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 1980s era 8-bit computers is the lack of line numbers, as well as an integrated full-screen editor. This is similar to newer programming environments, giving the programmer a higher degree 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 relative to its peers from the 1980s.

The other reason is that the program is parsed on run, generating optimized code for very fast execution.

Currently, FastBasic supports:

- 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”, floating point and strings.
- User defined procedures, with integer parameters.
- 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 4.6 - (c) 2022 dmsc
'
' Editor Help
' -----------
'  Ctrl-A : Move to begining of line
'  Ctrl-E : Move to end of line
'  Ctrl-U / Ctrl-I : Page up / down
'  Ctrl-Z : Undo (only current line)
'  Ctrl-C : Set Mark to current line
'  Ctrl-V : Paste from Mark to here
'  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
'  Ctrl-G : Go to line number
'
'- 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. Please 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 are ready to start writing your own BASIC program. Try the following example, pressing `RETURN` after each line to advance to the next:

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

The parser will let you know if you made any mistakes. To make corrections 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 below 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 there are no errors with your program, it will be run now: the computer screen will show `WHAT IS YOUR NAME?`, type anything and press `RETURN`, the computer will reply with a greeting and the program will end.

After this, the IDE waits for any key press before returning to the editor, so you have a chance to see your program's output.

If you press the `BREAK` key when the program is running, it will terminate, wait for a key press and return to the IDE.

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

Remember to save often by pressing the `CONTROL` and `S` keys and entering 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 filename for your compiled program. It is common practice to name your compiled programs with an extension of ".COM" or ".XEX." With ".COM" extension files you don't need to type the extension in some versions of 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 filename to write the compiled program.

4 Advanced Editor Usage

The editor includes a few commands, most of those are already explainded above.

- `CONTROL-A` and `CONTROL-E` Moves the cursor ro the beggining or the end of the line respectively.
- `CONTROL-U` and `CONTROL-I` Moves the cursor 19 lines up or down respectively.
- `CONTROL-G` Moves the cursor to a specific line.
- `CONTROL-Z` Reverts all editing of the current line. Note that changing the line clears the undo buffer, so you can't undo more than one line.
- `CONTROL-C` Sets the current line as the source for the copy operations.

- `CONTROL-V` Copy one line from the source marked with the `CONTROL-C` to the current cursor position. After the copy, the source line is advanced, so by pressing `CONTROL-V` multiple times you can copy multiple consecutive lines.
- `CONTROL-L` and `CONTROL-S` Loads or Saves the file being edited, respectively.
- `CONTROL-Q` Returns to DOS, abandoning the changes in the current file.
- `CONTROL-R` Parses the current program and runs it.
- `CONTROL-W` Compiles the current program and saves it to a binary file.

5 Making the Editor Faster

The FastBasic IDE uses the Atari screen handler for writing text, so it is compatible with all 80 column and other expansions available.

As the original screen handler in the Atari OS is slow, there is a screen accelerator included in the FastBasic disk.

To use the accelerator, just type `EFAST` in the DOS prompt, before loading the IDE, and enjoy editting programs faster.

6 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 has 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*".

7 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.

7.1 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`)

7.2 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%`.

In FastBasic, variables can't be used in an expression before being assigned a value, the first assignment declares the variable.

7.3 Numeric Operators

There are various "operators" that 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 a need to alter the precedence, you can put the expression between parenthesis.

Note that `MOD` and `EXOR` can be abbreviated `M.` and `E.` respectively.

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, as this allows mixing integers and floating point in some calculations, but you must take 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 than 32767.

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

7.4 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.
- `<=` `>=` `<>` `<` `>` `=` For integer or floating point comparisons, 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.

The words `OR` , `AND` and `NOT` can be abbreviated `O.` , `A.` and `N.`

7.5 Arrays

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

In FastBasic, arrays must be dimensioned before use (see `DIM` statement below). The index of the element is written between parentheses and goes from 0 to the number of elements. Note that FastBasic does not check for out of boundary accesses, so you must be careful with your code to not overrun the size of the arrays.

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 four 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.
- Floating point arrays, works like any floating point variable, and use six bytes of memory for each element.
- String arrays store a string in each element. String arrays use two bytes of memory for each element that is not yet assigned (containing empty strings), and 258 bytes for each element with a string assigned.

7.6 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:

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

Will print:

```
Hello "world"
```

You can also include any character with it’s hexadecimal code using `$` just after the closing quote, with no spaces around. This is the only way to include an ENTER character inside a string constant, see this example:

```
PRINT "Hello"$9B"world"$2E$2E
```

Will print:

```
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"
```

7.7 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
```

7.8 Functions

Functions take parameters (normally between parentheses) and produce a result. Functions can be abbreviated by using a shorter name ended in a dot, for example you can write `R.(10)` instead of `RAND(10)`.

You can also omit parentheses on functions that take one numerical value as argument, for example, `RAND 10`. This is not possible when the argument is a string or in `ADR` and `USR`.

Some functions don't take parameters, and you must provide a set of parentheses, like `KEY()`, in this case when abbreviated, you can omit the parenthesis, like `K.`.

7.9 Standard Functions

Following is a list of all the general purpose functions supported by FastBasic. Shown are the full syntax and the abbreviated syntax.

- TIME / T. : Returns the current time in "jiffies." This is about 60 times per second in NTSC systems or 50 times per second in PAL systems. Use `TIMER` statement to reset to 0.
- ABS(*num*) / A.(*num*) : Returns the absolute value of *num*. Can be used with integers and floating point.
- SGN(*num*) / SG.(*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*) / R.(*num*) : Returns a random, non negative number, a maximum of 1 less than *num*.
- FRE() / F. : Returns the free memory available in bytes.
- ERR() / E. : Returns the last Input/Output error value, or 1 if no error was registered.
- LEN(*string*) / L.(*string*) : Returns the length of the *string*.
- VAL(*string*) / V.(*string*) : Converts *string* to a number. If no conversion is possible, `ERR()` is set to 18. Can be used with integers and floating point.
- ASC(*string*) / AS.(*string*) : Returns the ATASCII code of the first character of the *string*.

7.10 Atari Specific Functions

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

- PADDLE(*n*) / PA.(*n*) : Returns the value of the PADDLE controller *n*.
- PMADR(*n*) / PM.(*n*) : Returns the address of the data for Player *n* or the address of the Missiles with *n* = -1.
- PTRIG(*n*) / PT.(*n*) : Returns 0 if the PADDLE controller *n* button is pressed, 1 otherwise.
- STICK(*n*) / S.(*n*) : Returns the JOYSTICK controller *n* position. `STICK(_n_)` values are:

```
`10`  `14`  ` 6`

`11`  `15`  ` 7`
```

```

` 9`  `13`  ` 5`
```

- STRIG(*n*) / STR.(*n*) : Returns 0 if JOYSTICK controller *n* button is pressed, 1 otherwise.
- KEY() / K. : 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, 255` statement).

  Hint: The value returned is actually the same as `(PEEK(764)EXOR 255)` . The following program will show the `KEY()` codes for pressed keys:

```
PRINT "Press keys, exit with ESC"
REPEAT
  REPEAT : UNTIL KEY()
  PRINT "Key code: "; KEY()
  GET K
  PRINT "ATASCI code: "; K
UNTIL K=27
```

7.11 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 will return an invalid value, and the `ERR()` function returns 3.

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

7.12 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.

7.13 Low level Functions

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

- ADR(*arr*) / &*arr* : Returns the address of the first element of *arr* in memory. Following elements of the array occupy adjacent memory locations. Instead of `ADR(X)` you can simply type `&X`.
- ADR(*str*) / &*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.
- ADR(*var*) / &*var* : Returns the address of the *variable* in memory.
- DPEEK(*addr*) / D.(*addr*) : Returns the value of memory location *addr* and *addr*+1 as a 16 bit integer.
- PEEK(*address*) / P.(*address*) : Returns the value of memory location at *address*.
- USR(*address*[,*num1* ...]): Low level function that 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 code snippet:

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

- $(*addr*) : Returns the string at memory address *addr*.

  This is the inverse of `ADR()`, and can be used to create arbitrary strings in memory. For example, the following code prints "AB":

```
DATA x() byte = 2, $41, $42
? $( ADR(x) )
```

Also, you can store string addresses to reuse later, using less memory than copying the full string:

```
x = ADR("Hello")
? $( x )
```

8 List Of Statements

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

8.1 Console Print and Input Statements

Reads Key From Keyboard
GET *var* / GE.

Waits for a keypress 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* INPUT ; *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`). In the case you don't want any prompt, you can use a semicolon alone.

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().

See the *Device Input and Output Statements* section for the `INPUT #` usage.

Moves The Screen Cursor
POSITION *column*, *row* / POS.

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

Rows and columns are numerated from 0.

Print Strings And Numbers
PRINT *expr*, ... / ?
PRINT *expr* TAB(*expr*) ...
PRINT *expr* RTAB(*expr*) ...
PRINT COLOR(*expr*) *expr* ; ... 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, semicolon or `TAB`, where the cursor stays in the last position.

If there is a comma before any expression, spaces are printed to advance the printing column to the next multiple of 10, allowing easy printing of tabulated data.

The `COLOR` function alters the color the text that follows, depending on the graphics mode. This is abbreviated `C.`. Use 0 or 128 in graphics 0, for normal or inverse video. Use 0, 32, 128 or 160 in graphics mode 1 and 2 for the four available text colors, see the two examples bellow:

```
' In GRAPHICS 0:
? "NORMAL"; COLOR(128) "INVERSE"

' In GRAPHICS 2:
S = 1234
? #6, "SCORE: "; COLOR(32) S
```

Note that the color only applies to one argument, and it is reset to 0 after a `;`, `,` or `TAB`.

The `TAB` function advances the position to a column multiple of the argument, so that `TAB(10)` is the same as using a comma to separate arguments. This is abbreviated `T.`.

The `RTAB` function advances the position so that the next argument to print ends just before a column multiple of the argument, right aligning the printing of the data. This is abbreviated `RT.`.

Note that `,`, `TAB` and `RTAB` always print at least one space, and that to separate `TAB` or `RTAB` and the previous and next arguments you can use a `;` or simply a space.

See the *Device Input and Output Statements* section for the `PRINT #` usage.

This example shows the usage of `TAB` and `RTAB`, note that the columns will be left and right aligned respectively:

```
FOR i=0 TO 10
  n = i*(9-2*i)*134
  ? TAB(8) "Val:" RTAB(20) n
NEXT
```

The output is:

```
Val:0          0
Val:1        938
Val:2       1340
Val:3       1206
Val:4        536
Val:5       -670
Val:6      -2412
Val:7      -4690
Val:8      -7504
Val:9     -10854
Val:10    -14740
```

Advanced: To implement the spacing on `,`, `TAB` and `RTAB`, FastBasic uses the current column in the OS, so that `POSITION` and printing to a graphics screen will work ok, unlike Atari BASIC; but when printing to a file or other devices the number of spaces will not be correct. Avoid using the functions to print to any device except the screen.

Advanced: The `COLOR` function does an *exclusive or* of the given value with the value of each character in the original string before printing.

Writes A Character To Screen
PUT *num* / PU.

Outputs one character to the screen, given by it’s ATASCII 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`.

8.2 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 an EXIT statement.

Calls A Subroutine
EXEC *name num1*, … / EXE. / @

Calls the subroutine *name*, with the optional parameters *num1* and so on, separated by commas.

Note that the subroutine must be defined with PROC with the same number of parameters, but can be defined before or after the call.

Instead of `EXEC` you can simply use a `@` in front of the procedure name.

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. T. 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, the command first compares the value of *var* with *end*, if the value is past the end it 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, the iteration ends and the if value of *var* is bigger than *end*, but if *step* is negative, the 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.

A slightly modified usage of the FOR/NEXT loop allows for excluding the variable name from NEXT; this is required if *var* is an array.

This is an example of NEXT without variable:

```
 ' sample of FOR/NEXT loop without
 ' NEXT variable name
FOR i=0 to 1000 step 100
  ? i
NEXT
```

Conditional Execution
IF *condition* THEN *statement* / I. T.
IF *condition*
ELIF *condition* / ELI.
ELSE / EL.
ENDIF / E.

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

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

If the 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 var1 . . .* / PR.
ENDPROC / ENDP.

PROC statement starts the definition of a subroutine that can be called via EXEC or `@`.

You can pass a list of integer variables separated by spaces after the PROC name to specify a number of parameters, the variables will be set to the values passed by the EXEC call. Those variable names are always global, so the values set are seen outside the PROC.

The number of parameters in the PROC definition and in all the EXEC calls must be the same.

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.

Firstly it evaluates the condition. If false, it skips the whole loop to the end. If true, it 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.

8.3 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 the 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 the last position to the given *x* and *y* position using `COLOR` number. For each plotted point it also paints all points to the right with the `FCOLOR` number, until a point with different color than the first is reached.

Sets Graphic Mode
GRAPHICS *num* / G.

Sets the graphics mode for graphics operations. Below is a basic chart of GRAPHICS modes, their full screen resolution and number of available colors.

Mode	Resolution	# Of Colors
GR. 0	Text 40x24	1
GR. 1	Text 20x24	5
GR. 2	Text 20x12	5
GR. 3	40x24	4
GR. 4	80x48	2
GR. 5	80x48	4
GR. 6	160x96	2
GR. 7	160x96	4
GR. 8	320x192	1
GR. 9	80x192	16 shades of 1 color
GR. 10	80x192	9
GR. 11	80x192	16
GR. 12	Text 40x24	4
GR. 13	Text 40x12	4
GR. 14	160x192	2
GR. 15	160x192	4

Get color of pixel
LOCATE *x*, *y*, *var* / LOC.

Reads the color of pixel in the specified *x* and *y* coordinates and store into variable *var*.

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 Graphics Mode
PMGRAPHICS *num* / PMG.

Set up Atari Player / Missile graphics. A value of 0 disables all player and missiles; a value of 1 sets up single line resolution; a value of 2 sets up double line resolution.

Single line mode uses 256 bytes per player, while 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* / PM.

Set the 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

Adjusts sound parameters for *voice* (from 0 to 3) of 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, it clears all voices so that no sounds are produced.

8.4 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*, . . . / ?

Uses the 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:

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

8.5 General Statements

Line comments
' / .

Any line starting with a dot or an apostrophe will be ignored. This is analogous to REM in Atari BASIC.

Clears variables and free memory
CLR

Clears all integer and floating-point variables to 0, all strings to empty strings and frees all memory associated with arrays.

After `CLR` you can't access arrays without allocating again with `DIM`.

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

This statement defines an array of fixed 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 no type is given, a word data is assumed.

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:

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

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

Advanced Usage

Byte DATA arrays can be used to include assembler routines (to call via `USR`, see the example above), display lists and any other type of binary data.

To facilitate this, you can include constant strings and the address of other byte DATA array by name.

All the bytes of the string, including the initial length byte are included into the DATA array.

Example:

```
DATA str() B. = "Hello", "World"
X = ADR(str)
? $(X), $(X+6)
DATA ad() B. = $AD,str,$A2,0,$60
? USR(ADR(ad)), str(0)
```

Loading data from a file

The cross-compiler also supports loading data from a file directly into the program, using the `BYTEFILE` (abbreviated `BYTEF.`) and `WORDFILE` (abbreviated `WORDF.` or simply `F.`) types and a file name enclosed in double quotes.

Example:

```
DATA img() bytefile "img.raw"
DATA pos() wordfile "pos.bin"
```

The compiler will search the file in the same folder than the current basic source.

Storing data into ROM

In addition to the above, the cross compiler allows to specify that the data should be stored in ROM, instead of the default in RAM. This means that the data can't be modified in targets that use ROM (cartridges), but will lower RAM usage.

To specify this, simply add the `ROM` word after the type:

```
DATA img() ROM 1234,5678
DATA pos() BYTE ROM 1,2,3,4
```

Decrements variable by 1
DEC *var* / DE.

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

var can be any integer variable or integer array element.

Allocate an Array / Define Var
DIM *arr*(*size*) [type], . . . / DI.
DIM *var*, *var$*, *var%* . . .

The `DIM` statement allows defining arrays of specified length, and declaring variables explicitly, without assigning a value.

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 `$` or a '%' symbol, this defines a string array or floating point array respectively, in this case you can't specify a type.

The size of the array is the number of elements plus one, the elements are numerated from 0, so that an array dimensioned to 10 holds 11 values, from 0 to 10.

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

In the second form, the variables given in the list are defined with the correct type, without giving a default value. The variables can be defined multiple times without an error if the types are always the same.

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

Example:

```
DIM A(10), X, T$
? A(5), X
```

Ends Program

END : Ends program.

Terminates current program. END is 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.

var can be any integer variable or integer array element.

Pauses Execution

PAUSE *num* / PA. PAUSE

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.

Omitting *num* is the same as giving a value of 0, and pauses until the vertical retrace. This is useful for synchronization to the TV refresh and for fluid animation.

Resets internal timer

TIMER/ T.

Resets value returned by `TIME` function to 0.

8.6 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.

8.7 Low Level Statements

These 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 between the two MOVE 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 not be a copy.

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

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

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

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

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

Writes A Byte To Memory
POKE *address*, *value* / P.

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

8.8 Display List Interrupts

Note: This is an advanced topic.

Display list interrupts (normally called `DLI`) are a way to modify display registers at certain vertical positions on the screen.

You can use them to:

- Display more colors in the image, by changing color registers - registers from $D012 to $D01A.
- Split one Player/Missile graphics to different horizontal positions - registers from $D000 to D007.
- Change scrolling position, screen width, P/M width, etc.

FastBasic allows you to specify one or more DLI routines, activate one or deactivate all DLI by using the `DLI` statement:

Define a new DLI
DLI SET *name* = *op1*, *op2*, . . . / DLIS.

Setups a new DLI with the given name and performing the *op* operations.

Each operation is of the form: *data* `INTO` *address* or *data* `WSYNC` `INTO` *address*.

data is one constant byte or the name of a `DATA BYTE` array, and *address* is a memory location to modify.

If *data* is a DATA array, the first element (at index 0) will be used at the first line with DLI active in the screen, the second element at the second active line, etc.

The `WSYNC` word advances one line in the display area (this is done by writing to the `WSYNC` ANTIC register), so the value is set in the next screen line. You can put the `WSYNC` word multiple times to advance more than one line. This allows one DLI to modify multiple lines at the screen.

Multiple `INTO` words can be used to write more than one register with the same value.

`INTO` can be abbreviated to `I.` and `WSYNC` to `W.` .

You can specify any number of operations, but as each one takes some time you could see display artifacts if you use too many.

Note that by defining a DLI you are simply giving it a name, you need to activate the DLI afterwards.

You can split a DLI definition over multiple lines, just like DATA by ending a line with a comma and starting the next line with `DLI =`

Enable a DLI
DLI *name* / DL.

This statement enables the DLI with the given name, the DLI must be defined before in the program.

This setups the OS DLI pointer to the named DLI and activates the interrupt bit in the display processor (the ANTIC chip), but does not activates on which lines the DLI must be called.

To define on which lines the DLI is active you must modify the *Display List*, see the example at the end of the section.

You can also pass the name of a DATA BYTE array with a custom machine language routine to the `DLI` statement, the routine must begin with a *PHA* and end with *PLA* and *RTI*.

Disable a DLI
DLI / DL.

This statement simply disables the DLI, returning the display to the original

DLI Examples

This is the most basic example of a DLI that simply changes the background color at the middle of the screen:

```
' Define the DLI: set background
' color to $24 = dark red.
DLI SET d1 = $24 INTO $D01A
' Setups screen
GRAPHICS 0
' Alter the Display List, adds
' a DLI at line 11 on the screen
POKE DPEEK(560) + 16, 130
' Activate DLI
DLI d1
' Wait for any keyu
? "Press a Key" : GET K
' Disable the DLI
DLI
```

The next example shows how you can use a DLI to change multiple values in the screen:

```
' An array with color values
DATA Colors() BYTE = $24,$46,$68
```

```
' Define the DLI: set background
' color from the Color() array
' and text back color with value
' $8A in the same line and then
' the black in to the next line.
DLI SET d2 = Colors INTO $D01A,
DLI        = $8A INTO $D018,
DLI        = $00 WSYNC INTO $D018
' Setups screen
GRAPHICS 0
' Adds DLI at three lines:
POKE DPEEK(560) + 13, 130
POKE DPEEK(560) + 16, 130
POKE DPEEK(560) + 19, 130
' Activate DLI
DLI d2
' Wait for any keyu
? "Press a Key" : GET K
' Disable the DLI
DLI
```

The final example shows how you can move multiple P/M using one DLI

```
' Player shapes, positions and colors
DATA p1() BYTE = $E7,$81,$81,$E7
DATA p2() BYTE = $18,$3C,$3C,$18
DATA pos() BYTE = $40,$60,$80,$A0
DATA c1() BYTE = $28,$88,$C8,$08
DATA c2() BYTE = $2E,$80,$CE,$06
' Our DLI writes the position and
' colors to Player 1 and Player 2
DLI SET d3 = pos INTO $D000 INTO $D001,
DLI        = c1 INTO $D012, c2 INTO $D013
GRAPHICS 0 : PMGRAPHICS 2
' Setup our 4 DLI and Players
FOR I = 8 TO 20 STEP 4
  POKE DPEEK(560) + I, 130
  MOVE ADR(p1), PMADR(0)+I*4+5,4
  MOVE ADR(p2), PMADR(1)+I*4+5,4
NEXT
' Activate DLI
DLI d3
? "Press a Key"
```

```
REPEAT
  PAUSE
  pos(0) = pos(0) + 2
  pos(1) = pos(1) + 1
  pos(2) = pos(2) - 1
  pos(3) = pos(3) - 2
UNTIL KEY()
DLI
```

Some usefull registers

This is a table of some useful registers to change during a DLI:

Address	Register
$D000	Player 0 horizontal pos.
$D001	Player 1 horizontal pos.
$D002	Player 2 horizontal pos.
$D003	Player 3 horizontal pos.
$D004	Missile 0 horizontal pos.
$D005	Missile 1 horizontal pos.
$D006	Missile 2 horizontal pos.
$D007	Missile 3 horizontal pos.
$D012	Color of player/missile 0
$D013	Color of player/missile 1
$D014	Color of player/missile 2
$D015	Color of player/missile 3
$D016	Color register 0
$D017	Color register 1
$D018	Color register 2
$D019	Color register 3
$D01A	Color of background

9 Atari 5200 console support

Note: This is an appendix to the main FastBasic manual.

The FastBasic cross compiler supports the Atari 5200 console as a target, allowing to port programs from the Atari 8-bit computers with minor modifications.

9.1 Controllers Support

The Atari 5200 does not have standard digital joysticks, console keys and keyboard, all controllers are analog and have a keypad.

FastBasic emulates the standard 8-bit joystick and keyboard using the controllers for easy porting:

- The `STICK()` and `STRIG()` functions return the same values as in the computers.
- The `PTRIG()` function returns the state of the secondary button for the controllers.
- The `PADDLE()` functions return the analog value of each controller axis, so `PADDLE(0)` and `PADDLE(1)` are the horizontal and vertical axis of the first controller respectively, and so on with the next 4 controllers.
- The `GET` statement waits for a key pressed on any controller, the value returned is from 0 to 15 for the first controller, 16 to 31 for the second, etc.
- The `KEY()` function returns the last pressed key on *any* controller, or 0 if no key was pressed.

The values returned by `GET` are given in the following table, note that you can *and* the number with 15 to get the number of the first controller, this is useful to accept keys from all controllers in your code:

Key	Controller 1	Controller 2	Controller 3	Controller 4
`0`	0	16	32	48
`1`	1	17	33	49
`2`	2	18	34	50
`3`	3	19	35	51
`4`	4	20	36	52
`5`	5	21	37	53
`6`	6	22	38	54
`7`	7	23	39	55
`8`	8	24	40	56

Key	Controller 1	Controller 2	Controller 3	Controller 4
`9`	9	25	41	57
`*`	10	26	42	58
`#`	11	27	43	59
Start	12	28	44	60
Pause	13	29	45	61
Reset	14	30	46	62

The value returned by the `KEY()` function is always 255 minus the numbers above, for example, when pressing the key `4` in the second controller, `KEY()` will return 255-20 = 235.

9.2 RAM Usage

The Atari 5200 console has only 16kb of RAM, this means that you have to minimize RAM usage in your program.

For compatibility with most programs, `DLI` and `DATA` statements are stored in RAM, this means that you should put big `DATA` statements in ROM by using the construct:

```
DATA mydata() BYTE ROM = 1, 2, 3, ....
DATA font() BYTEFILE ROM "myfont.fnt"
```

Note that DATA in ROM can't be modified at runtime, so you must select the type appropriately.

9.3 Operating System support

The Atari 5200 console lacks the OS ROM of the Atari 8-bit computers, so all the functionality that depends on the OS must be reimplemented.

Currently, the target lacks:

- Floating point support: only integer operations are supported.
- Graphics modes: the included runtime only implements graphic modes 0, 1, 2, 7 to 13 and 15.
- Text window: the graphic modes with a text window are not implemented.
- Graphics statements: only PLOT, DRAWTO and LOCATE are implemented, there is no `FILLTO` command.
- Input/Output: Only the screen output an keypad input are supported, there is no I/O channels, `OPEN` or `XIO`.

9.4 Hardware Registers

The Atari 5200 changes the location of hardware registers for POKEY (from $D200 to $D800) and GTIA (from $D000 to $C000), so direct `POKE`s to memory must be changed.

Also, `DLI` support also needs to change the address of color and P/M registers, you can use this table for useful registers:

Address	Register
$C000	Player 0 horizontal pos.
$C001	Player 1 horizontal pos.
$C002	Player 2 horizontal pos.
$C003	Player 3 horizontal pos.
$C004	Missile 0 horizontal pos.
$C005	Missile 1 horizontal pos.
$C006	Missile 2 horizontal pos.
$C007	Missile 3 horizontal pos.
$C012	Color of player/missile 0
$C013	Color of player/missile 1
$C014	Color of player/missile 2
$C015	Color of player/missile 3
$C016	Color register 0
$C017	Color register 1
$C018	Color register 2
$C019	Color register 3
$C01A	Color of background