[3]` selects from the third character to the end, effectively removing the leftmost two characters.
- [*num1* , *num2*] This form selects at most *num2* characters from *num1*, or up to the end of the string if there is not enough characters.

Example:

```
PRINT "Hello World"[7]
A$ = STR$(3.1415)[3,3]
? A$
? A$[2,1]
```

Will print:

```
World
141
4
```

Note that the bracket operator creates a new string and copies the characters from the original string to the new one. As the buffer used for the new string is always the same, you can't compare two values without first assigning them to a new variable.

This will print "ERROR":

```
A$="Dont Work"
IF A$[2,2] = A$[3,3] THEN ? "ERROR"
```

But this will work:

```
A$="Long string"
B$=A$[2,2]
IF B$ = A$[3,3] THEN ? "ERROR"
```

5.7 String Variables

String variables are named the same as numeric variables but must end with a `$` symbol. Valid variable names are `Text$`, `NAME1$`.

String variables always use 256 bytes, the first byte stores the string length and the following bytes store up to 255 characters.

There are two types of string assignments:

- The standard `=` sign copies the string expression in the right to the variable in the left.
- The `=+` sign copies the string expression at the right to the end of the current string, concatenating the text.

Example:

```
A$ = "Hello "
A$ =+ "World"
? A$
```

Will print:

```
Hello World
```

5.8 Standard Functions

Functions take parameters between parenthesis and produce a result. Following is a list of all the functions supported by FastBasic.

- TIME : Returns the curren time in "jiffies". This is about 60 times per second in NTSC systems, 50 times per second in PAL systems.
- ABS(*num*) : Returns the absolute value of *num*. Can be used with integers and floating point.
- SGN(*num*) ; Returns the sign of *num*, this is 1 if positive, -1 if negative or 0 if *num* is 0. Can be used with integers and floating point.
- RAND(*num*) : Returns a random, non negative number, less than *num*.
- FRE() : Returns the free memory available in bytes.
- ERR() : Returns the last Input/Output error value, or 1 if no error was registered.
- LEN(*string*) : Returns the length of the *string*.
- VAL(*string*) : Convert *string* to a number. If no conversion is possible, ERR() is set to 18. Can be used with integers and floatign point.
- ASC(*string*) : Returns the ATASCI code of the first character of the *string*.

5.9 Atari Specific Functions

The following functions allows interacting to the Atari hardware to read controller and keyboard input and to program with Player/Missile graphics.

- PADDLE(*n*): Returns the value of the PADDLE controller *n*.
- PMADR(*n*): Returns the address of the data for Player *n* or the address of the Missiles with *n* = -1.
- PTRIG(*n*) : Returns 0 if the PADDLE controller *n* button is pressed, 1 otherwise.
- STICK(*n*) : Returns the JOYSTICK controller *n* position, this is 15 if centered. See Atari Basic manual for all possible values.
- STRIG(*n*) : Returns 0 if JOYSTICK controller *n* button is pressed, 1 otherwise.
- KEY() : Returns 0 if no key was pressed, or a keycode. The returned value only goes to 0 after reading the key in the OS (via a `GET` or `POKE 764,0` statement). *Hint: The value returned is actually the same as* `(PEEK(764)EXOR 255)` .

5.10 Floating Point functions

This functions use floating point values, and are only available in the floating point version.

In case of errors (such as logarithm or square root of negative numbers and overflow in the results), the functions returns an invalid value, and the ERR() function returns 3.

- ATN(*n*): Arc-Tangent of *n*.
- COS(*n*): Cosine of *n*.
- EXP(*n*) : Natural exponentiation.
- EXP10(*n*) : Returns ten raised to *n*.
- INT(*num*) : Converts the floating point number *num* to the nearest integer from -32768 to 32767.
- LOG(*n*) : Natural logarithm of *n*.
- LOG10(*n*): Decimal logarithm of *n*.
- RND(): Returns a random positive number strictly less than 1.
- SIN(*n*): Sine of *n*.
- SQR(*n*): Square root of *n*.

5.11 String functions

- STR$(*num*): Returns a string with a printable value for *num*. Can be used with integers and floating point. Note that this function can't be used at both sides of a comparison, as the resulting string is overwritten each time it is called.
- CHR$(*num*): Converts *num* to a one character string with the ATASCII value.

5.12 Low level Functions

The following functions are called "low level" because they interact directly with the hardware. Use with care!.

- ADR(*arr*): Returns the address of the first element of *arr* in memory. Following elements of the array occupy adjacent memory locations.
- ADR(*str*): Returns the address of the *string* in memory. The first memory location contains the length of the string, and following locations contain the string characters.
- DPEEK(*addr*): Returns the value of memory location *addr* and *addr*+1 as a 16 bit integer.
- PEEK(*address*): Returns the value of memory location at *address*.
- USR(*address*[,*num1* ...]): Low level function, calls the user supplied machine code subroutine at *address*.

  Parameters are pushed to the CPU stack, with the LOW part pushed first, so the first PLA returns the HIGH part of the last parameter, and so on.

  The value of the A and X registers is used as a return value of the function, with A the low part and X the high part.

  This is a sample usage:

```
1 ' PLA / EOR $FF / TAX / PLA / EOR $FF / RTS
2 DATA ml() byte = $68,$49,$FF,$AA,$68,$49,$FF,$60
3 FOR i=0 TO 1000 STEP 100
4   ? i, USR(ADR(ml),i)
5 NEXT i
```

6 List of statements

In the following descriptions, the statement usage is presented and the abbreviation is given after a /.

6.1 Console Print and Input Statements

Reads key from Keyboard
GET *var* | GE.

Waits for a key-press and writes the key value to *var*, which can be a variable name or an array position (like “array(123)”)

Input variable or string
INPUT *var* | I.
INPUT “prompt”; *var*
INPUT “prompt”, *var*

Reads from keyboard/screen and stores the value in *var*.

A “?” sign is printed to the screen before input, or the “prompt” if given. Also, if there is a comma after the prompt, spaces are printed to align to a column multiple of 10 (similar to how a comma works in PRINT).

If the value can’t be read because input errors, the error is stored in ERR variable. Valid errors are 128 if BREAK key is pressed and 136 if CONTROL-3 is pressed.

In case of a numeric variable, if the value can’t be converted to a number, the value 18 is stored in ERR().

Moves the screen cursor
POSITION *row*, *column* | POS.

Moves the screen cursor position to the given *row* and *column*, so the next PRINT statement outputs at that position.

Rows and columns are numerated from 0.

Print strings and numbers
PRINT *expr*, . . . | ?

Outputs strings and numbers to the screen.

Each *expr* can be a constant string, a string variable or any complex expression, with commas or semicolons between each expression.

After writing the last expression, the cursor advanced to a new line, except if the statement ends in a comma or a semicolon, where the cursor stays in the last position.

If there is a comma before any expression, the column is advanced to the next multiple of 10, so that tabulated data can be printed.

Writes a character to the screen
PUT *num* | PU.

Outputs one character to the screen, given by it's ATASCI code.

Clears the screen
CLS

Clears the text screen. This is the same as `PUT 125`. For clearing the graphics screen, you can use `CLS #6`.

6.2 Control Statements

Endless loops
DO
LOOP / L.

Starts and ends an endless repetition, when reaching the LOOP statement the program begins again executing from the DO statement.

The only way to terminate the loop is via de EXIT statement.

Calls a subroutine
EXEC *name* / EXE.

Calls the subroutine *name*. Note that the subroutine must be defined with PROC, but can be defined before or after the call.

Exits from loop or PROC
EXIT / EX.

Exits current loop or subroutine, by jumping to the end.

In case of loops, the program continues after the last statement of the loop, in case of PROC, the program returns to the calling EXEC.

Loop over values of a variable
FOR *var*=*value* TO *end* [STEP *step*] / F. TO S.
NEXT *var* / N.

FOR loop allows performing a loop a specified number of times while keeping a counting variable.

First, assigns the *value* to *var*, and starts iterations. *var* can be any variable name or a word array position (like "array(2)").

In each iteration, first compares the value of *var* with *end*, if the value is past the end, terminates the loop.

At the end of the loop, *var* is incremented by *step* (or 1 if STEP is omitted) and the loops repeats.

An EXIT statement also terminates the loop and skips to the end.

Note that if *step* is positive, iteration ends if value of *var* is bigger than *end*, but if *step* is negative, iteration ends if value of *var* is less than *end*.

Also, *end* and *step* are evaluated only once at beginning of the loop, that value is stored and used for all loop iterations.

If at the start of the loop *value* is already past *end*, the loop is completely skipped.

As an extension, you can left out the variable name in NEXT, and currently the variable name is actually ignored.

Conditional execution
IF *condition* THEN *statement* / I. T.
IF *condition*
ELIF *condition* / ELI.
ELSE / E.
ENDIF / END.

The first form (with THEN) executes one *statement* if the condition is true.

The second form executes all statements following the IF (up until any of ELIF, ELSE, ENDIF) only if condition is true.

If condition is false, optional statements following the ELSE (until an ENDIF) are executed.

In case of an ELIF, the new condition is tested and acts like a nested IF until an ELSE or ENDIF.

This is an example of a multiple IF/ELIF/ELSE/ENDIF statement:

```
IF _condition-1_
  ' Statements executed if
  ' _condition-1_ is true
ELIF _condition-2_
  ' Statements executed if
  ' _condition-1_ is false but
  ' _condition-2_ is true
ELIF _condition-3_
  ' Also, if _condition-1_ and
  ' _condition-2_ are false but
  ' _condition-3_ is true
ELSE
  ' Executed if all of the above
  ' are false
ENDIF
```

Define a subroutine.
PROC *name* / PRO.
ENDPROC / ENDP.

PROC statement starts definition of a subroutine, that can be called via EXEC.

Note that if the PROC statement is encountered while executing surrounding code, the full subroutine is skipped, so PROC / ENDPROC can appear any place in the program.

Loop until condition is true
REPEAT / R.
UNTIL *condition* / U.

The REPEAT loop allows looping with a condition evaluated at the end of each iteration.

Executes statements between REPEAT and UNTIL once, then evaluates the *condition*. If false, the loop is executed again, if true the loop ends.

An EXIT statement also terminates the loop and skips to the end.

Loop while condition is true
WHILE *condition* / W.
WEND / WE.

The `WHILE` loop allows looping with a condition evaluated at the beginning of each iteration.

First, evaluates the condition. If false, skips the whole loop to the end. If true, executes the statements between `WHILE` and `WEND` and returns to the top to test the condition again.

An EXIT statement also terminates the loop and skips to the end.

6.3 Graphic and Sound Statements

Set color number
COLOR *num* / C.

Changes the color of `PLOT` , `DRAWTO` and the line color on `FILLTO` to *num*.

Draws a line
DRAWTO *x*, *y* / DR.

Draws a line from last position to the given *x* and *y* positions.

Sets fill color number
FCOLOR *num* & FC.

Changes the filling color of `FILLTO` operation to *num*.

Fill from line to the right
FILLTO *x*, *y* / FI.

Draws a line from last position to the given *x* and *y* position, using `COLOR` number, and for each plotted point also paint all points to the right with the `FCOLOR` number until a point with different color than the first.

Sets graphic mode
GRAPHICS *num* / G.

Sets the graphics mode for graphics operations. See Atari Basic manual for a list of graphics modes, sizes and number of colors.

Plots a single point
PLOT *x*, *y* / PL.

Plots a point in the specified *x* and *y* coordinates, with the current `COLOR` number.

Player/Missile graphic mode
PMGRAPHICS *num* / PM.

Set ups Atari Player / Missile graphics. A value of 0 disables all player and missiles, a value of 1 set ups for single line resolution, a value of 2 set ups for double line resolution.

Single line uses 256 bytes per player, double line uses 128 bytes per player.

For retrieving the memory address of the player or missile data use the `PMADR()` function.

Player/Missile horizontal move
PMHPOS *num*,*pos* / PMH.

Set horizontal position register for the player or missile *num* to *pos*.

Players 0 to 3 correspond to values 0 to 3 of *num*, missiles 0 to 3 correspond to the values 4 to 7 respectively.

This is the same as writing: `POKE $D000 + num , pos`

Sets displayed color
SETCOLOR *num*, *hue*, *lum* / SE.

Alters the color registers so that color number *num* has the given *hue* and *luminance*.

To set Player/Missile colors use negative values of *num*, -4 for player 0, -3 for player 1, -2 for player 2 and -1 for player 3.

Adjust voice sound parameters
SOUND *voice*, *pitch*, *dist*, *vol* / S.

SOUND *voice*
SOUND

Adjust sound parameters for *voice* (from 0 to 3) ot the given *pitch*, *distortion* and *volume*.

If only the *voice* parameter is present, that voice is cleared so no sound is produced by that voice.

If no parameters are given, clears all voices so that no sounds are produced.

6.4 Device Input and Output Statements

Binary read from file
BGET #*iochn*,*address*,*len* | BG.

Reads *length* bytes from the channel *iochn* and writes the bytes to *address*.

For example, to read to a byte array, use `ADR(array)` to specify the address.

On any error, `ERR()` will hold an error code, on success `ERR()` reads 1.

Binary read from file
BPUT #*iochn*,*address*,*len* | BP.

Similar to `BPUT`, but writes *length* bytes from memory at *address* to the channel *iochn*.

On any error, `ERR()` will hold an error code, on success `ERR()` reads 1.

Close channel
CLOSE #*iochn* | CL.

Closes the input output channel *iochn*, finalizing all read/write operations.

On any error, `ERR()` will hold an error code, on success `ERR()` reads 1.

Note that it is important to read the value of `ERR()` after close to ensure that written data is really on disk.

Reads bytes from file
GET #*iochn*, *var*, ...

Reads one byte from channel *iochn* and writes the value to *var*.

var can be a variable name or an array position (like `array(123)`)

In case of any error, `ERR()` returns the error value.

Input variable or string from file
INPUT #*iochn*, *var* | IN.

Reads a line from channel *iochn* and stores to *var*.

If *var* is a string variable, the full line is stored.

If *var* is a numeric variable, the line is converted to a number first.

On any error, `ERR()` will hold an error code, on success `ERR()` reads 1.

Opens I/O channel
OPEN #*ioc*,*mode*,*ax*,*dev* | O.

Opens I/O channel *ioc* with *mode*, *aux*, over device *dev*.

To open a disk file for writing, *mode* should be 8, *aux* 0 and *dev* the file name as "D:name.ext".

To open a disk file for reading, *mode* should be 4, *aux* 0 and *dev* the file name as "D:name.ext".

See Atari Basic manual for more documentation in the open modes, aux values and device names.

On any error, `ERR()` will hold an error code, on success `ERR()` reads 1.

Print strings and numbers to a file
PRINT #*iochn*, ... | ?

Same rules as the normal print, but all the output is to the channel *iochn*. Note that you must put a comma after the channel number, not a semicolon.

On any error, ERR() will hold an error code, on success ERR() reads 1.

Note that you can only read the error for the last element printed.

Outputs one byte to the file
PUT #*iochn*, *num* | PU.

Outputs one byte *num* to the channel *iochn*.

On any error, ERR() will hold an error code, on success ERR() reads 1.

Generic I/O operation
XIO #*iochn*, *cmd*, *aux1*, *aux2*, *dev* | X.

Performs a general input/output operation on device *dev*, over channel *ioc*, with the command *cmd* and auxiliary bytes *aux1* and *aux2*.

Note that the arguments of XIO statements are in different order than Atari Basic, for consistency with other statements the *iochn* is the first argument.

Example: to delete the file "FILE.TXT" from disk, you can do:

```
1   XIO #1, 33, 0, 0, "D:FILE.TXT"
```

6.5 General Statements

Line comments
' / .

Any line starting with a dot or an apostrophe will be ignored.

Defines array with initial values
DATA *arr()* [type] = n1,n2, / DA.

This statement defines an array of fixes length with the values given.

The array name should not be used before, and type can be `BYTE` (abbreviated `B.`) or `WORD` (abbreviated `W.`).

If you end the `DATA` statement with a comma, the following line must be another `DATA` statement without the array name, and so on until the last line.

Example:

```
1   DATA big() byte = $12,$23,$45,
2   DATA       byte = $08,$09,$15
```

Note that the array can be modified afterwards like a normal array.

Decrements variable by 1
DEC *var* / DE.

Decrements the variable by 1, this is equivalent to "*var* = *var* - 1", but faster.

Allocate an array
DIM *arr*(*size*) [type], . . . / DI.

The `DIM` statement allows defining arrays of specified length.

The type must be `BYTE` (abbreviated `B.`) to define a byte array, with numbers from 0 to 255, or `WORD` (can be left out) to define an array with integers from -32768 to 32767.

If the name *arr* ends with a `$` symbol, this defines a string array, and you can't specify a type.

The size of the array is the number of elements, the elements are numerated from 0, so that an array of size 10 holds values from 0 to 9.

You can `DIM` more than one array by separating the names with commas.

The array is cleared after the `DIM` , so all elements are 0 or an empty string.

Ends program
END : Ends program.

Terminates current program, only valid at end of input.

Increments variable by 1
INC *var*

Increments the variable by 1, this is equivalent to "*var* = *var* + 1", but faster.

Pauses execution
PAUSE *num* | PA.

Stops the current execution until the specified time.

num is the time to pause in "jiffies", this is the number of TV scans in the system, 60 per second in NTSC or 50 per second in PAL.

A value of 0 pauses until the vertical retrace, this is useful to synchronize to the TV and get fluid animations.

6.6 Floating point statements

Those statements are only available in the floating point version.

Sets "degrees" mode
DEG

Makes all trigonometric functions operate in degrees, so that 360 is the full circle.

Sets "radians" mode
RAD

Makes all trigonometric functions operate in radians, so that 2pi is the full circle.

This mode is the default on startup.

6.7 Low level statements

Those are statements that directly modify memory. Use with care.

Writes a 16bit number to memory
DPOKE *address*, *value* | D.

Writes the *value* to the memory location at *address* and *address+1*, using standard CPU order (low byte first).

Copies bytes in memory
MOVE *from*, *to*, *length* | M.
-MOVE *from*, *to*, *length* | -.

Copies *length* bytes in memory at address *from* to address *to*.

The `MOVE` version copies from the lower address to the upper address, the `-MOVE` version copies from upper address to lower address.

The difference of the two statements is in case the memory ranges overlap, if *from* is lower in memory than *to*, you need to use `-MOVE`, else you need to use `MOVE`, otherwise the result will no be a copy.

`MOVE a, b, c` is equivalent to:

```
1   FOR I=0 to c-1
2     POKE b+I, PEEK(a+I)
3   NEXT I
```

but `-MOVE a, b, c` is instead:

```
1   FOR I=c-1 to 0 STEP -1
2     POKE b+I, PEEK(a+I)
3   NEXT I
```

Sets memory to a value
MSET *address*, *length*, *value* | MS.

Writes *length* bytes in memory at given *address* with *value*.

This is useful to clear graphics or P/M data, or simply to set an string to a repeated value.

`MSET a, b, c` is equivalent to:

```
1   FOR I=0 to b-1
2     POKE a+I, c
3   NEXT I
```

Writes a byte to memory
POKE *address*, *value* | P.

Writes the *value* (modulo 256) to the memory location at *address*.