Bob's Tiny BASIC

Version 0.3

bob@corshamtech.com

Ever since reading the first year of Dr. Dobb's Journal of Computer Calisthenics and Orthodontia (yes, that was the original name of the magazine), I wanted to write a tiny BASIC interpreter using the intermediate language (IL) method. The first couple years of DDJ printed source code to several BASICs but none of them used IL.

Well, the idea was always in the back of my mind, so one day I re-read the articles, found some good web pages about the topic, and started writing my own in 6502 assembly language. While it can easily be argued that this was not a good use of my time, it was fun and very satisfying, reminding me of the days when I dreamed of having a high level language on my KIM-1 computer.

So here it is, Bob's Tiny BASIC. It's not as tiny as it could be, but it does have some support for program storage/retrieval. It has support for the base KIM-1 computer, the xKIM monitor by Corsham Technologies, and the CTMON65 monitor by Corsham Technologies. The source is on github.

Licensing is too complicated, so here are my desires in fairly plain language: no restrictions.

Numbers and Variables

BTB only supports 26 integer variables named A to Z.

Numbers are signed 16 bit integers, with a range of -32768 to 32767.

Expressions

FREE()

RND()

ABS()

Commands

END

Stops the currently running program, returning the user to the prompt.

EXIT

Returns back to the underlying OS/Monitor.

GOTO <expression>

Computes the value of the expression and then jumps to that line number, or the next line after it, if that specific line does not exist.

GOSUB <expression>

Compute the value of the expression and then calls a subroutine at that line, or the next line after it. Return back to the calling point with the RETURN keyword.

IF <expression> THEN <statement>

INPUT <variable> [<variable> …]

Prints a question mark, gets the user's input, convers to a number, then saves the value to the specified variable.

LET <variable> = <expression>

Assigns a value to a variable. Unlike some BASICs, BTB does not assume a LET. Ie, you can't just type "A = 42", you must use "LET A = 42."

LOAD <filename>

Loads the specified file into memory. The file is just a text file, so you can edit programs using another editor, then load them with this command. Note that this like typing in lines at the prompt, so if there is an existing program in memory and another is loaded, they are "merged" together.

NEW

This clears the program currently in memory. There is no mercy, no second chances, and no confirmation. The existing program is gone, instantly.

PRINT <values>

Print can have quoted strings, commas, semicolons, numbers and variables. Commas move to the next tab stop, while semicolons don't advance the cursor.

REM [<comments>]

The rest of the line is ignored. It is a comment. It is not mandatory to have any text after the REM keyword. Comments made code easier to read, but they also take time to execute, so too many comments can slow down the code.

RETURN

Will return to the next statement following the GOSUB which brought the program to this subroutine.

RUN

Begins execution of the program currently in memory starting at the lowest line number.

SAVE <filename>

Save the current program to the specified filename. Note that the filename is used exactly as specified; nothing (like ".BAS") is automatically added.

Error Codes

1 = Expression
2 = Stack underflow (expression error)
3 = Stack overflow (expression is too complex)
4 = Unexpected stuff at end of line
5 = Syntax error (possibly unknown command)
6 = Divide by zero
7 = Read fail loading a file

8 = Write fail saving a file
9 = No filename provided

Improving Speed

Tiny BASIC on a 6502 using IL is slower than a machine language program, by a huge margin, but there are steps to slightly improve performance.

- Don't use a lot of REM statements, at least not near the beginning of the code. Every REM must be skipped at run time.
- Put heavily used code closer to the front of the program so those line numbers can be found quicker.
- Put one-use initialization code near the end, call them as a subroutine and return.
- Use variables instead of constants. Constants have to be converted from ASCII characters into an integer, while variables are quick to look up the binary values.