My Tiny BASIC Extended

Concurrent Tasks,Irq Handling , Tokenized IL

(mytb)

Version 1.1.14 IRQ/TASKING/IPC 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, Concurrent 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:

Original tiny basic @ https://github.com/CorshamTech/6502-Tiny-BASIC

For the Concurrent version supporting IRQ and Task extensions, Tokens:
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.

Numbers and Variables

There are 26 integer variables named A to Z. And An exit code Variable ^

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.
Tasks may access other tasks Variables with the following notation
<PID-Expression>!<Variable-Name>[Subscript] . Subscript is optional.
A tasks exit code is available using PID!^
Examples:

1. Access Tasks Variable A : a = task(1000) : ? a!a
2. Access Tasks EXIT code: a = task(1000) : taskw(a): ? a!^

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.
But may be printed as unsigned values using a % before the value to be printed.
Special Variables:

PID	-	Represents the Process ID of the current task. Currently some multiple of 25.
TRUE		Represents the value -1 or hex $FFFF
FALSE		Represents the value 0

Parameters Passed to a GOSUB or TASK may be accessed using the # variable
Example:

1. Access parameter 0 : ? #[0]
2. " " 1: ? #[1]

See Task Section for details

Multi Statement lines

A colon may be used to place more than one statement on a line.
Any line starting with an if statement execute all then and following statements on that line until CR is reached.
Examples:
<statement>[:<statement] -

Any line may contain any number of statements

if <expression> [then] <statement>:<statement>…

When true everything after the THEN is executed until the end of line is reached.
When false then move to next line of code.
As many statement as fit in 132 characters are permitted.

Putting as many statements as make sense on each line will significantly improve performance of your application.

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

DEC <Variable-Name>
Decrement variable by 1. Considerably faster than a=a-1

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 <(Line Number-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.
Example:
Goto(1000) – everything within the brackets is an expression returning a line number.

GOTO <Valid-Line-Number>
This type of goto is pre-computed just before the program executes. It does not search for the line number during execution. It has a direct memory transfer.
Example:
 Goto 1000 – this must be a valid line number. It is compiled to a direct memory transfer address just before the application executes.

GOSUB <(Line Number-expression)>[(Parameter 1, ...)]

GOSUB <Valid-Line-Number>[(Parameters 1, ...)]
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.
Parameters are passed on the Stack. So are limited by the size of the stack which is also used for math. Currently 20 entries deep.
As with the goto the second form is pre-computed just before the program begins execution.

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.

INC <Variable-name>
Increments the variable by 1. Considerably faster than a=a+1

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. Let is not required when assigning values to a variable.

LOAD <"filename">
NOTE: from Version 1.0.4 Quotes are required.

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. NOTE: Quotes are not used to enclose file names.

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 an expression starts with a % then the value is displayed as an unsigned 16 bit value.
If $ trails an expression the value is written as a character.

Examples:
? free() , %free(), $free() ----→ outputs : -22344 43192 A8B8

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][(Return Value expression)
Will return to the next statement following the GOSUB which brought the program to this subroutine. Also used by tasks see Task Section.

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

SAVE <"filename">
NOTE: from Version 1.0.4 Quotes are required.

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

Trace <Switch value>

1. %10000000 - IPPC trace the core VM
2. %01000000 - Basic program trace
3. %01000001 – Interactive Basic program trace

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 512 IL instructions.. See SLICE

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.

SLICE <Time-Slice-Count Expression>

Defines the number of ticks for each time slice used by the task manager. This defaults to 512.

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 context after the task exits. This is synonymous with the use of return(exit code-expression). The exit code is accessed using the special ^ variable.

1. Pid-expression!^

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.

Task Specific variables

PID

Is the PID of the current 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 = #[0] : b= #[1]

These values are read/write and may be used as local variables.

Inter-process communication

Inter process communications is supported by the system.

ipcs(<message-expression>,<task PID-expression>)

Send a msg to another task

Write messages to the ipc message queue of another task

On Return - True-good or False-failed

The message may not be sent if queue is full.

Currently 10 entries are available but this is shared by the Gosub stack.

Example :

On Main task : `b = task(1000) : a = ipcs(100,b): ? a`

On Task B : `a = ipcr(b) : ? "Msg From "a;" Mgs is ";b`

ipcr(<variable name>)

Read messages from the IPS message queue

Returns	message value from message queue a message -1 is reserved meaning no entry found The provided variable contains the pid of the sending task. This is optional. This always waits for a message before returning.

Ipcc()

Check the message Queue for messages and return the count

Returns	Number of messages waiting

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.

Task 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.

Task Control Block Definition

1.	27 private variables A-Z,^	54 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,R1	07 Bytes
9.	Indirect Pointers 3	06 Bytes
10.	Total	216 Bytes

Context Control Block Definition

1.	VARIABLES	2 bytes	pointer to, 26 A-Z
2.	ILPC	2 byte	IL program counter
3.	ILSTACK	2 byte	IL call stack
4.	ILSTACKPTR	1 byte	Pointer ti current entry
5.	MATHSTACK	2 bytes	MATH Stack pointer
6.	MATHSTACKPTR	1 byte	Pointer to current stack position
7.	GOSUBSTACK	2 bytes	pointer to gosub stack
8.	GOSUBSTACKPTR	1 byte	current offset in the stack, moved to task table
9.	MESSAGEPTR	2 bytes	Pointer to active message, from bottom of il stack
10.	CURPTR	2 bytes	Pointer to current Basic line
11.	CUROFF	1 byte Current offset in Basic Line	
12.	R0	2 bytes	arithmetic register 0
13.	R1	2 bytes	;arithmetic register 1
14.	MQ	2 bytes	used for some math
15.	R2	1 byte	General purpose work register(tasking)
16.	Total	25 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. Or the Corsham products.

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
17=Out of space on queue to send new message
18=The expect Stack frame was not found.

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.

Example programs

Example Task program

```
10 print "Simple task test, to activate all tasks",r
15 a = 2000 : b= 10 :c=1
20 x = task(1000): ? "PID = ";x
30 y = task(2000): ? "PID = ";y
40 z = task(3000): ? "PID = ";z
50 k = task(4000): ? "PID = ";k
60 l = task(5000): ? "PID = ";l
70 m = task(6000): ? "PID = ";m
80 n = task(7000): ? "PID = ";n
90 o = task(8000): ? "PID = ";o
100 p = task(9000): ? "PID = ";p

110 taskw(x,y,z,k,l,m,n,o,p)     : Rem wait for all the tasks to finish

191 print "End of loop ",r
195 r = r + c : if r < a goto b
200 print "Test complete"
210 end

1000 print "Begin new task 1000 PID=";PID
1005 a = 0 : b = 20 :c = 1040 : d= 1010 :e=1
1010 a = a + e
1020 if a > b then goto c
1030 goto d
1040 print "End of task 1000 ";PID
1050 taske

2000 print "Begin new task 2000 PID=";PID
2005 b = 0 : s =20 : c=2010 :d=1: a = 3
2010 b = b + d
2020 if b < s then goto c
2040 print "End of task 2000 ";PID
2050 taske

3000 print "Begin new task 3000 PID=";PID
3005 c = 0 : a=20 :b=3040 :d = 3010 :h=1
3010 c = c + h
3020 if c > a then goto b
3030 goto d
3040 print "End of task 3000 ";PID
```

```
3050 taske

4000 print "Begin new task 4000 PID=";PID
4005 d = 0 :e=20 : f=4040 : g=4010 : h =1
4010 d = d + h
4020 if d > e then goto f
4030 goto g
4040 print "End of task 4000 ";PID
4050 taske

5000 print "Begin new task 5000 PID=";PID
5005 e = 0 :f = 20 :g=5010 : h = 1
5010 e = e + h
5020 if e < f then goto g
5040 print "End of task 5000 ";PID
5050 taske

6000 print "Begin new task 6000 PID=";PID
6005 f = 0 : a=1 :b=10000 :c=20:d=6010
6010 f = f + a
6015 gosub b
6020 if f < c then goto d
6040 print "End of task 6000 ";PID, q
6050 taske

7000 print "Begin new task 7000 PID=";PID
7005 g = 0 : a=1 :b=10000:c=7010:d=20
7010 g = g + a
7015 gosub b
7020 if g < d then goto c
7040 print "End of task 7000 ";PID, q
7050 taske

8000 print "Begin new task 8000 PID=";PID
8005 h = 0 : a = 1 :b=8010 : c=20 : d=10000
8010 h = h + a
8015 gosub d
8020 if h < c then goto b
8040 print "End of task 8000 ";PID, q
8050 taske

9000 print "Begin new task 9000 PID=";PID
9005 i = 0 : j = 1 : k = 9010 : a=20 : b = 10000
9010 i = i + j
9015 gosub b
9020 if i < a then goto k
```

```
9040 print "End of task 9000 ";PID, q
9050 taske

10000 Rem gosub test
10010 q = q + 1
10020 return
```

Sample IPC program

```
10 REM test ipc is working
15 cls
20 a = Task(2000) : b = task(3000,a)
30 taskw(a,b)
40 ? : ? : print "Test completed  a's exitcode =";a!^; "  B's exit code = ";b!^
50 end

2000 Rem this task will wait for a message
2005 print "PID : ";PID, "Waiting for message" : ?
2010 a = ipcr(b)
2020 print "I am PID : ";PID,"Recieved msg : ";a, "From PID : ",b : ?
2030 return(a)

3000 Rem Send a message to another task
3010 a = #[0] : Rem should be the pid start for first task
3020 Print "PID : ";PID, " Sending message to PID : ";a

3030 print "Result of sending msg = ";:if ipcs(300, a) ? "Sent" : return(84)

3040 print "Failed": return(84)
```