**My** Tiny BASIC Extended

**(mytb)**

Version 1.1 IRQ/TASKING Support

bob@corshamtech.com

Extended by JustLostIntime@yahoo.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.

Now supports both upper and lower case characters for commands and variables.

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:

https://github.com/CorshamTech/6502-Tiny-BASIC

For the Extended version supporting IRQ and Task extensions:

<https://github/JustLostInTime/em6502>

This version of not so tiny basic is useful to learn about multi threaded and multi tasking system and the basic functionality they provide. Besides it lets you run multi task programs on your Corsham CT6502 SS-50 system and Kim system with at least 32K.

Licensed under GNU GPLv3.

# Numbers and Variables

There are 26 integer variables named A to Z.

Variables may be subscripted var[<expr>]. Any Variable may be subscripted up to end of the variable SET. For example A may have 1 to 26 subscripts representing A-Z in the variable set. B may have subscripts 1 to 25 representing b to z and so forth up to Z which may only have a single subscript 1 . Subscripts start at 1.

Each Task has its own Variable set A-Z. This version is not so small.

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

Special Variables:

PID - Represents the Process ID of the current task

TRUE Represents the value -1 or hex $FFFF

FALSE Represents the value 0

TASK[Parameter-Index-expression]

Represents the parameter passed to a task.

See Task Section for details

# Expressions/Functions

## ABS(<number>)

Returns the absolute value of the number.

## FREE()

Returns the number of free bytes for user programs.

## CALL(<Address expression>,<Value Expression>)

Call a system function with optionally passing a value in Accumulator.

The Call returns what ever is in the Accumulator when the system function returns.

## GETCH()

Returns the next character from the tty keyboard.

## RND(<upper limit>)

Returns a random number from 1 to limit. If limit is not specified, it is set to 32767

## PEEK(<address expression>)

Returns the value at the specified location. Treats address value as unsigned.

# Commands

## CLS

Clear the screen by sending the ANSI ESC[3J sequence

## DIR

Lists the content of the disk.

## END

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

## ERASE <File Name>

Delete file from the disk.

## 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>[:<statement>]

If the expression evaluates to a non-zero value (TRUE) then the statements following THEN will be executed. THEN Keyword is optional.

## INPUT [prompt string ; ] <variable> [,<variable> …]

Prints a question mark, gets the user’s input, converts to a number, then saves the value to the specified variable. If a string follows the keyword then it is printed as a prompt. If the variable ends with a $ then a single character is read from the input.

Example :

input "Enter a letter",a$

This will read a letter from the keyboard and stores the value in a

## [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.”. Let is not required when assigning values to a variable. If the subscript form of a variable is used then LET is required.

## 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. Filename must match the case on the directory listing.

## NEW

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

## POKE <address-expression>, <byte Value expression>

Sets the memory address to the specified byte value

## [PRINT | PR | ?] <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.

Using the ? Reduces the size of the program and speeds execution.

Print by its self prints a CR LF

A comma or semi colon at the end will not output the CRLF.

If an expression starts with a $ then the value is output as hex.

If $ trails an expression the value is written as a character.

## PUTCH <expression>

Put a character to the output device. Range is 0-255

## 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 | RET]

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.

# TASKS and TASK MANAGEMENT

Time Sliced Circular scheduling multitasking is supported by Tiny Basic. There are 10 available task entries , The Main Task always uses the first entry leaving 9 available entries for user tasks. The following TASK management commands and functions are available.

Time slices are set by default to 200 IL instructions.

## KILL <Task PID – expression>

Kills the task specified by the expression should be the value returned by TASK() when a task is started.

## STAT(<Task PID – expression>)

Returns the 0 if the task has stopped, 1 otherwise.

## TASK(<Line Number expression>[,Parameter-expression]...)

As a command This creates a new task starting at the specified line number.

As a function it returns the PID of the new task.

## TASKE[(<Exit value-Expression>)]

This may be used within an executing task to end task. If used within the MAIN Line It acts the same as and END statement.. The exit value is optional and is stored in the tasks Math Stack after the task exits.

## TASKN

This Release the rest of the tasks time slice to the system. Execution of the task continues at the next statement when the task receives another time slice.

## TASKW(<Task PID Expression>[,<Task PID Expression>]...

Wait for a task or group of tasks to complete.

## TASKT(Task PID-Expression,MessageValue-Expression)

Transmit a message to a running task. Adds the message to the Message Queue and the Receiving task must issue a TASKR to receive the message.

## TASKR(Wait-Nowait-Expression)

Wait-NoWait is a boolean expression True = wait

Receives an inter-task message and returns the value or 0 if no wait and nothing received. So don't send a message value of zero! The message may be any other integer value.

## Task Specific variables

PID

Is the PID of the current task.

TASK[Parameter index-expression]

This is the parameter from the parameter list passed when the task was started. Basically the parameters are pushed onto the math stack when the task is started. So the stack size and the need to do math limit the number of parameters that can be passed. No checking is done...So be careful. Parameter index start at zero.

Example: a = task[0] : b= task[1]

These values are read only and may not be changed.

# Task Support Implementation Description.

Tasks are implemented in the IL interpreter and are really a crude form of tasking. Allowing each task a number of IL instructions before the Task is suspended and the next task is started. Task me to some degree be cooperative and issue a task next command to release the remainder of their time slice.

Each task is allocated a memory area consisting of:

1. 26 private variables A-Z 52 Bytes
2. Math stack of up to 20 entries 40 Bytes
3. Gosub/For-next Stack 16 entries 64 Bytes
4. IL Interpreter stack 20 entries 40 Bytes
5. Pointers for each stack 3 Bytes 03 Bytes
6. Basic Application Instruction Pointer 02 Bytes
7. Basic Application Index Register 01 Bytes
8. Math Work Registers R0,R1,MQ 06 Bytes
9. Indirect Pointers 3 06 Bytes
10. Total 214 Bytes

There are ten Task slots Allocated by default in the provided Source Code. This can be altered by the user. Therefore a total of 2140 bytes are required to support multitasking out of the box. A lot of area for some machines.

The interpreter occupies 6K of memory so for a useful Multi tasking system a minimum of 16K is required, Prefer 32 to 48K. Corsham's 6502 ss-50 system comes with 64K. They also sell memory upgraded for the KIM systems and the Rockwell systems.

This system is a good educational tool. And practical for small projects requiring multiple tasks. It is useful to explore Tasking using the em6502 emulator.

# IRQ and IRQ MANAGEMENT

## IRQ <line number -expression>

Enables the interrupts and Sets the line number to go to when an IRQ is received.

IRQ's are disabled until the IRQ subroutine completes with a ireturn statement. Setting a line number of zero stops the IRQ requests and disables interrupts.

## [IRETURN | IRET]

Returning from an interrupt service routine. Enables the IRQ interrupt.

# 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

10= File Not Found

11=Gosub Stack – underflow, too many returns

12=Gosub stack – overflow, to many nested gosub statements

13=Bad Line Number specified, not found

14=Unable to create new task, no more slots

15=Array Subscript out of range

16=Invalid Task PID provided

# Multi Statement lines

A colon may be used to place more than one statement on a line. Any line starting with an if statement will only execute the remainder of the line if the expression is true. Even when containing a : and more statements.

<statement>[:<statement].. as many statement than can fit in 132 characters per line

# 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. An old trick was to have a GOTO at the start of the program which jumps to a very high line number which does the initialization.
* 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.