STABLE - COOKBOOK

an extreme small an fast FORTH-VM

I wrote this document based on my current experience and knowledge and how I see the way of stack programming. Improvements and critique are always welcome (klimas@w3group.de).

I believe in the simplicity and power of stack programming. It takes some time to getting used to, but then it is extreme powerful. The stack is there to communicate between functions, so only the arguments should be there. Maybe one or two calculated termporarly values. Stackoperations are only for accessing TOS (top of stack) and NOS (next of stack). You can't access the 3rd element easily. If you are in this situation then rethink about the stack flow. If you have to do stack gymnastics (#\@$) then rethink about you stack usage. Use registers or global memory to store values more permanentely. It is not C (or C like) where all variables are given as arguments.

Interactivity

Since STABLE is a very tiny system interactivity becomes very different and quite nice. Instead of entering long commands and buffering of a whole command line which will be interpreted, each keystroke should be interpreted immediately. See the video (22M).

For an example you can look at block 99 which contains an obvious implementation of a command shell. I will comment it here. For more information see block 0 which contains an index for all blocks, block 8 for Terminal ANSI sequences and block 9 for the instruction set documentation. **Note** that the following code is slightly older, blocks are code blocks, data blocks (persistence) is missing here.

```
In variable k we are holding numbers entered from
the keyboard. We try to keep k as long as it is possible but a
loaded program might change the value to be invalid, so we check
the validity of k.
k           select k as current variable
;0<(0:)     trim k to zero if it is less then 0
;1000>(0:) trim k to zero if it is greater then 1000

Enter endless loop
1[
        k  again, select k as current variable

        Display the menu. For a hierarchy of menues put
        each submenu into a different block and load them
        13,"esc=retry q=quit l=load e=edit c=cls, h=home(block 0)"10,
        {E27,"[K"}

        This loop is reading and displaying the current
        block number from keyboard, until a non number is pressed. The
        number built so far is stored in variable k.
        The key which leads to an exit is on top of stack.
        1[\13,k;."> "E^#47>@58<&(48-;10*+:0)#0=]

        #27=(0:)     if last key was ESC reset k to 0
        #`l=(;8')    load (and run) block stored in k
        #`e=(;7')    edit block
        #`h=(0#:7')  set block to 0 and edit it
        #`q=("quit"10,0 11') quit STABLE
        #`c=(27,"c") clear screen (ANSI sequence)
]
```

logical operators

STABLE does not provide logical operators, only binary (and, or, not) 32 bit operators. To combine the result of conditions (< = >) it is important to work with -1 as true (all 32 bits set to 1) and 0 as false.

Mind that you can combine these values with not (~) which reverse true or false. True must be -1 and false 0.

```
        OR  AND
-1 -1   -1   -1
-1  0   -1    0
 0 -1   -1    0
 0  0    0   -1
```

A typical example is to combine a loop counter and a value. The loop is finished if either the loop counter has become 0 or a specific value has occurred. Note that a loop counter not equal zero is not -1 and hence not valid for logical-binary operation. To get a valid boolean simply compare it agains zero (0=), then you get -1 if the loop counter ist zero and 0 if it is not

```
flow1: key is blank, terminate. counter is 0 terminate:      terminate
flow2: key is blank, terminate. counter is != 0 continue:    terminate
flow3: key is not blank, continue. counter is 0 terminate:   terminate
flow4: key is not blank, continue. counter is != 0 continue: continue

1[        flow 1     flow 2    flow 3     flow 4
  ^32=    ( -1)      ( -1)     (  0)      (  0)
  We want to terminate the loop if 32 has arrived, negate the logic
  ~       (  0)      (  0)     ( -1)      ( -1)
  a-;     (  0  0)   (  0  n)  ( -1  0)   ( -1  n)
  0=      (  0 -1)   (  0  0)  ( -1 -1)   ( -1  0)
  We want to exit the loop if the counter is zero. But if not the
  flag must become -1 not the current value of the counter
  ~       (  0  0)   (  0 -1)  ( -1  0)   ( -1  -1)
  &       (  0)      (  0)     (  0)      (  1)
]
flow 4 is the only flow which continues the loop
```

Note this idioms:
0=~ to convert a logical true (!=0) into a binary true (-1). A logical false is also a binary false
~ to invert a binary compare operation

if/else and switch/case conditions

STABLE is a very tiny engine hence conditons are limited and probably don't work the way you would expect. In fact we don't have an else part, only if, so to implement an else part, or a list of options (switch/case) you have to care about the stack flow.

The IF and ELSE part are working together. The IF part has to left -1 on TOS (top of stack). So with ~ (not) the ELSE part will be skipped.

if/else

```
1. the value to check is either not zero => if part, or zero => else part
We don't need the value for the else part, because it is
always zero. So we can save one stack cell.

operation            stack picture not zero    stack picture zero
                     ( value)                  ( 0)
#                    ( value value)            ( 0 0)
(                    ( value)                  ( 0)
   "HELLO"           ( value)                  skipped
   \1_               ( -1)                     skipped
)                    ( -1)                    (  0)
~                    (  0)                    ( -1)
(                    (   )                    (   )
   "BYE"             skipped                  (   )
)

in short
#("HELLO"\1_)~("BYE")
```

2. the value on top must be compared

Here we need one more stack entry, because we need the value
in the else part as well.

```
operation            stack picture cmp true    stack picture cmp false
                     ( val1)                   ( val2)
#                    ( val1 val1)              ( val2 val2)
32>                  ( val1   -1)              ( val2    0)
#                    ( val1   -1  -1)          ( val2    0   0)
(                    ( val1   -1)              ( val2    0)
   @                 ( val1   -1  val1)        skipped
   .                 ( val1   -1)              skipped
)                    ( val1   -1)              ( val2    0)
~                    ( val1    0)              ( val2   -1)
(                    ( val1)                   ( val2)
   "is "             skipped                   ( val2)
   .                 skipped                   (     )
)

in short:
#32>#(@.)~("is ".)
```

3. nested if conditions

Evaluate the next condition only if needed. If one condition evaluates
to false, in an and condition series, we can stop the whole cascade,
letting the value on stack to know which item failed the test. A 0

```
value show that all tests are passed successfully.
Note that the conditions must be in tail position and one closing
bracket is enough. To keep STABLE small ( parses only the next ).

Example:
^#32=(                  (  key1 -1)   ( key1 0)  ( key1 -1)
  \#32=(                (  key2 -1)   skipped    ( key2  0)
    \^#32=("SUPER"\0)   (  key3 -1)   skipped    skipped
                        (  0)         ( key1)    ( key2)

0= for booelan          ( -1)         (    0)    (    0)
```

switch/case Instead of having an else part which inverse the comparation we are doing a comparation for each case

```
operation            stack effekt true  stack effekt false   comment
^                    ( key)             ( key)               read key
#                    ( key key)         ( key key)           dup
27=                  ( key -1)          ( key 0)             is ESC key?
(                    ( key)             ( key)
  #                  ( key key)         skipped
  .                  ( key)             skipped              display key value
)                    ( key)             ( key)
#                    ( key key)         ( key key)           dup
65=                  ( key 0)           ( key -1)            is A key?
(                    ( key)             ( key)
  #                  skipped            ( key key)
  ,                  skipped            ( key)               display key
)                    ( key)             ( key)

in short:
^
#27=(#.)
#65=(#,)
```

```
#`Q=("Letter Q pressed"10,)
\                               drop key from stack
```

switch/case/default

```
^                               read key, push on stack ( key)
#27=("ESC pressed"10,\0)        change value to 0
#65=("Letter A pressed"10,\0)   change value to 0
#0=("Default action"10,)
\                               drop key from stack
```

Working with STRINGS

Strings are not part of STABLE. STABLE didn't know how to read or write bytes either, only integers. Since we know that integers in STABLE are 32 bit wide, we can pack (and unpack) 4 characters in one cell. Lets look how we can achieve that. With a sequence of cells we easily can build a string of any size.

Now we are reading up to 4 characters from input port (stdin). Note that each character would be processed immediately. No [ENTER] key is needed, key will not be displayed.
Stack effekt: (--keys f) if f is true then all 4 chars were read, false (0) otherwise. With this a loop which places all the cells into memory should be simple. For simplicity, I would limit the number of cells to a fixed size.

```
4 nested conditions, observe that we only
have one ) at tail position. This is the only
way in STABLE to nest conditions!
0^#32>($256*+^#32>($256*+^#32>($256*+^#32>($256*+0)0=
```

For outputting the chars
Stack effekt: P (keys--)

```
keys value on top of stack
{Q#255&$256/} extract one byte
{R#(,1_)~()}print out byte if not 0
{P define function P
        QQQ extract 4 bytes from 33 bits cell on stack
        RRRR print out each character
}
P print up to 4 chars, keys top of stack
```

while loops

While loops are slightly different than in other languages. The while loop in STABLE is checking the TOS before entering the loop, without dropping TOS.

```
operation   stack effekt    stack effekt  stack effekt
1000        ( 1000)         (   0)        (  1)
[           ( 1000)         (   0)        (  1)
  1-        (  999)         skipped       (  0)
  #         (  999  999)    skipped       (  0 0)
]           (  999) REPEAT  (   0)        (  0) END LOOP
```

register selection

Register select outside of loop

block (persistent data)

When STABLE is starting data block 0 will be loaded at cell 1000 automaticly. The blocksize is 1000 cells. When changing the data block number (with n 9') the old block will be written and the new block will be read. If STABLE is exited, the current data block will be written as well.

On the STABLE IDE you can select the block number with <number>b and display the content with dump. Note that the first persistent cell starts at 1000, so you have to enter: <1000>d.

To store values the only way is through registers. In this example we are using register a.

```
100 9'"loading block 100"
1000a:100!"store 100 into first cell of block 100"
```

Now leave STABLE and start it again (or change the block number with the STABLE-IDE, go to block 100 and inspect the value

```
In the STABLE IDE enter
1e  this starts the editor with code block 1
100 9'1000a:100!    enter the code, save and exit editor

Back to STABLE IDE
l          load and run the code block 1
100b1000d  select block 100 and dump at 1000
1000>
1000:   100     0       0       0       0
1005:   0       0       0       0       0
1010:   0       0       0       0       0
1015:   0       0       0       0       0

Switching back to block 0
<ESC>b1000d  select block 0 and dump at 1000
1000>
1000:   0       0       0       0       0
1005:   0       0       0       0       0
1010:   0       0       0       0       0
1015:   0       0       0       0       0

100b1000d  select block 100 again and dump at 1000
1000>
```

```
1000:   100     0       0       0       0
1004:   0       0       0       0       0
1010:   0       0       0       0       0
1015:   0       0       0       0       0
```

See video demonstation (this video show the address with 1024 instead of the newer 1000. The use 1000 based system is more convenient in STABLE) here (12M)

Copy register from and to persistent storage

Tracing

Tracing whole on 3' Tracing whole off 4' Tracing spot 10'