Stack Checking — A Debugging Aid

Abstract

Most programming errors in Forth programs result from words which have an incorrect stack effect, i.e. which actual stack effect is different from the one specified or intended. Rapidly locating these words will result in more robust programs and applications. A technique for adding run time stack checks for some or all words of an application will be explained. This allows stack effects of words to be tested during the design phase. Sample implementation code will be presented.

Keywords: data stack, debugging, checking, run time, stack effect

To produce readable Forth source code it is absolute necessary to add stack comments to the definitions made. Everyone who has experienced the problem to understand Forth programs without or, even worse, with incorrect stack effects will agree on that.

Normally stack comments describe the state of the data stack before and after execution of a word. A commonly used form for stack comments [2] is the following:

```
( items before execution -- items after execution )
```

thus the stack comment is enclosed by parens. The stack items are seperated by spaces and the stack contents before and after execution is seperated by a double hyphen.
The exact form stack comments look like changes slightly from programmer to programmer. Some use **(S** to start the stack comment or --- or --> to seperate the stack contents. Nevertheless all different stack commenting schemes are similar and reflect the same idea.

Normally stack comments are comments in a literal sense. The Forth interpreter skips them, because they contain human readable but not machine readable information.
This gives the programmer freedom to include in comments whatever he or she likes to put in. However, respecting the above form of stack comments allows the compiler to generate stack checking code for words directly from their stack comments.

Often, if a word goes wrong, it will leave the stack unbalanced, i.e the actual changes performed on the data stack contradict the stack comment given. The proposed technique restricts itself to use only stack comment information available about the number of stack items before and after execution. It does not try to abstractly infere semantics of the given stack comment. Thus it will regard

```
: 2dup ( a b -- a b a b ) over over ;
```

and

```
: 2dup ( a b -- b b b b ) over over ;
```

the same, namely **2dup** expects (at least) two items on the stack and will leave two more.

This kind of stack behaviour can be described by two numbers:

1) The number of stack items a given word expects on the data stack. On entry the data stack must contain at least this number of stack items, or else the word can not possibly work right.
   In most cases a violation will indicate improper use of the checked word.

2) The number of items a given word will additionally put on top of the data stack. If the word leaves less items on the stack than it receives this number will be negative.
   On exit of the checked word the stack must contain exactly that more items than it contained on entry.
   In most cases a violation will indicate improper behaviour of the word itself.

In this way the stack effect of some simple Forth words can be described by:

word	stack comment	expects	changes
over	(a b -- a b a)	2	1
nip	(a b -- b)	2	-1
rot	(a b c -- b c a)	3	0
true	(-- f)	0	1

i.e. **over** can be described by 2 (expects at least two items on the stack) and 1 (puts another item on the stack) respectively.

Unfortunately there are some words, e.g. **?DUP** which do not have a fixed stack effect, but change the stack according to the data received. These words cannot be checked by this checking scheme. However a method for explicitly suppressing stack checking is provided.

With this restrictions the generated stack checking code should work as follows:

Part 1: On entry of a word, the stack check will examine the stack depth to assure that a sufficient number of items for this word is on the stack (and issue a runtime error if not). It then will calculate the number of stack items that should be present on exit. It passes this information to part 2.

Part 2: When the execution of the word has finished, the expected stack depth will be compared against the actual stack depth and if different, will also result in a runtime error.

Since it should be possible to switch stack checking on and off without massive changes in the source code, it is reasonable to redefine the commenting word **(** so that it will be able to generate stack checking code.

Stack checking code needs to be generated:

a) If it is desired. It should be possible to selectively generate checking code on a definition by definition basis. It also sould be possible to explicitly suppress stack checking.

b) If the system is compiling. Interpretive comments should never result in stack checking.

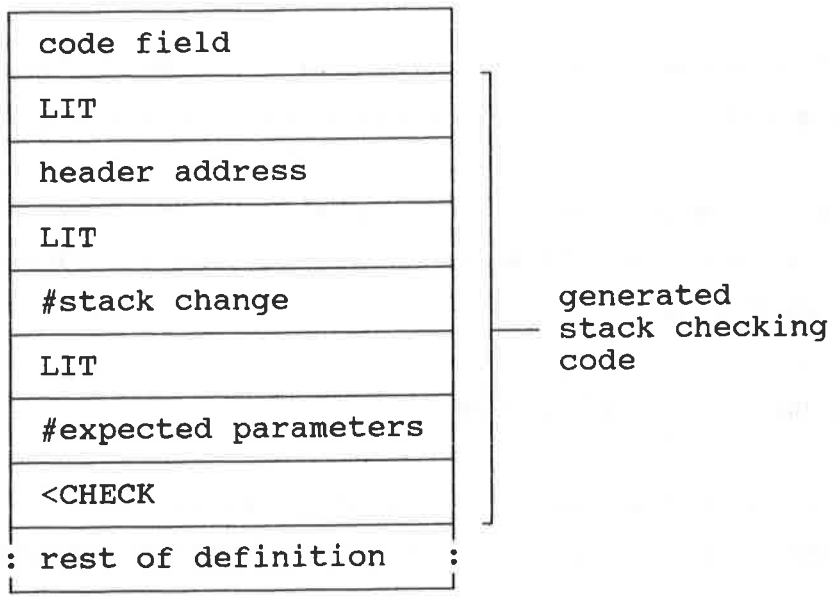
c) If the stack comment is the first comment in a definition. It should be possible to include more comments in the definition of a word than just the stack comment.

A visual marker, say **!!!** should be included in definitions without fixed stack effects to alert the programmer about these potential dangerous words. This marker can be used to explicitly switch stack checking off for the current definition. Anyway it is better to avoid using these words whenever possible!

Implementation

While stack checking is in progress the information about the required stack depth and the assumed stack change is needed. (will have to extract this information from the stack comment and embedd it into the code. If a runtime error will be raised, it should not only report about the fact, that there was a checking failure, but it should also name the word, which causes the problem. So additionally the header address of the executing word is needed during stack checking.

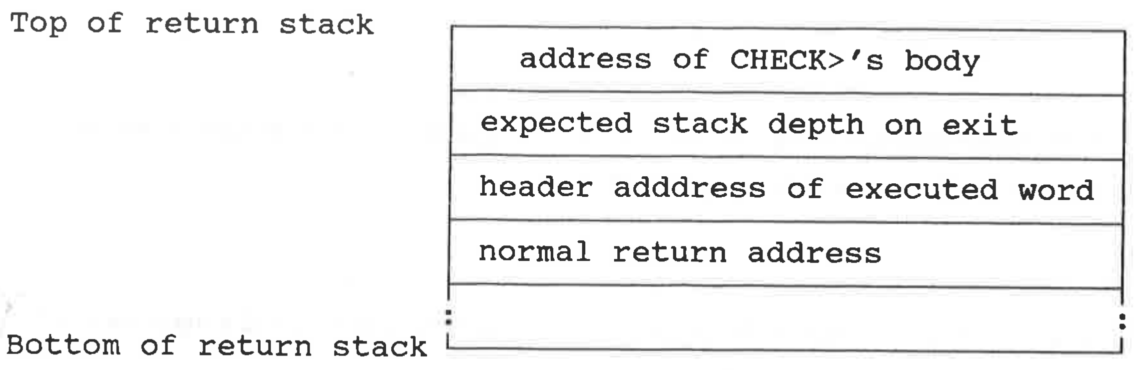
So (should generate code at the beginning of a word like:



The word **<CHECK** should perform the initial stack check (enough arguments available?) and then manipulate the system, so that on exit of the checked word, the second part of the stack check can be made with the necessary information available.

One technique to accomplish this is to build a return stack frame. **<CHECK** then has to put several items on the return stack along with a valid return address, which corresponds to an EXIT-handler (a colon definition).
Later, if the checked word terminates, it will transfer control to the EXIT-handler before returning to the originally calling word.
This technique is thoroughly described in [3].

The second part of the stack check needs to know about the expected stack depth on exit and also, to give a meaningfull error message, the checked word's name. Suppose the second part is performed by an EXIT-handler called **CHECK>** , then an appropriate return stack frame could look like:



CHECK> will receive control, if the checked word will exit. It then can pop the expected stack depth and the checked word's name from the return stack an raise a runtime error if the actual stack depth does not match the expected one. If the stack check is succesful the control flow is transfered to the word which originally called the checked word.

Conclusions

The proposed stack checking technique allows Forth words to test their stack behaviour on the fly. During software development this could be a valuable tool to detect programming errors, although there is a runtime overhead involved.
The proposed code should easily be ported among different systems. However the code expects the Forth system to be a pre-incrementing system, which puts only a single item on the return stack for nesting.

Bibliography

[1] Cole, Barry A., "A Stack Diagram Utility", Forth Dimensions Vol.3 No.1 p.23ff, Forth Interest Group, San Jose, 1981

[2] Brodie, Leo "Thinking Forth", p.151ff, Prentice Hall Inc., Engle Wood Cliffs, 1984

[3] Schleisiek, Klaus "Error Trapping and Local Variables", 1984 FORML Confererence Proceedings pp. 359-64, Forth Interest Group, San Jose, 1984

Glossary

!!! (--)

Used in the form

```
: name ( items before -- items after ) !!! ... ;
```

to explicitly avoid generating of stack checking code. This is necessary if the defined word is known to have a strange stack effect such as **?DUP**.

((--)

generates stack checking code if **check** is true, if the system is compiling and if the comment is the first stack comment in the current definition.
Otherwise ignores text upto the next ")".

,LIT (n --)

compiles N as a literal value into the dictionary.

<check (name #chge #before --) (R ret -- name #after check> ret)

performs the first part of the stack check. It reads the number of expected data stack items #BEFORE from the stack and compares it to the actual stack depth. If there are not enough stack items **<check** will display the word's name NAME and raise the runtime error "needs more arguments!".
Otherwise it will calculate the expected stack depth on exit of the checked word. This can be deduced from the actual depth and the known stack change #CHGE. **<check** then constructs a return stack frame to transfer the necessary information to **check>** after the checked word has finished execution.

check (-- addr)

is a Variable which controls the stack checking mechanism.
If **check** is true (-1), it will activate stack checking. The stack comments of subsequent colon definitions will generate stack checking code.
If **check** is false (0), it will deactivate stack checking. Stack comments will be ignored by the Forth compiler and interpreter.

check> (--) (R name #after --)

performs the second part of the stack check. It reads the expected stack depth #AFTER from top of return stack and compares it to the actual stack depth. If they are different **check>** will display the word's name NAME and raise the run time error "has incorrect stack effect!". Otherwise it will remove NAME and #AFTER and will transfer control to the originally calling word.

ins (-- n)

counts the number of stack items in front of the "--".

is)? (str -- flag)

FLAG is true, if STR is the string ")", otherwise false.

is--? (str -- flag)

FLAG is true, if STR is the string "--", otherwise false.

last-body (-- body)

puts the body address of the last defined word on the stack.

last-name (-- name)

puts the header address of the last defined word on the stack.

outs (-- n)

counts the number of stack items in front of ")".

Source Code

The existing code assumes that:

- a single items is stored on the return stack for nesting.
- the system is pre-incrementing, so a word can be executed by pushing its body address on the return stack and then performing EXIT.

Most of the words used can be found in the FORTH-83 standard document with the exception of:

- .NAME	(nfa --)	Prints a name from a given header address.
- LAST	(-- addr)	Is a variable which contains the header of the latest defined word.

Used words not required by the FORTH-83 standard:

- BL	(-- n)	controlled reference word set
- NAME>	(nfa -- cfa)	experimental proposal
- ASCII	(-- char)	uncontrolled reference word set

```
File CHECK.SCR   Scr 2
( Runtime Stack Checking  1/2                      uh 28sep91 )
Variable check   -1 check !

: check> ( -- ) ( R name #after -- )
   depth r> = IF r> drop  EXIT THEN
   r> .name ." has incorrect stack effect!"  abort ;

: <check ( name #chge #before -- )
         ( R ret -- name #after check> ret )
   depth 3 - >
     IF  swap .name ." needs more arguments!" abort  THEN
   depth 2 - +
   r>  rot >r  swap >r  [ ' check> >body ] Literal >r  >r ;

: last-name ( -- nfa )  last @ ;
: last-body ( -- pfa )  last-name  name> >body ;
```

```
File CHECK.SCR   Scr 7

    Controls stack checking. CHECK true enables stack checking code
    generation.
    Tests stack depth on exit of a word. During execution of a
    checked word the body address of CHECK> is stored on the return
    stack, so it is executed when the word exits. CHECK> then reads
    the name and the expected stack depth from the return stack.

    Tests stack depth on entry of a word. Reads the checked word's
    name, the expected stack change and the initial stack depth.
    Builds a frame on the return stack if enough arguments available



    Leaves the header address of the latest definition.
    Leaves the body address of the latest definition.
```

```
File CHECK.SCR   Scr 3
( Runtime Stack Checking  2/2                      uh 28sep91 )
: ,LIT ( n -- )  [compile] Literal ;
: !!! ( -- )  last-body here  - allot ; immediate

: is--? ( str -- f )  count 2 =
     over c@ Ascii - = and   swap 1+ c@ Ascii - =  and ;

: is)?  ( str -- f ) count 1 =  swap c@ Ascii ) =  and ;

: ins   ( -- n ) 0 BEGIN  bl word  is--? not WHILE  1+ REPEAT ;
: outs ( -- n ) 0 BEGIN  bl word  is)?  not WHILE  1+ REPEAT ;

: (  ( -- )  check @ IF  state @ IF last-body here =
   IF  last-name ,LIT   ins  outs over -   ,LIT  ,LIT
       compile <check  EXIT THEN THEN THEN
   [compile] ( ; immediate
```

```
File CHECK.SCR   Scr 8

    Compiles literal value from stack into dictionary.
    Undoes last stack checking code generation

    Checks, if the string STR is "--". Leaves flag.


    Checks, if the string STR is ")". Leaves flag.

    Counts the number of items in a stack comment in front of "--"
    Counts the number of items in a stack comment in front of ")"

    Generates stack checking code, if checking desired, compiling
    and first comment in definition.
    Otherwise behaves just like old "(".
```

```
File CHECK.SCR   Scr 4
( Runtime Stack Checking -- examples               uh 28sep91 )

: test ( a b c -- 1 2 )   drop drop drop 1 ;

: t ( a -- b )  dup dup test drop ;

: -dup ( n -- n n | 0 )  !!!
    dup IF dup THEN ;


```

```
File CHECK.SCR   Scr 9


    TEST has incorrect stack effect.

    T is used to call TEST.
    1 T results in  "TEST has incorrect stack effect!"
    Switch off stack checking for a single word.
    -DUP has different stack effect on different input data.
```