STABLE - TUTORIAL

an extreme small an fast FORTH-VM

stable allows you to trace every executed operation. You can show the current instruction index, the current instruction character, the stack depth and the stack Items. To switch on the tracing mode, you simply enter 3` at the first two characters in your application .

Each character in STABLE represent an operation. Whitespace characters (space, tab, newline, ...) are simply ignored. Comments are not available, write the documentation of your program in a different place..

Lets start with a hello world. As file extension we are using .txt, but you can choose whatever you want, even no extensen at all.
Edit hello.txt and enter

```
"Hello World"
```

Don't forget the double quotes. And now we run the program.

```
$ stable hello.txt
Hello World$
```

As you can see, the prompt follws immediately the output, which is not nice, so we are adding a newline. The ascii code of newline is 10

```
"Hello World"10,
```

If STABLE encounters a number literal this literal will be pushed on data stack. After the literal you see the ,. This character take a number from data stack and send it to the screen, which means 'newline' in this case. Lets try again

```
$ stable hello.txt
Hello World
$
```

Much better.

Next we want to do a litte arithmetik and print out the result. Say (10 + 15). Since we are working with a stack machine, the arguments has to go there at first. So we push 10, then 15 and then we call the + operation. Edit our hello.txt

```
"10 + 15 is "10 15+.
```

First we are printing out the text *10 + 15*, then we push 10 on stack. The push is implizit. Every literal entered this way will pushed on stack. The white space between 10 and 15 is needed to separete the two numbers. So 10 will pushed first and then 15. The notation is (10 15) where 10 is next of stack and 15 is top of stack. With + this two items get added and left the sum on top of stack (TOS). The notation is: + (a b--c) (10 15--25).
With **.** (dot) the top of stack will be sent to display as a decimal number.

```
$ stable hello.txt
10 + 15 is 25$
```

Uups, we forgot the newline

```
$ vi hello.txt
"10 + 15 is "10 15+.10,
$ stable hello.txt
```

```
10 + 15 is 25
$
```

This way we can do even complexer arithmetik. Try the following

```
12 23+5/.10,
=>7
```

(12+23)/5 = 7

The sequence .10, is needed many times, let factorize this sequence into a function. In STABLE we can define up to 26 functions named from A to Z. We are using function N for newline.
Open the editor an enter

```
{N.10,}12 23+5/N
```

With **{** we begin to define a function. The first letter gives the function name, N in our case. Followed by the sequence of commands: .10, which we have discussed already. **}** closes the function. Now we can call it simply by writing N.

Lets make a litte bit more complex sample

```
{N1+#.10,}0NNNNN
=>
1
2
3
4
5
```

With the # operation the top of stack (TOS) will be duplicated. For this example the stack effect picture will be

```
0 ( 0)
N
  1  ( 0 1)
  +  ( 1)
  #  ( 1 1)
  .  ( 1)
  10 ( 1 10)
  ,  ( 1)
N
  1  ( 1 1)
  +  ( 2)
  #  ( 2 2)
  .  ( 2)
  10 ( 2 10)
  ,  ( 2)
...
```

There is one thing with our litte application. After the last N, 5 will be left on stack. To keep the stack clean, after the last call to N we should drop this value with the operator \

```
{N1+#.10,}0NNNNN\
...
N ( 5)
\ ( )  stack empty
```

We are now making our first loop. Print out Hello World 5 times.

```
{N10,}5[#." Hello World"N1-#]\
=>
5 Hello World
4 Hello World
3 Hello World
```

```
2 Hello World
1 Hello World
```

The while operator is **[**. If the TOS is true (every number not 0) the loop will be entered. If the TOS is false, the loop will not be entered. At the end of the loop (**]**) the loop will be repeated while TOS is true, or ended if TOS is 0. The **]** operator is dropping the TOS value. Note that the loop condition flag remains (0 in any case) on stack and must be dropped (as you can see in this example by the final \).

Regular conditons (if/else) are built with (. If TOS is true then the code within the brackets will be executed).

```
{N10,}1("TRUE")N
```

(drops the flag in TOS. So if you want to do an if/else condition you have to duplicate TOS first

```
{N10,}{T#("TRUE")~("FALSE")N}1_T0T
=>
TRUE
FALSE
```

Note that, other than the while loop, you don't need a drop operation after the condition. The **~** operator binary negates the TOS value. So 0 becomes -1 and -1 becomes 0. Be sure that for **~** -1 or 0 is on TOS. Use compare functions like < > or =. For example 0= compares TOS against 0 and leave -1 or 0 on stack.
See the stack flow

```
1  ( 1)
_  ( -1)
T
  # ( -1 -1)
  ( ( -1)  is executed (-1=true)
  ~ ( 0)
  ( ( )   is not executed (0=false)
0 ( 0)
T
  # ( 0 0)
  ( ( 0)  is not executed (0=false)
  ~ ( -1)
  ( ( )   is executed (-1=true)

```

you can singlestep as you can see stable.tutorial.mp4. For single stepping you need the debug version of stable (stable_debug.c).

STABLE provides us with plenty of registers, named from a to z (you see lower case letters for variables, upper case letters for functions). These registers could be used universaly. While startup, the registers will be filled from the argument line. With : (fetch register) and ; (store register) you can access the values of any register. a; is fetching the content of register a and pushing it on stack (stack comment is a; (--a)). a: stores a value from stack into register a. (stack comment is a: (value--). Show following example:
in hello.txt write

```
"The value of a is "a;." and the value of b is "b;.
```

Then, on command prompt

```
$ stable hello.txt 10 20
The value of a is 10 and the value of b is 20
```

With conditional operators

```
"The value of a ("a;.") is " a;b;>#("greater")~("less or equal")\" then b ("b;.")"10,
```

Then, on command prompt

```
$ stable hello.txt 10 20
The value of a (10) is less or equal then b (20)
$ stable hello.txt 20 10
The value of a (20) is greater then b (10)
$ stable hello.txt 20 20
The value of a (20) is less or equal then b (20)
```

More advanced sample

```
{A"The value of a is "a;." and the value of b is "b;.10,}
Aa;b;a:b:A
```

```
$ stable hello.txt 10 20
The value of a is 10 and the value of b is 20
The value of a is 20 and the value of b is 10
$
```

A very common operation is to increment an decrement registers without involving the stack. x+ (x immediatly following by +) increments x by 1. x- decrement x by 1.
Note: while you can say: a;#: and : is addressing register a this is not valid with +. Because + is either the operator for addition or the increment operator for a register. So in a:+ the **+** is the normal add operation.

```
{A"The value of a is "a;." and the value of b is "b;.10,}
Aa+a+a+b+A
```

```
$ stable hello.txt 10 20
The value of a is 10 and the value of b is 20
The value of a is 13 and the value of b is 21
$
```

Registers can be used as address register where you can fetch or store values from or into memory. To do that you have to set a memory address into the register. Then, with ? you can fetch the memory from the address given in the register, with ! you can store a value into an address gigen in register.
Address from 0..29 are reserved (0 is the address of register a, 1 is the address from register b and so forth).

```
{A"The value of a is "a;." and the value of b is "b;.10,}
0x:1000x!A
1x:x?0x:x!A
```

With 0x:1000x!A we select address 0 in register x (0x:) (which addresses register a) and store the value 1000 there (1000x!). So register a will be set to 1000. Then we call function A to prove the value

With 1x:x?0x:x!A we select address 1 in register x (1x:) (which addresses register b) and fetching the value onto stack (x?). Then we select address 0 in register x (0x:) and store that value into that address (x!). So infact we are reading content of register b and storing that value in register a. we call function A to prove the value

```
$ stable hello.txt 10 20
The value of a is 1000 and the value of b is 20
The value of a is 20 and the value of b is 20
$
```

More advanced memory operation.
As you have seen registers are powerful address registers with auto increment or decrement. If we need memory, we are thinking about the memory layout in first place. Now we want an array for keeping a matrix of 10x10 cells (we are only accessing cells (32 bit wide) not bytes). Since the addresses 0..25 are reserved for the register a..z, we start at address 30. From address 30 to 129 we have our matrix, at address 130 there can be the next memory segment if needed.
If we want to access the cell at x=4/y=6: and add the value 4711 to that cell

```
6 10*4+     offset to the cell
30+         base address of array added
r:          address into register r
r?          get value of that cell
4711+       add 4711
r!          and store the new value into the cell
```

Since entering lower letter a..z selects a register, we can make multiple register operations without reselecting it again. Our code becomes smaller then (see missing r 4. and 6.):

```
6 10*4+     1. offset to the cell
30+         2. base address of array added
r:          3. address into register r
?           4. get value of that cell
4711+       5. add 4711
!           6. and store the new value into the cell
```

To select our cell in register r we can write a small function (lats take R) for that. The stack diagram: R (x y--) . register r is selected than

```
{R10*+      Function R 10 ( x y 10) * ( x y*10) + ( x+y*10)
30+         30 ( x+y*10 30) + ( x+y*10+30)
r:}         select register r and set address into r
6 4R?       fetch cell at x=6 y=4
4711+       add 4711 to the value
!           store new value back into array
6 4R?.10,   write content of the new value on stdout
```

We can write that sample slightly shorter

```
{R10*+30+r:}
6 4R?4711+!
```

```
6 4R?"value is ".10,
6 4R?1000-!
6 4R?"value is ".10,
```

```
value is 4711
value is 3711
```

And we can take advance of auto increment register to address more than one cell in sequence.

```
{R10*+30+r:}
6 4R       select register
?10+!r+    add 10 to x=6 y=4, select next cell
?11+!r+    add 11 to x=7 y=4, select next cell
?12+!r+    add 12 to x=8 y=4, select next cell
?13+!r+    add 13 to x=9 y=4, select next cell
           now lets check the content
{P?." "r+} function P (print) for display the value in current register
6 4R       select x=6 y=4 again
PPPP10,    content of the next 4 cells with autoincrement.
10,
6 4R?." "  content with individual addressing
7 4R?." "
8 4R?." "
9 4R?." "
10,
```

```
10 11 12 13
10 11 12 13
```

Since STABLE is a stack engine we have some Stack operations. We already know # (duplicate top of stack), \ (drop top of stack), now we have $ (swap top of stack with next of stack) and @ (copy next of stack to top of stack, called over)

```
The stack diagrams
# ( a--a a)    duplicate 1st item on stack
\ ( a--)       drop item from stack
$ ( a b--b a)  swap 1st and 2nd item of stack
@ ( a b--a b a) over: copy next of stack on top of stack
```

Combine logical operations with 'and' and 'or'. If a comparsion results in true, -1 (all bits set) will be on stack. If a comparsion results in false, 0 (all bits clear) will be on stack. Any value other than 0 will be interpreted as 'true' in if and while conditions. So you can write a down counter simply by:

```
15[#.10,1-#]
```

If you have to combine some logical conditions, you can use | for 'or' and & for 'and'

```
&  ( f1 f2--f2) f2 is true (-1) if f1 and f2 are true
|  ( f1 f2--f2) f2 is true (-1) if f1 or f2 is true
~  ( f--f') if f is true (-1) it will be false (0), if
            f is false (0) it will be true (-1). So you
            can invert the logic
```

At last all arithmetik operations

```
+ ( a b--a+b)  add
- ( a b--a-b)  minus
* ( a b--a*b)  multiply
/ ( a b--a/b)  division
% ( a b--a%b)  reminder (modulo)
```

With ^ you can read one character from keyboard. The character will be delivered as soon as it is pressed.

```
^ ( --key|0)   0==EOF
```

Downloads

- makefile.stable
- stable.c ANSI-C code, modified by Andreas Klimas
- stable_fast.c gcc optimized code, modified by Andreas Klimas
- stable_debug.c gcc optimized code with single step monitor
- stable.ori.c original C code (K&R) from inventor Sandor Schneider
- mandel.st Mandelbrot sample
- fib.txt Fibonacci calculation

... more see STABLE main page