TRS-8051 BASIC Interpreter Overview

Version 1.01 (c) 2017 by Dennis Hawkins
dennis@galliform.com

INTRODUCTION

This project started out as a hardware simulation of a TRS-80 Model III running Level 2 BASIC. The hardware consisted of a single Silicon Labs Laser Bee 8051 CPU and a RAM chip. The Laser Bee alone literally has about the same power as the original Model I TRS-80. Using the Laser Bee and a single RAM chip, I was able to generate a stable VGA video signal entirely in software with plenty of CPU cycles left to actually do stuff.

With the hardware working, I turned my attention to developing a BASIC interpreter. There are numerous interpreters already out there, but they are lacking in functionality. Many were based on Tiny Basic which is an integer only BASIC with only single letter variables. So I decided to write my own Level II BASIC complete with full floating point support. This was easier said than done and after having done it, I have found a great deal of respect for the original author, Bill Gates, who allegedly wrote Level II BASIC in a couple of months. This project took me about 6 months to complete in my spare time and it still isn't really done.

Then to add insult to injury, after the project was nearly complete, I stumbled across another TRS-80 Level 2 BASIC interpreter and source called AWBASIC written by Anthony Wood. He had already written his interpreter back in 2002. Although not designed for the 8051, had I known about it earlier, it would have shortened my design cycle considerably.

When I decided to write this interpreter from scratch, I decided to start out developing it on a Windows platform because the development tools are a lot easier to use than they are for the 8051. I use a hardware abstraction layer to emulate my TRS-8051 hardware at the source level. That is why this program exists in a Windows version.

Someone may look at the source and say, “Wow, this is a really stupid way to do it”, but I can assure everyone that some odd coding was done deliberately to make the code compile better binaries for the 8051. On the Laser Bee, there is only 60K of program space and 2K internal RAM remaining after the Video resources are deducted. As a result, the code is laced with tons of globals and goto statements, and function parameters are deliberately kept to a bare minimum. The hope is that this style of coding will help keep the 8051 code segment size down.

The problem is that the 8051 instruction set has far less code density than the x86 processor used by Windows. Since the Windows version uses 97K, it means that it wont fit inside of an 8051. Most of the Windows code that is used for the hardware abstraction layer is done with DLL's so its not as big a part of the code size as you would think. So the final version that is actually ported back to the TRS-8051 hardware will have to have stuff hacked out of it. I also try to take advantage of sub-local variables any chance I get.

DIFFERENCES WITH THE REAL MICROSOFT LEVEL 2 BASIC

I started out trying to make my interpreter 100% compatible with the real TRS-80 Level 2 BASIC, but ended up having to settle for 95% compatible. There are a few differences.
Some, I hope, for the better, and some not. Here are the ones that I can think of.

Overall Syntax

The original Level 2 BASIC allowed seemingly anything goes syntax rules. For example, in most other BASICs, you need some kind of separator between arguments of a PRINT statement such as a semicolon. Not so in Level 2 BASIC. As long as the interpreter can tell what you want, it doesn't complain and operates properly. With TRS-8051 BASIC, you will need to supply argument separators and syntax is more strictly enforced.

So: "PRINT 4 5 6" works on the real LEVEL II BASIC, but you need "PRINT 4;5;6" here.

Likewise, with the IF-THEN statement. Most other BASICs require the THEN keyword, but Level 2 BASIC does not as long as it doesn't make things too ambiguous. So: "IF X=1 PRINT A$;" works in the original, but "IF X=1 **THEN** PRINT A$" is required here. I did make a special exception for GOTO. In TRS-8051 BASIC, "IF X=1 GOTO 300" Works fine.

Variable Hashing

In order to allow the most flexibility with variable names, its important not to restrict the variable name length too much. But to allow long variable names would waste program space and also cause the program to operate more slowly. With interpreters, a good compromise is to store the variable with a hashed version of the variable name rather than the variable name itself.

The original LEVEL 2 BASIC used a “brain dead” hash which consisted of the first to characters of the variable name and that was it. The problem is that variables with the same first two characters were considered the same by the interpreter, so "COUNT1" and "COUNT2" are the same variable. I had heard rumors growing up that Gates actually used the Length of the variable in his hash, but that rumor is false.

In TRS-8051 BASIC, the variable is hashed more uniquely. Here the hash is composed of the first and last characters, the variable name length, a 14 bit CRC and a 2 bit type field. Together these are packed into a unique 4 byte hash.

Although possible, its very unlikely with this hash to have a name collision that allows two variables with different names to be treated as the same. However, in order to limit the name collision possibilities, the variable names are limited to 31 characters. The original Level 2 BASIC did not have this restriction.

Tokenizer

I used a significantly different tokenizer than what the the original Level 2 BASIC used. As the line is entered, it gets tokenized then stored in the program file, and that is what the interpreter actually executes.

The original Level 2 BASIC tokenizer appears to be geared toward making the code size smaller. With TRS-8051 BASIC, its designed to make the program faster and actually takes slightly more space than untokenized code.

The original was simply a keyword tokenizer. So when the characters "P R I N T" were found in the source, they were replaced with a single byte TOKEN. So instead of taking 5 bytes, it only took 1.

Problems occur when keywords appear in variable names. So "SANDERS" is seen by the original BASIC as "S" AND "ERS" with "AND" being a keyword. This is a keyword collision with a variable name. In TRS-8051 BASIC, keyword collisions with variable names still occur, but are much less likely. A keyword collision only occurs if the keyword starts a variable name. So “SANDERS” works fine, but “ANDERSON” results in a collision.

With TRS-8051 BASIC, the interpreter pre-converts number and string constants to binary form. So the constant "3" takes five bytes instead of just one. Doing this conversion ahead of time makes the interpreter faster since converting on the fly takes quite a few CPU cycles.

In addition, it pre-hashes variable names to make execution faster. Because we need to have the original variable name still available for program listings, we have to store both the name and the hash. This means 4 more bytes than just the name alone. However, because the interpreter doesn't have to hash variables on the fly, it runs faster.

Variables

Variables must be pre-defined before use in TRS-8051 BASIC. Most other BASIC interpreters allow you to use a variable first without ever defining it. An undefined variable simply returns 0. In this basic, using an undefined variable results in an UNDEFINED VARIABLE error. To define a variable, just assign a value to it or use the DIM statement. DIM statements can be used to define simple variables as well as array variables.

In the original Level 2 BASIC, arrays under 10 elements did not have to be defined by a DIM statement. In this BASIC, all arrays, regardless of size, must be pre-defined or an UNDEFINED VARIABLE error occurs.

Only float, integer and string variables are allowed. There is no support for double precision floats. Integers are 32 bit longs instead of 16 bit shorts.

Video Screen

The original TRS-80 had a 1K RAM video buffer that supported a 64x16 character display. It generated an analog NTSC video signal. The hardware on the TRS-8051 uses an 8051 cpu to generate a VGA video signal in software. Because of the timing constraints of VGA, the number of character lines was increased to 25. This means the video buffer is now 1600 bytes instead of 1K. The Windows version of TRS-8051 BASIC emulates that hardware.

The TRS-8051 does not support wide mode (32x16) or switchable fonts. It does have a full 256 character font which includes both upper and lower case, various special characters, and graphics plotting characters. The character set is not identical to the TRS-80 and contains a mix of IBM and TRS-80 symbols and line drawing characters.

Specific Commands

There are some commands that are simply not here. Some are obvious. Because of hardware constraints, the commands USR, VARPTR, POKE, SYSTEM, INP and OUT are not supported.

TRON and TROFF are not supported.

CLOAD and CSAVE are replaced with LOAD and SAVE. Files are found in the "BasFiles" subdirectory. A listing of the files in the BasFiles directory is done with the FILES command. LOAD and SAVE only work with ASCII files and wont work with tokenized or cassette files.

Print to file, "PRINT #-1", is not supported.

The ERROR statement and ERR system variable uses the actual error number and not a calculated one.

RANDOM is not currently supported, but RND() is and works fine without it.

SET and RESET are now PLOT and UNPLOT.

Specific Functions

The POS(dummy parameter) function has now been replaced with POSX and POSY system variables. POSX and POSY together return the current cursor location.

FRE() is not supported because the string storage in this BASIC is more efficient and automatic than was in the original. Use the system variable MEM to get the total free space.

CDBL() is the same as CSNG() because doubles are not supported.

DATA statements must appear first on any given line or they wont be seen by the READ command.

NEW Commands and Functions

The following commands and functions are new in the TRS-8051 and are not supported by the original Level 2 BASIC.

A RENUM command that renumbers the program listing has been added that isn't in the original Level 2 BASIC. If the program line numbers get a little disorganized, just enter “RENUM” and the program is renumbered cleanly including any references inside GOTO and GOSUB statements.

GOTO and GOSUB statements now accept a line number expression instead of just a line number constant. So “GOTO 1000 + X*10” works. As long as the base line number appears first in the expression, it will be correctly handled by RENUM as well.

The INSTR() function searches a big string for a little string and returns the index of the position it was found at. This function was present in disk BASIC, but not in ROM BASIC.

RESTORE now takes an optional line number.

Memory Space

Because the BASIC program resides in external RAM, the program and variables are mapped differently from the original LEVEL 2 BASIC. With the original, the first 16K of program space was for the BASIC interpreter ROM code. This left only 48K for program memory. In this version, the BASIC interpreter resides on the Laser Bee 8051 so that leaves almost a full 64K RAM for program space.

The program memory is partitioned off into several variable sized partitions. Starting at the bottom of memory is the BASIC program itself. Followed by simple variables, array variables, the FOR-GOSUB stack, temporary strings, durable strings, and finally the command line area.

Command Line Editor

The original command line editor was convoluted and difficult to operate, and definitely not obvious. The line editor here is far simpler and more intuitive to use. Normal cursor keys work as expected. If you enter a line and don't have the syntax of your statement exactly right, it will return the line back to you and let you fix it.

FUNCTIONS

Parameterless Functions (A.K.A. System variables):

PI	Returns 3.14159265359 (**NEW**).
MEM	Returns unused memory amount.
POSX	Returns cursor X position.
POSY	Returns cursor Y position.
INKEY$	Returns keystroke or empty string.
ERR	Returns ERROR number of last error.
ERL	Returns Line number of previous error.
EXTKEY	Returns an integer containing the current status of the keyboard control keys. The bit fields are: (MSB to LSB) **(NEW)** R-WIN L-WIN R-ALT L-ALT R-CTRL L-CTRL R-SHIFT L-SHIFT
TIMER	Returns the number of milliseconds since the program started. **(NEW)**
TIME$	Returns numer of seconds since program started. **(MODIFIED)**

Single Parameter Functions:

ABS(Num)	Returns the Absolute Value.
ATN(Float)	Returns Arc Tangent (in radians).
COS(Float)	Returns the Cosine (in radians)
EXP(Float)	Returns the Natural Exponential.
LOG(Float)	Returns the Natural Logarithm.
SIN(Float)	Returns the Sine (in radians).
SQR(Float)	Returns the Square Root.
TAN(Float)	Returns the Tangent (in radians).
INT(Float)	Returns the parameter converted to an INT.
LEN(String)	Returns the Length of a string.
RND(Float)	Returns a random number.
SGN(Num)	Returns 1, 0, or -1 indicating sign of parameter.
VAL(String)	Returns numerical value of string.
ASC(String)	Returns ASCII code of first character in string.
CHR$(Integer)	Returns string made from specified ASCII code.
STR$(Float)	Returns a string version of the numerical parameter.
TAB(Integer)	Returns an empty string, but tabs cursor to specified position.

Double parameter functions:

STRING$(Integer, String)	Returns string with specified number of characters.
POINT(Integer, Integer)	Returns pixel value at given coordinates.

LEFT$(String, Integer)	Returns leftmost characters of string.
RIGHT$(String, Integer)	Returns rightmost characters of string.
AT$(Integer, Integer)	Returns empty string, but sets X/Y cursor position.
INSTR(String, String)	Returns index of little string inside of big string (**NEW**).

Triple parameter function:

MID$(String, Integer, Integer)	Returns substring of string parameter.

COMMANDS

Editing Commands:

AUTO {Line_Number}{, Increment}

Causes the line editor to automatically provide a new line number each time you hit return. Starts at specified line number or 10 if not specified. Upon hitting return, the next line number is incremented by the specified increment or 10 if not specified. If a line already exists, the '>' prompt is replaced with a '=' prompt. As long as you don't hit Return, the original line will not be replaced. To get out of AUTO mode, press ESC.

DELETE {Line_Number} {-} {Line_Number}

Deletes one or more program lines. A single line number deletes only that line number. If a '-' is specified, a range of line numbers will be deleted. If the beginning or ending line number in a range is omitted, then all lines from the first line or to the last line will be deleted. If no line numbers are specified, then this command does nothing.

EDIT Line_Number (**IMPROVED**)

Calls up the line editor for the specified line. Normal cursor keys work. ESC aborts editing. Return saves/executes line. If the line has no line number, it will be executed immediately and not stored in the program.

LOAD "<File_Name>" (**NEW**)

Attempts to load a BASIC file from the BasFiles sub-directory. This file must be in plain ASCII and not tokenized. Proper syntax is mandatory or else a load failure will occur and the load will be aborted. Lines are added according to the specified line number so the line numbers in the input file do not have to be in order.

The LOAD command has a built in preprocessor with several functions. Those functions are defined here.

AUTOMATIC LINE NUMBERS: If a line in the input file does not have a line number, it will be automatically added. The assigned line number will be 10 plus the previous line number. Lines in the input file with specified line numbers will change the line number counter to the specified line number. If a line number already exists, the hard coded line number is ignored.

AUTO RUN: If the keywords “AUTO RUN” appears at the first part of any line without a line number in the input file, the file will be automatically run as soon as loading is complete. The rest of the line is ignored.

HIDDEN COMMENTS: Regular REM and apostrophe comments take up space in the actual program. With the preprocessor, comments may be added to the input file that are ignored and not loaded. Lines with a semicolon in the very first column are ignored.

LABELS: Text labels may be used instead of line numbers in GOTO/GOSUB statements. Labels must start with an underscore character. For example, “_ThisIsALabel” and “ThisIsNotALabel”. So “GOTO _Loop2” works as long as a target label is defined called “_Loop2”. If a Label occurs as the first token in the line, it will define a target label. If a label is used in a GOTO/GOSUB statement that is not defined, it will cause the LOAD function to generate an “INVALID LINE NUMBER” error. Multiple definitions of the same label result in a “Multiple Definition” error.

It cannot be stressed enough that the preprocessor statements do not become part of the BASIC program once its loaded. So <u>if the program is subsequently saved, none of the preprocessor statements will be in the saved file.</u>

SAVE "<File_Name>" (**NEW**)

Attempts to save the current BASIC program to a plain untokenized ASCII file in the BasFiles directory. The saved file will not contain any preprocessor hidden comments, labels or explicit RUN commands.

FILES (**NEW**)

Shows the current files located in the BasFiles directory.

LIST {Line_Number} {-} {Line_Number}

List the program to the screen. Can list a single line or a range of lines between the '-' character. Will list all lines starting at first line number (or at the beginning if omitted) to the second line number (or the end if omitted). LIST by itself or with only the '-' will list the entire program.

RENUM {Old_Line_Number}{,{New_Line_Number}{,Increment}} (**NEW**)

Renumber the program starting at Old_Line_Number (or the beginning if not specified), changing to New_Line_Number (or 10 if not specified) and changing each subsequent line to a multiple of Increment (or 10 if not specified). All parameters may be omitted. Renum will fail if the New_Line_Number is less than the Old_Line_Number. Other commands that specify line numbers are adjusted to the new line number.

Direct Commands:

CLEAR

Clears the variable space including strings.

CLS

Clears the screen.

CONT

Continues program after a STOP command is executed.

NEW

Erases entire program space.

RENUM

Renumbers the program file.

RUN {Line_Number}

Erases all variables and jumps to Line_Number to begin execution. If Line_Number is omitted, jumps to first line.

EXIT

Exits the basic interpreter to Windows. This function will be omitted in the 8051 version.

Program Flow commands:

END
Terminates program execution and returns control to the line editor.

FOR Loop_Variable '=' Start_Val TO End_Val {STEP Step_Val}
NEXT {Loop_Variable}*
Initiates a FOR-NEXT loop. NEXT without any parameters loops to the last defined FOR loop. NEXT with parameters loops to the specified FOR loop. NEXT with multiple Loop_Variables loops to all of them in the order specified. FOR loops may be nested, but may not overlap.

GOSUB <Line_Expression>
RETURN
Evaluate the Line number expression and jump to the line number. When a return is executed, jump to the command after the GOSUB. GOSUB's may be nested.

GOTO <Line_Expression>
Evaluate the Line number expression and unconditionally jump to the line number.

IF <Expression> THEN <Cmd_List> { ELSE <Cmd_List>}
IF <Expression> THEN <Line_Number> { ELSE <Line_Number>}
IF <Expression> GOTO <Line_Expression> { ELSE <Command_List or Line_Number>}
Evaluate the Expression and if TRUE, begin executing commands after THEN up to the end of line or an ELSE keyword. A Line_Number by itself has an implicit "GOTO". The "THEN" keyword is optional for the GOTO command, but not for any other.

ON ERROR GOTO <Line_Number>
RESUME {Line_Number or "NEXT"}
When an error occurs, instead of printing an error, branch to the error handler at Line_Number. This command must be executed before the actual error occurs. At the end if the error handler, program execution is resumed by using the RESUME command. RESUME without any parameters jumps to the same command that caused the error. RESUME NEXT jumps to the following command. RESUME <Line_Number> jumps to a specific line number.

ON <Expression> GOTO Line_Number {,Line_Number}*
ON <Expression> GOSUB Line_Number {,Line_Number}*
Evaluates the expression and branches (GOTO or GOSUB) to the first Line_Number if expression equals 1, then the second if it equals 2, and so on. If the expression does not match a Line_Number, then fall through to the next command. The expression must evaluate to a number between 0 and 255 inclusive.

STOP
Stops program execution.

Other Commands:

DATA <String_or_Numeric_Constant> {, String_or_Numeric_Constant}*
Sets constant data in program. Does not do anything if executed. Strings must be quoted.

DATA commands MUST only appear as the first command on the line.

READ <String_or_Numeric_Variable> {,String_or_Numeric_Variable}*
Read the constant data from the DATA element pointed to by the DATA pointer. After reading, the DATA pointer is advanced to the next DATA element. The DATA element type and the variable type must match or a TYPE CONFLICT error is generated.

RESTORE {Line_Number} (**IMPROVED**)
Change the DATA pointer to the Line number specified or to the first DATA statement if not. The original LEVEL 2 BASIC did not support restoring to a specific line number.

DIM variable {'(' <dim_0> {,<dim_1>}* ')' } {, variable {'(' <dim_0> {,<dim_1>}* ')' }}*
(where <dim_n> is a numerical expression).
Reserve space for a simple variable or array. Up to 255 dimensions may be defined for an array variable. The DIM statement can also define simple variables. Arrays can be Integer, float or string types.
The original Level 2 BASIC would generate an error if an array was DIMed more than once. This BASIC will not generate an error, but the previous array will be deleted.

INPUT {String_Constant_Prompt ';'} Variable {, Variable}*
Print the string constant prompt if supplied, then accept console input for the specified variables separated by commas. If the input is a string, quotes are optional unless the input contains a comma. An error occurs if the variable type doesn't match the input.

LINE INPUT {String_Constant_Prompt ';'} String-Variable **(NEW)**
Print the string constant prompt if supplied, then accept all console input as a single line and store it in the String-Variable. Commas, quotes and all other puctuation are accepted as string characters and do not delineate fields. Spaces are not trimmed.

LET Variable '=' <expression>
Evaluate the expression and assign the value to the specified variable. An error occurs if the variable type doesn't match the expression. The LET keyword is optional on input, but will be added automatically by the tokenizer.

PAUSE <Numerical_Expression>
Pauses BASIC execution for the specified number of milliseconds.

PLOT <X_Coordinate> ',' <y_Coordinate>
UNPLOT <X_Coordinate> ',' <y_Coordinate>
Turns on or off a pixel at the given coordinates.

REM {Text}
Creates a comment to the end of the line. An apostrophe may also be used as a shortcut for REM.

PRINT {'@'<location> ','} {Item_List}
(Where {Item_List} is {{String_or_Numeric_Expression} {';' or ','}}*)
Evaluates the expression and Prints the string or numeric value on the screen. An optional '@' sequence at the beginning will change the starting print position to the screen buffer index supplied. A ',' generates a tab and a ';' is the default separator. A PRINT without a ',' or ';' at the end will generate a newline. A '?' is shorthand for “PRINT”.
The AT() and TAB() functions may be used anywhere a string can be used and will move the

current printing location accordingly.

PRINT {'@'<location> ','} USING <Format_string_expression> ';' <variable> {,Variable}*

Prints formatted text on the screen. An optional '@' sequence at the beginning will change the starting print position to the screen buffer index supplied. The Format string can contain literals or special formatting characters.

The formatting characters are divided into three groups: Literals, Strings, and Numericals. There may be multiple fields of varying types in a format string. An error is generated if at least one variable is not specified. If the variable type does not match the field, then a TYPE CONFLICT error is generated.

String format characters are: '!', '%' and '&'. The '!' prints the first char of a string. The '%' defines a fixed size string field. The total number of characters occupied by a "% %" field (including the '%' characters themselves) define the string field size. The '&' defines a variable sized string field.

Numerical fields use '$', '-', '+', '*', ',', '.' and '#' format characters. See the TRS-80 manual for a more detailed description of numerical field descriptors. In short, '#' defines a numerical field digit, '$' and / or '*' cause those characters to print at the beginning of the field. The '+' character forces the sign to print even if its positive. The '-' character can be placed at the end to move a normal sign to the right of the number. The decimal point fixes its position in a numerical field. And finally, the comma to the left of the decimal causes commas to print as thousands separators.

Literal fields are virtually all other characters. A non-literal character may be converted to a literal by prefixing it with a '_' (underscore).

Operators and Precedence

Operator precedence works pretty much just like in any other BASIC. All operators use Left to Right association except those specifically labeled Right to Left Association.

Precedence	Operator	Associativity
23	&	Left to Right
22	)	Left to Right
21	; (semicolon)	Left to Right
20	: (colon)	Left to Right
19	^	Right to Left
18	NOT	Right to Left
17	-Unary Minus	Right to Left
14	*, /, MOD	Left to Right
12	+, - (Binary minus)	Left to Right
6	= (compare), >, <, <=, >=, <>	Left to Right
5	AND	Left to Right
4	OR	Left to Right
3	(	Left to Right
2	, (Comma)	Left to Right
1	= (Assignment)	Left to Right