Implementing the Forth Inner Interpreter in High Level Forth

Ulrich Hoffmann <uh@fh-wedel.de>

Abstract

This document defines a Forth threaded code (inner) interpreter written entirely in high level standard Forth. For this it defines a specific threaded code structure of colon definitions, a compiler from high level Forth to this threaded code and a corresponding inner interpreter to execute it. This inner interpreter can run in a stepwise way and so gives the surrounding environment control of its execution behavior. A real time environment thus might slice the execution of threaded code in small pieces and provide an interactive command shell while still meeting its real time requirements.

1 Introduction

Forth 94 and Forth 2012 define the semantics of many Forth words. As this opens the space for various implementations and optimizations they do not however specify much of the dictionary structure: words are identified by their so called execution token (*xt*) and later executed; given a word name the *xt* can be identified (`FIND`) along with its immediacy status.

For words defined via `CREATE`, the execution token can be transformed into the memory address of its parameters by means of `>BODY`. Standard programs must not assume a specific header structure or the structure of colon definitions, nor must they rely on a specific way the system uses the return stack. Certain programming techniques that are based on such assumptions cannot be expressed in standard programs. That's ok.

Traditionally Forth is implemented as threaded code [2]. The body of colon definitions contains a list of addresses of the words it invokes. There are *primitives* expressed in machine code and *high level definitions* that are defined in threaded code.

This document defines a threaded code interpreter written entirely in standard Forth. It defines a specific threaded code structure of colon definitions. This allows to also define an inner interpreter (traditionally known as `NEXT`) for this threaded code in high level Forth. The inner interpreter defined below can run in a stepwise way so the execution of threaded code can be sliced in small pieces in a real time environment.

The primitives of this threaded code interpreter are all the words that the underlying Forth system defines (be they machine code or

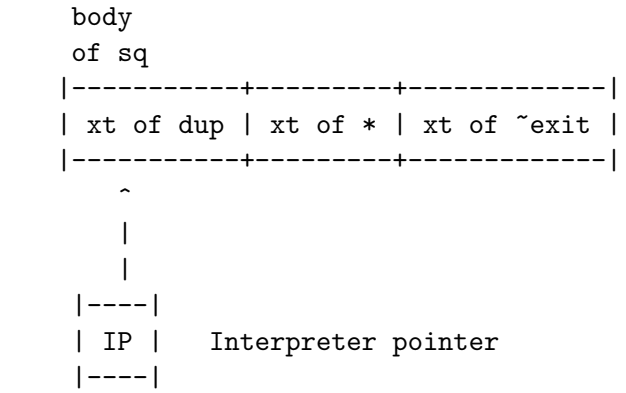

Figure 1: Threaded code of the definition `~: sq ( x -- x ) dup * ~;`

```
 body
 of test
|------------+---+------------+---+---------+-------------|
| xt of ~lit | 3 | xt of ~lit | 4 | xt of + | xt of ~EXIT |
|------------+---+------------+---+---------+-------------|
```

Figure 2: Threaded code for literals `~: test ( -- u ) 3 4 + ~;`

colon words there). Threaded code high level words are defined by a special version of colon (*thread-colon* `~:`). Its definition is given and explained in section 3.

Also, an outer compiler that compiles *to* threaded code as well as an outer interpreter defined *in* threaded code is defined below giving an interactive shell to real time systems.

The current implementation does not define dictionary/header structures but leaves these as unspecified as the standards do. We still get an interactive system with a well defined threaded code structure. If more specific knowledge about a system is required, it would of course be possible to also specify the exact structure of headers and the dictionary layout and to define appropriate operations on these structures.

2 Threaded Code

This section describes the chosen threaded code structure of colon definitions. We will look at simple words with invocation of primitives, at string and number literals, and at control structures.

Let's assume the definition:

```
~: sq ( x -- x ) dup * ~;
```

then the threaded code for `sq` looks as shown in figure 1 on the preceding page. For each word, that is referenced by `sq` a corresponding execution token is stored. A thread-colon definition ends with the execution token of `~exit` compiled by `~;` (*thread-semicolon*). `sq` invokes only primitives, but the threaded code structure would be identical if thread-colon words were invoked: They also have execution tokens and these would be stored in the body of the newly defined word.

An interpreter pointer `IP` references the current point of execution. The threaded code interpreter will modify `IP` while executing the code.

2.1 Number literals

When there is a number literal in the source code, it is later processed by means of `~lit`:

```
~: test ( -- u )  3 4 + ~;
```

as can be seen in figure 2 on the top of this page.

2.2 Printing string literals

Printing string literals (used by `~."`) is handled by `(~."`, see figure 3 below.

```
~: test3 ( -- )   ~." it works" ~;
```

```
  body
  of test3                                      aligned
|------------+---+-----+-----+-----+-----+-----+---+-------------|
| xt of (~." | 8 | 'i' | 't' | ' ' | ... | 's' |   | xt of ~exit |
|------------+---+-----+-----+-----+-----+-----+---+-------------|
```

Figure 3: Threaded code for string literals `~: test3 ( -- ) ~." it works" ~;`

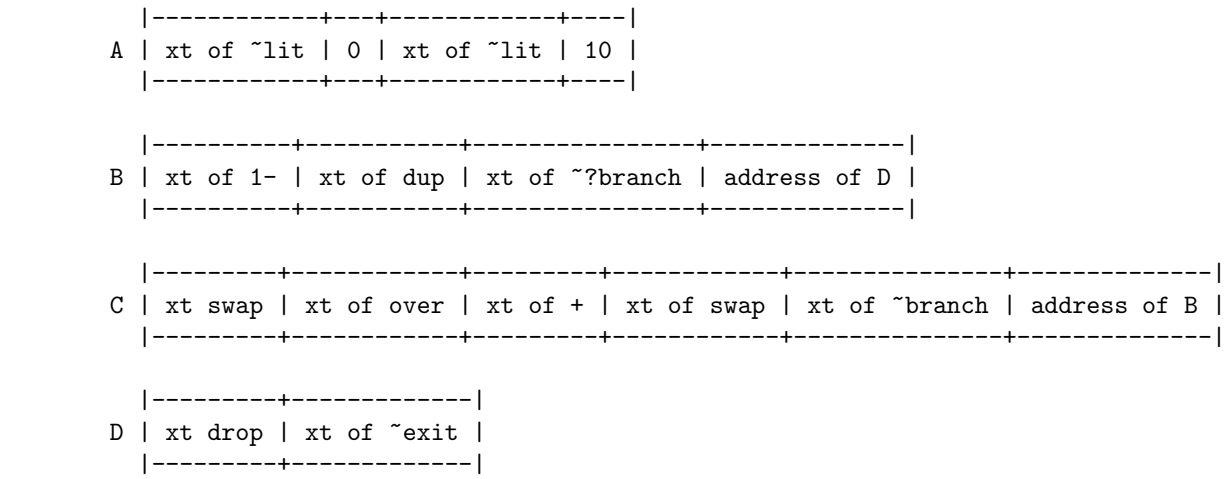

Figure 4: Threaded code for a `BEGIN WHILE REPEAT` loop

A counted string with string length and the string characters is placed inline in the code . In order to place the following token on an aligned address the bytes after the inline string are padded.

Our threaded code is (arbitrarily) restricted to just printing strings. A more general approach could easily define threaded string literal words corresponding to `S"`.

2.3 Control structures

Control structures compile unconditional and conditional absolute branches:

For illustration let's define the word `looptest` with a `BEGIN WHILE REPEAT` loop:

```
: looptest ( -- )
   0 10
   ~BEGIN
     1-  dup
   ~WHILE
     swap over + swap
   ~REPEAT drop  ~;
```

Figure 4 shows the corresponding threaded code. There are 4 basic blocks labeled A through D with appropriate branches at the end of basic block B (conditional) and C (unconditional). The threaded code primitives `~?branch` and `~branch` modify the interpreter pointer `IP` appropriately.

2.4 Limitations

The current threaded code defines just as much structure so that a simple interactive Forth outer interpreter can be defined on top.

It does not define a code for a complete standard system. Specifically the current threaded code does not contain

- (user) variables or constants. However because definitions of the underlying system become primitives of threaded code, it inherits variables and constants.
- `DO LOOP`s. Adding this would be similar to the branching words already available.
- Neither defining words nor `DOES>` as they are not required to program an interactive outer interpreter.
- a primitive for pushing address and length of string literals on the stack. It could easily be defined similar to the inline string printing word.

3 Implementation

This section explains how standard Forth definitions will allow to interpret and construct the threaded code structure explained above. As a general naming convention, names that start

```
\ Perform a single interpretation step
: step ( i*x -- j*x )
   IP @ dup cell+ IP !
   @ catch
   ?dup IF cr ." Error " . reset THEN ;

\ Loop steps
: run ( i*x -- j*x )
   BEGIN IP @ WHILE  step  REPEAT ;
```

Figure 5: The threaded code inner interpreter

with the tilde-character ~ denote threaded code words. They often have corresponding words in the underlying system with similar functionality.

The state of the threaded code interpreter has the following components:

- the already mentioned Interpreter Pointer `IP`:

  ```
  Variable IP  0 IP !
  ```

- a return stack `~RP` with its corresponding return stack pointer `RP`.

  ```
  Create ~R0  20 cells allot
  Variable RP  ~R0 RP !
  ```

  Having a return stack of our own, allows us to explicitly define the return stack behavior when nesting. Using that knowledge return stack tricks can work (we make no use of them here, though).

- a data stack shared with the underlying system, and

- memory (code and data) also shared with the underlying system.

Also headers, wordlists and the dictionary structure is shared with the underlying system.

3.1 Inner Interpreter

We now define the threaded code inner interpreter. It is defined in Figure 5. It works very similar to the `NEXT` code in classical Forth implementations:

`step` first gets the address of the next execution token and increments `IP`. It then fetches the execution token and executes it, which modifies the interpreter state as desired. In case of an error the word `reset` is invoked, which re-initializes the interpreter.

In traditional Forth implementations every invoked word ends in a jump to the `NEXT` code (or inlines it, if short enough) which results in continuous threaded code interpretation. In our case, as `step` invokes words via `catch`, they return to `step` and no continuous interpretation takes place. This has the benefit, that we can stepwise interpret threaded code (thus the name `step`) and gain control after each step. Continuous interpretation is handled by `run` in simple cases which calls `step` in a loop. Setting `IP` to `0` in one of the invoked words would stop `run`. Note, that `IP` had been initialized to `0` so that a `~exit` from the top level word will set `IP` to `0` as well and thus also stop threaded code interpretation. A real time environment would not call `run` but would do single interpretation `step`s when appropriate.

3.2 Return stack operations

The return stack we defined above grows towards increasing addresses. It should operate

```
: ~>r ( x -- )
  \ push a cell to the return stack
  RP @ !   1 cells RP +! ;

: ~r> ( -- x )
  \ pop a cell from the return stack
  -1 cells RP +!   RP @ @ ;
```

Figure 6: Return stack operations

using post increment and pre decrement operations. `RP` is supposed to point to the next available cell. Figure 6 shows the appropriate definitions for `~>r` and `~r>`.

3.3 Inline number literals

Inline number literals are prefixed with the `~lit` instruction as shown in figure 7. Its definition is

```
: ~lit ( -- n )
  \ extract inline number literal
  IP @ @  1 cells IP +! ;
```

Before execution, the interpreter pointer points to the code cell that contains `val` (solid line). Execution extracts the value *val* and puts it on the stack. After execution the interpreter pointer points past the literal (dotted line). This threaded code structure is generated by the threaded code outer interpeter, that we will define in section 3.6.

3.4 Printing inline string literals

Inline string literals are handled similar to number literals by `(~."` as shown in figure 8 on the next page. Note, that we are only interested in printing inline string literals here:

```
: (~." ( -- )
   \ extract inline string
   \ literal and print it
   IP @  count  2dup + aligned IP !
   type ;
```

Moving the interpreter pointer past the inline string requires alignement as specified for our threaded code structure.

The threaded code of figure 8 is generated by the compiling word `~."` that is defined like this:

```
: ~." ( <ccc>" -- )
  \ Compile inline string to be
  \ printed later when executed.
  \ Like ." but for threaded code
  ['] (~." ,
  [char] " word count
  here over 1+ chars allot place align
; immediate
```

It first compiles `(~."`, then the counted string and also takes care of the required alignment.

3.5 Control structures

Up to now threaded code execution is sequential. `step` moves the interpreter pointer suc-

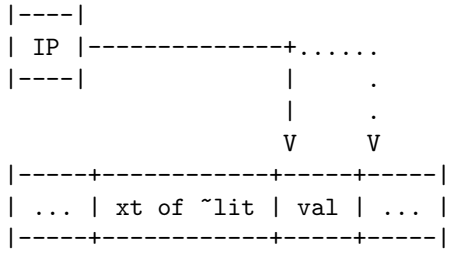

Figure 7: The execution of `~lit`

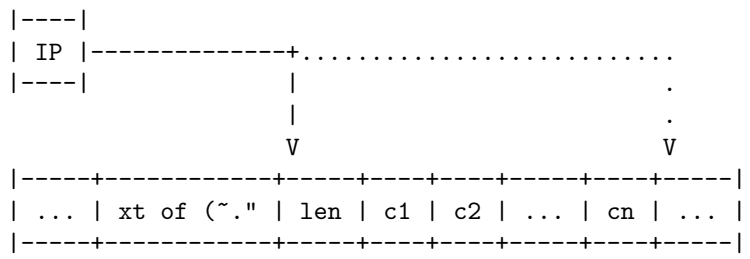

Figure 8: The execution of `(~."`

cessively over threaded code cell by cell. No branches take place.

Defining branches and control structures is simple. As in traditional Forth implementations we define unconditional (`~branch`) and conditional (`~?branch`) branches. For the sake of simplicity they branch to absolute addresses. Other branching regimes or additional branching instructions such as `DO-LOOP` primitives would easy to add.

```
: ~branch ( -- )
   \ absolute unconditional jump
   IP @ @ IP ! ;

: ~?branch ( f -- )
   \ absolute conditional jump
   IF  1 cells IP +!
   ELSE ~branch THEN ;
```

The interpreter pointer is adjusted appropriately. After execution it points to the branch target or (in case of a non taken conditional branch) to the instruction following `~?branch` (skipping the branch address).

In order to compile branches in a structured way we define the zoo of Forth control structure as depicted in figure 9.

These correspond to the standard control structures but compile threaded code branches with embedded absolute threaded code addresses.

3.6 Compiler *to* threaded code

In section 3.3 we saw the handling of inline number literals. It is still open, how the appropriate threaded code structure (figure 7 on the previous page) is generated. As in a traditional implementation this is done by the Forth text interpreter.

In figure 10 on the following page we define a variant of the classical outer compiler that looks for words in the dictionary, executes them when they are immediate or compiles them when not.

If a word is not found in the dictionary, the compiler tries to see if it is a number and then compiles it as number literal if so, or else raises an error. Note, that this compiler does not handle double numbers or base prefixes.

```
: ~IF ( -- x )  ['] ~?branch , here 0 , ; immediate
: ~AHEAD ( -- x )  ['] ~branch here 0 , ; immediate
: ~ELSE ( x -- x' )  ['] ~branch , here 0 , swap here swap ! ; immediate
: ~THEN ( x -- )  here swap ! ; immediate

: ~BEGIN ( -- x ) here ; immediate
: ~WHILE ( x1 -- x2 x1 )  ['] ~?branch ,  here 0 , swap ; immediate
: ~AGAIN ( x -- )  ['] ~branch , , ; immediate
: ~UNTIL ( x -- )  ['] ~?branch , , ; immediate
: ~REPEAT ( x2 x1 -- )  postpone ~AGAIN  postpone ~THEN ; immediate
```

Figure 9: Threaded code control structures

```
variable ~state   ~state off  \ threaded code outer interpreter state

-13 Constant #notfound

: ~] ( -- )  ~state on
   BEGIN ( )
      BEGIN ( )
        bl word dup c@   \ scan next token
      WHILE ( c-addr )   \ another token found
        find  ?dup       \ look up in dictionary
        IF   -1 = IF ,  ELSE execute  THEN  ~state @ 0= IF EXIT THEN  \ found
        ELSE   0 0 rot count
          over c@ [CHAR] - = dup >r IF  1- swap char+ swap THEN \ word not found
          >number IF  #notfound throw THEN
          drop drop r> IF negate THEN
          ['] ~lit , ,  \ compile threaded code literal
        THEN
      REPEAT ( c-addr )  \ no more tokens in input stream
      DROP
      SOURCE-ID 0= IF CR ." ] " THEN
      REFILL 0=    \ read more from input stream
   UNTIL ;  \ input stream exhausted

: ~[ ( -- )  ~state off ; immediate   \ stop threaded code compiler

: ~: ( <name> -- )
   \ push IP to return stack and set IP to start of threaded code.
   Create ~]  Does> IP @ ~>r  IP ! ;

: ~EXIT ( -- )
   \ Pop IP from return stack
   ~r> IP ! ;

: ~; ( -- )
   \ Compile end of definition and leave threaded code outer compiler
   ['] ~EXIT ,   ~state off   ; immediate
```

Figure 10: Compiler to threaded code

Threaded code words are defined with

```
~: name ....   ~;
```

`~:` invokes `~]` that compiles the following source code until `~state` becomes `false` (by executing `~;` or `~]`) or the input stream is exhausted. Inside the definition we have to use corresponding threaded code words (e. g. control structurres) to compile the right code.

We can then interactively execute the threaded code definition with

```
name run
```

The (normal) Forth word `name` sets the interpreter pointer to the beginning of its threaded code. `run` then interprets this code. If the interpreter executes `name`'s `~exit` `IP` will become zero (being initialized to zero and pushed on the return stack) and the `run` loop terminates. We return to the underlying system.

As dictionary structure and headers are shared with the underlying system, the headers for threaded code are just defined in the base Forth system.

Note also that execution of a threaded code word is split into two parts. On execution of the word in the underlying system interpretation

```
~: ~interpret ( -- )
    ~BEGIN ( )
      bl word dup c@    \ scan next token
    ~WHILE ( c-addr )   \ another token found
      find              \ lookup in dictionary
      dup  1 = ~IF drop execute  ~ELSE   \ immediate
      dup -1 = ~IF drop state @ ~IF  compile,  ~ELSE  execute ~THEN ~ELSE
      \ word not found, number?
      drop   0 0 rot count   over c@ 45 = dup ~>r ~IF  1- swap char+ swap ~THEN
           >number ~IF  #notfound throw ~THEN drop drop    \ maybe number
           ~r> ~IF  negate ~THEN
           state @ ~IF  postpone LITERAL  ~THEN  \ compile literal
      ~THEN ~THEN
    ~REPEAT ( c-addr )
   drop ~;

   ~: ~quit ( -- ) clear-stack   ~R0 RP ! ~state off  interpret-mode
     ~BEGIN cr state @ ~IF ~." ] " ~THEN  ~query ~interpret ~." ~ok"  ~AGAIN ~;
```

Figure 11: Interpreter in threaded code

does *not* start immediately but only the interpreter pointer is adjusted appropriately (saving its old content to the return stack). By this we can explicitly control execution by `step` and `run`.

3.7 Interpreter *in* threaded code

Ultimately we want to have a Forth outer interpreter that is defined *in* threaded code so that we can have an interactive shell which can be executed via `step` and `run`. Up to now we just have a compiler *to* threaded code, but this is defined in the underlying Forth system and we cannot control its execution.

So — here we go. We define a Forth outer interpreter in threaded code. Figure 11 shows a FIG forth style interpreter loop that combines interpretation and compilation state. It compiles to code of the base system (using `compile,` and `Literal`). So — it is similar in function to the base system outer interpreter but is itself compiled to threaded code using the threaded code compiler `~[` defined above. And so, we can control its execution via `step`.

`~quit` is the corresponding `quit` loop that successively expects user input and interprets it. (`~query` is an appropriate `query` implementation in threaded code, not shown).

4 Conclusion

In this document we defined a threaded code structure for Forth colon definitions and an inner interpreter, `step`, in high level Forth. Stepwise execution of threaded code can be controlled by periodicaly invoking `step`.

We constructed a compiler to generate this threaded code and also an interactive outer interpreter *in* threaded code. As `step` can control the execution of this outer interpreter, its execution time can be distributed according to the requirements of a real time system, such as the synchronous Forth framework in [1].

References

[1] *A synchronous FORTH framework for hard real-time control*, U. Hoffmann and A. Read, euroForth 2016

[2] *Threaded Interpretive Languages: Their Design and Implementation*, R. G. Loeliger, McGraw Hill, 1981