TESTING TOOLKIT

PHIL KOOPMAN, JR. - WEXFORD, PENNSYLVANIA

One of Forth's strong points is its support of interactive development and testing. Sometimes, however, interactive testing is not enough. During the development of low-level software for the RTX family, we wanted a method to create a permanent record of test cases for Forth words. This record serves as documentation for users and maintainers. In addition, a full suite of test cases for a program provides a way to be sure that a change in one part of the program does not disturb other parts of the program.

How to Use It

Each test case consists of code that places elements on the data and return stacks, creates and executes a test definition, then verifies that the correct results were placed on both stacks. For example, a test case for the word DUP would be:

The test case can be any sequence of Forth words.

```
DS( 1111 --
RS( --
TEST: DUP ;DONE
-- )RS
-- 1111 1111 )DS
```

The first line of the test case specifies that the data stack input to the test is the number 1111. The second line specifies that no elements are to be placed onto the return stack. The third line creates and executes a temporary Forth word with a body of DUP, carefully handling the data and return stack contents before and after the test. The fourth line specifies that no values should

```
\ Forth testing support
\ By Philip Koopman Jr., for Harris Semiconductor
\   Derived from test code used for the RTX chip family
\   Developed on F-TZ (an F-PC and F-83 derivative) version 3.X11

VARIABLE #STACK -1 #STACK !   \ Saves number of stack elements for testing
CREATE R-SAVE 8 ALLOT         \ Note: F-TZ uses 32-bit return addresses!

: GET-DEPTH  ( ..stack.stuff.. - ..stack.stuff.. )
  DEPTH  #STACK @ --  #STACK ! ;

: DS(    ( -- $BAD1 $BAD2 )
  \ Init RS to -1 so that '-' will know it is a DS input
  \ Uses hex 0BAD1 and hex 0BAD2 as sentinel values for DS
  -1 #STACK !     $BAD1 $BAD2 ;

: RS(    ( -- $BAD3 $BAD4 )
  \ Uses hex 0BAD3 and hex 0BAD4 as sentinel values for RS
  DEPTH #STACK !  $BAD3 $BAD4 ;

: --     ( n1 n2 n3 .. n.n - n1 n2 n3 .. n.n sentinel )
  #STACK @  0<  NOT    IF ( if RS( )  GET-DEPTH   THEN ;

: ?DATA    ( n1 n2 -- )
  = NOT  ABORT" DATA STACK ERROR" ;

: ?RETURN  ( n1 n2 -- )
  = NOT  ABORT" RETURN STACK ERROR" ;

: --     ( --)
  DEPTH  #STACK !  ;

: PERCOLATE  ( r1 n.n .. n1 --  n.n .. n1 r1 )
  #STACK @  ROLL  -1 #STACK +! ;

: )RS        ( r.n .. r3 r2 r.1  n1 n2 n3 .. n.n -- )
  GET-DEPTH   #STACK @
  IF  BEGIN  PERCOLATE  ?RETURN   #STACK @ 0= UNTIL  THEN
  $BAD4 ?RETURN  $BAD3 ?RETURN  -1 #STACK !  ;

: )DS        ( r.n .. r3 r2 r.1  n1 n2 n3 .. n.n -- )
  GET-DEPTH   #STACK @
  IF  BEGIN  PERCOLATE  ?DATA     #STACK @ 0= UNTIL  THEN
  $BAD2 ?DATA    $BAD1 ?DATA    -1 #STACK !  ;

: REVERSE    ( n.1 n.2 .. n.n  n  --  n.n .. n.2 n.1 )
  DUP 0> IF  0 DO   I ROLL   LOOP  ELSE DROP THEN ;

: INIT-TEST  ( ..DS.stuff..  ..RS.stuff..  -- ..DS.stuff.. )
             ( RS:   -- ..RS.stuff.. )
```

be left on the return stack, and generates an error message if this is not the case. The fifth line specifies that two values of the number 1111 should be returned from the test, again generating an error message if this is not the case. It is very important that the test cases be written in exactly this order, with no missing items, for proper operation.

The body of the test case between TEST: and ;DONE can be any sequence of Forth words, including primitives that manipulate the return stack. The words INIT-TEST and FINISH-TEST are automatically compiled with the test case to handle the data and return stacks for proper execution.

In order to be sure that a word is working properly, it is not enough to simply place the required number of parameters on the stack and then see if the correct results are returned. The problem is that a word may cause unexpected side effects (such as corruption of elements on the data and return stacks) that are not detected immediately. In order to handle this case, the test words place two "sentinel values" onto both the data stack and the return stack, then check to ensure that no corruption has occurred. While side effects are usually not a problem in high-level code, they can easily create problems when dealing with assembly language or microcode word implementations.

Ideas for Further Refinements

The test capability presented here is rather simple, in order to keep the code (somewhat) understandable. Features that could be added to improve its usability include: allowing RS ( ) RS to be optional, so tests that deal only with data stack operations could automatically generate and test return stack sentinel values; more sophisticated error messages that show exactly what is wrong with a stack when an error does occur; methods to ensure that only desired memory locations are modified for words that perform fetches and stores; and methods to ensure that only desired on-chip registers are modified for assembly language definitions.

The code is written for F-TZ, a version of F-PC, developed by Tom Zimmer. F-PC is a descendent of F-83, but allows using a dictionary space of greater than 64K bytes. The code presented should be relatively

(Continued on page 41.)

```
  CR ."  TEST-"
  #STACK @  0<  ABORT" You must specify both DS( and RS(."
  R> R-SAVE !  R> R-SAVE 2+ !          \ Save return address
  #STACK @  REVERSE
  BEGIN  #STACK @  0> WHILE  >R  -1 #STACK +!  REPEAT
  R-SAVE 2+ @ >R  R-SAVE @  >R   ;     \ Restore return address

: FINISH-TEST  ( ..DS.stuff.. -- ..DS.stuff.. ..reversed.RS.stuff.. )
               ( RS:  ..RS.stuff.. -- )
  R> R-SAVE ! R> R-SAVE 2+ !           \ Save return address
  \ Transfer return stack contents onto data stack for later compare
  0 >R
  BEGIN  R> R>  SWAP 1+ >R   DUP $BAD3 = UNTIL
  R> REVERSE
  R-SAVE 2+ @ >R  R-SAVE @  >R         \ Restore return address

  ." -DONE"  -1 #STACK !  ;

\ TEST and DONE use F-TZ specific words to compile a short
\  definition containing the word to be tested, execute that
\  definition, then FORGET it from the dictionary.
\ This borrows a compilation idea from Rick van Norman's RTX test code
CREATE MARKER  4 ALLOT
: TESTER ;
: TEST:   ( --)
   XHERE  2DUP MARKER 2!  PARAGRAPH +  DUP XDPSEG !  0 XDP !
   XSEG @ --  ['] TESTER >BODY  !
   COMPILE INIT-TEST  ] ;


: ;DONE
   COMPILE FINISH-TEST  COMPILE EXIT
   STATE OFF  TESTER  MARKER 2@  XDP ! XDPSEG !  ;
IMMEDIATE

\ Test ROT for proper operation
  DS( 1111 2222 3333 --
  RS( --
  TEST: ROT ;DONE
  -- )RS
  -- 2222 3333 1111 )DS


\ Test >R for proper operation
  DS( 5555 --
  RS( --
  TEST:  >R  ;DONE
  -- 5555 )RS
  -- )DS


\ Any combination may go between TEST: and ;DONE
  DS( 1111 2222 3333 --
  RS( 7777 2222 9999 --
  TEST:  SWAP R> ROT >R  ;DONE
  -- 7777 2222 3333 )RS
  -- 1111 2222 9999 )DS


\ Null test to be sure it works
  DS( --
  RS( --
  TEST:  ;DONE
  -- )RS
  -- )DS
```

(Continued from page 32.)

portable to other 83-Standard Forths, as long as the return-address-save sequences in `INIT-TEST` and `FINISH-TEST` are changed to save and restore only a single return stack element for most other Forths. Also, `TEST:` and `;DONE` should be redefined for use with other dictionary structures.

Interactive testing is important and useful (and, in fact, there is no reason why these tools cannot be used as an interactive testing format). However, once initial testing is done, it is often useful to have a permanent test suite in a consistent and readable format. Portions of many programs are so crucial to system operation that they merit a full validation suite to prove correct operation. At Harris, validation suites are being used on the instruction sets of some of the RTX processors. The tools presented here provide a starting point for creating a validation suite for a variety of applications.

Philip Koopman Jr. is a senior scientist at Harris Semiconductor and an adjunct professor at Carnegie Mellon University. The opinions in this article are his, and do not necessarily reflect the views of Harris Semiconductor.