noForth website

noForth

How noForth is made

noForth is a standalone forth for MSP430

(Albert Nijhof, september 2014, november 2017)

How noForth is made

1. What is needed?

- The **target code** (written in noForth) that describes the whole noForth
- The **meta code** (written in forth) that will include also the MSP430 meta assembler (cross assembler)
- A standard 32bit forth on the host computer

2. What to do?

- Include the meta code on the host computer (for example in Win32forth).
- Then include the target code.

  You will be asked to choose your processor+board. Then the meta compiler converts the target code into a **binary image** of noForth. That image is saved as an **intel-hex** file.
- The intel-hex file can be sent to the chip with a **programmer**.

3. The target code

The target code defines noForth "in terms of itself".
At first sight this seems nonsense but we use a **metacompiler** (an auxiliary noForth in the host forth) that is able to convert the target code into a noForth image for the MSP430.

```
\ Examples of target code

code DUP       tos sp -) mov   NEXT end-code
code DROP      sp )+ tos mov   NEXT end-code
code !         sp )+ tos ) mov
               sp )+ tos mov   NEXT end-code

20 CONSTANT BL
: SPACE  ( -- )     bl emit ;
: SPACES ( n -- )   false ?do space loop ;
: TYPE   ( a n -- ) false ?do count emit loop drop ;

VARIABLE BASE
VARIABLE STATE
: DECIMAL   0A base ! ;
: [      ( -- )   false state ! ; IMMEDIATE
: ]      ( -- )   true state ! ;

: CHAR   ( <name> -- ch )  bl word count 0= ?abort c@ ;
: [CHAR] ( <name> -- ) char postpone literal ; IMMEDIATE
: MS     ( n -- )      0 ?do ms) 0 ?do loop loop ;
: REPEAT        postpone again postpone then ; IMMEDIATE
```

4. Image building

The binary image of noForth is being built while the metacompiler is interpreting the target code. The metacompiler is able to look up words in the growing image. Of course the noForth words in the image can not be executed on the host computer.

First, let's take the **blue words** in the example above. These words (and numbers) must be compiled into noForth colon definitions. A word can be compiled only when:

(a) it already exists in the image,
(b) it is not immediate and
(c) STATE is true.

The **red words** are executed by the metacompiler: immediate words within colon definitions and words not in colon definities.

And the **green words**? - Green text is handled by the preceding red word, not by the metacompiler itself.

5. x-words

Probably all red words already exist in the host forth but most of them are useless for the metacompiler.
Example: host forth headers differ from noForth headers, so the red ":" has to do other things than the original ":" in the host forth.
Therefore we have to redefine the red words in such a way that they will behave like noForth words. In fact we make an (incomplete) noForth simulator on the host computer. That's not really hard, but a very confusing problem arises: conflicting names. Our solution is very simple and brute: we put an "x" before the names of redefined red words. So we get:

```
x;  x:  xCODE  xLOOP  xIF  xVARIABLE  xIMMEDIATE  etc.
```

When all the necessary x-words are defined we put them, without the "x", in a vocabulary META.

```
VOCABULARY META
ONLY FORTH ALSO META DEFINITIONS FORTH
: :     x: ;
: ;     x; ;
: CODE  xCODE ;
: LOOP  xLOOP ;
: IF    xIF ;           etc.
```

The red words end up in the META vocabulary with their normal names while META is not in the search-order. This approach seems a bit clumsy and not very elegant, but it is effective and above all: the code remains clear and easy to read.

Why not define the red words right away into META? - Because red words often contain other red words and that would require a lot of vocabulary juggling.
Keep it simple is the motto.

6. Our own meta-interpreter

Interpreting the target code (the red and blue words) becomes simple if **we don't use the host forth interpreter, but write our own interpreter instead**.

```
\ The  metacompiler
: WINTERPRET    ( bl-word -- )  \ Interpret one word
   xSTATE @
   IF search-word-in-image      \ Using xFIND
      found?
      IF not-immediate?
         IF xCOMPILE, EXIT
         THEN
      THEN
   THEN
   search-word-in-meta          \ Using SEARCH-WORDLIST
   found?
   IF EXECUTE EXIT
   THEN
   is-it-a-number?
   IF xSTATE @ IF xLITERAL THEN EXIT
   THEN
   Error ;

: METACOMPILING   BEGIN BL WORD WINTERPRET AGAIN ;
```

This metacompiler as such is straightforward and the task to build a noForth from the target code is now divided into a lot of relatively small problems: (re)defining all red words that appear in the target code.

The target code starts with the word `:::NOFORTH:::` that activates the metacompiler. The word `;;;NOFORTH;;;` at the end of the target code stops the metacompiler.

7. Natural order of definitions

The definitions in the target code appear in the natural forth order, a blue word must already exist in the image before it can be compiled in a colon definition. We made this choice because consequently no registration of addresses that must be filled afterward is needed. This can be problematic for defining words, but we solve that by giving DOES> parts a name.

8. Doers

See also "The kangaroo method" in chapter 13.

In the target code the named DOES> part, **the doer**, is defined apart from and long before the CREATE action of the defining word.

`:DOER ccc` defines a doer in high level forth, replaces `DOES>`
`CODEDOER ccc` defines a doer in assembler, replaces `;CODE`

```
:DOER DOCON   @ ;
CODEDOER DOCOL   ip push   w ip mov   NEXT end-code

20 CONSTANT BL
: SPACE   BL EMIT ;

\ And later on in the target code
: CONSTANT   CREATE DOCON , ;
: :          CREATE DOCOL ... ;
```

As soon as `DOCOL` and `DOCON` do exist in the image, the metacompiler is able to build colon definitions and constants in the image.

A doer is a data word. The doer body contains the DOES> routine (the "data"). A doer (when executed) puts its body address in the CFA of the newest word. This method makes it possible to metacompile noForth with only one or two forward references, not regarding the jump forward within colon definitions (`IF WHILE ELSE`).

A side effect is that a noForth decompiler easily detects word types.

9. The metacompiler is completed before metacompiling starts

Nothing is added to the metacompiler during the metacompiling process, the knowledge in the red meta words and the possibility to look up things in the image, must be enough.
There is no confusing intermingling of host compiling and target compiling.
It means too that we can put the noForth image in the dictionary space of the host forth (The host C, HERE and ALLOT can be used for image building).

10. Late binding

At the moment that `xCONSTANT` is defined as a red word in the meta code the address of `DOCON` is not known. We put "DOCON" as a string in the `xCONSTANT` definition, not its address.

```
: xCONSTANT   xCREATE DOCON x, ;
```

becomes something like:

```
: xCONSTANT   xCREATE xDOER" DOCON" x, ;
```

Every time the red `CONSTANT` is executed `xDOER" DOCON"` must look up `DOCON` in the image. Thus it happens in all red defining words and also in the compiling red words as

```
LITERAL  ."  S"  DO  ?DO  LOOP  +LOOP  POSTPONE  ?ABORT
```

At compile time the words they compile must be searched in the image.

11. Two reusable labels

`AMTSTERDAM` and `ROTTERDAM` are two labels. They can be used again and again. With them we can jump to code fragments that we want to reuse.
`LABEL-AMSTERDAM` puts the address where we are into the label, `AMSTERDAM` puts that address on the stack. The same with `ROTTERDAM`.

```
code TRUE  tos sp -) mov
           LABEL-AMSTERDAM   #-1 tos mov   NEXT end-code
code FALSE tos sp -) mov
           LABEL-ROTTERDAM   #0 tos mov    NEXT end-code
code =     sp )+ tos cmp     =? ROTTERDAM label-until,
           AMSTERDAM jmp end-code
code MIN   sp )+ w mov   tos w cmp
           LABEL-AMSTERDAM   >? if,   w tos mov   then,
           NEXT end-code
code MAX   sp )+ w mov   w tos cmp
           AMSTERDAM jmp   end-code
```

12. A closer look at "Image building"

The metacompiler is able to show how the image grows. The word `TRACE` activates and `NOTRACE` deactivates this function. Put these words around a few definitions in the target code and you can study in detail how the image is being built. This was our debugger.
Use the space bar for wait/continue.
Three xamples:
a.

```
trace
forth: : .S ( -- )
    ?stack (.) space
    depth false
    ?do  depth i - 1- pick
        base @ 0A = if . else u. then
    loop ;
notrace
```

The output:

```
E55A forth:         ( )
E55A :              ( )
E55A <<<<<  .S  >>>>>
E55A           E49E , E55A 0208 !       ( )
E55C           81 c, 82 c, ".S" m, DOCOL    C176 ,       ( 44 )
E562    (              ( 44 )
E562    ?stack   D1F4 ,       ( 44 )
E564    (.)      D58A ,       ( 44 )
E566    space    CF8E ,       ( 44 )
E568    depth    D2C8 ,       ( 44 )
E56A    false    C7C2 ,       ( 44 )
E56C    ?do      ?DO(     C2C4 , 0000 ,       ( 44 E56E 33 )
E570    depth    D2C8 ,       ( 44 E56E 33 )
E572    i        C326 ,       ( 44 E56E 33 )
E574    -        C9E0 ,       ( 44 E56E 33 )
E576    1-       C9A4 ,       ( 44 E56E 33 )
E578    pick     C738 ,       ( 44 E56E 33 )
E57A    base     C438 ,       ( 44 E56E 33 )
E57C    @        C578 ,       ( 44 E56E 33 )
E57E    0A       0015 ,       ( 44 E56E 33 )
E580    =        C7D4 ,       ( 44 E56E 33 )
E582    if       8FFF ,       ( 44 E56E 33 E582 11 )
E584    .        D104 ,       ( 44 E56E 33 E582 11 )
E586    else     7FFF , 8805 E582 !       ( 44 E56E 33 E586 11 )
E588    u.       D0F6 ,       ( 44 E56E 33 E586 11 )
E58A    then     7803 E586 !       ( 44 E56E 33 )
E58A    loop     LOOP)    C300 , E58C E56E !       ( 44 )
E58C    ;        EXIT     C102 ,       ( )
E58E notrace
```

b.

```
trace
forth: 20 constant BL
notrace
```

The output:

```
C342 forth:         ( )
C342 20             ( 20 )
C342 constant
      ( 20 )       ( 20 )
C342 <<<<<  BL  >>>>>
C342           C320 , C342 0200 !       ( 20 )
C344           81 c, 82 c, "BL" m,  DOCON    C19E , 0020 ,       ( )
C34C notrace
```

c.

```
trace
forth: code EXIT
LABEL-AMSTERDAM   rp )+ ip mov   NEXT end-code
extra: code ?EXIT   ( flag -- )
    #0 tos cmp   sp )+ tos mov   =? AMSTERDAM label-until,
    NEXT end-code
notrace
```

The output:

```
C0FA forth:         ( )
C0FA code           ( )
C0FA <<<<<  EXIT  >>>>>
C0FA           0000 , C0FA 0202 !       ( )
C0FC           81 c, 84 c, "EXIT" m,  C104 ,       ( 55 )
C104 LABEL-AMSTERDAM       ( 55 )
C104 rp             ( 55 1 )
C104 )+             ( 55 1 -3 )
C104 ip             ( 55 1 -3 5 )
C104 mov      4135 ,       ( 55 )
C106 NEXT     4F00 ,       ( 55 )
C108 end-code       ( )
C108 extra:         ( )
C108 code           ( )
C108 <<<<<  ?EXIT  >>>>>
C108           C0E6 , C108 0204 !       ( )
C10A           83 c, 85 c, "?EXIT" m,  FF c, C114 ,       ( 55 )
C114 (              ( 55 )
C114 #0             ( 55 3 )
C114 tos            ( 55 3 7 )
C114 cmp      9307 ,       ( 55 )
C116 sp             ( 55 4 )
C116 )+             ( 55 4 -3 )
C116 tos            ( 55 4 -3 7 )
C116 mov      4437 ,       ( 55 )
C118 =?             ( 55 2000 )
C118 AMSTERDAM       ( 55 2000 C104 )
C118 label-until, 23F5 ,       ( 55 )
C11A NEXT     4F00 ,       ( 55 )
C11C end-code       ( )
C11C notrace
```

13. The kangaroo method

In the newer noForth versions (november 2017) we use the "kangaroo method" for defining words and compiler words.

a. Defining words

DOES> parts remain unnamed. We put them in the body of the defining word. Examples for ":" and CONSTANT:

```
CODE :
ip push   w ip mov   NEXT end-code          \ This is docol

code CONSTANT
tos sp -) mov   w ) tos mov   NEXT end-code \ This is docon
```

This code can be compiled in an early stage of the metacompilation process. Now the metacompiler is able to build colon definitions and constants in the image because the addresses of DOCOL and DOCON are known. These ":" and CONSTANT will never be executed by the metacompiler, they are used by the red metacompiler words ":" and CONSTANT in order to obtain the addresses of DOCOL and DOCON.

```
20 CONSTANT BL
: SPACE   BL EMIT ;
```

Later on in the target we overwrite the code fields of ":" and CONSTANT with the addresses of their CREATE actions as a DOES> routine (noForth is indirect threaded).

```
TELL : TO-DO: header ( docol ) , ] ... ;
TELL constant TO-DO: header ( docon ) , ( value ) , reveal ;
```

When ":" is executed in the living noForth this TO-DO: code will be executed, with the DOCOL address on the stack! Similarly for CONSTANT.
Now we need no headers for doers and we have no forward references. A side effect is still that a noForth decompiler easily detects word types.

Decompiler output:

```
see bl
C32C z   C17A --- BL ( CONSTANT )
C32E       20

' constant @ msee
DFB8     12B0
DFBA N   C14E ( DOES> )
DFBC :   DB3A HEADER
DFBE     CDC6 ,
DFC0 8   D738 REVEAL
DFC2     CDC6 ,
DFC4 B   C142 ( ; )

' constant >body mdas
C17A: $    8324   #2    sp SUB
C17C:  G   4784   tos    0 sp x) MOV
C17E:         0
C180: 'F   4627   w )    tos MOV
C182: 6E   4536   ip )+    w MOV
C184: 0F   4630   w )+    pc MOV --->>
```

In the decompiler `( ccc )` can be read as: this is the body address of 'ccc'.

b. Compiler words

To save space we use the kangaroo method also for compiler words. Example:

```
code IF  chere cell+ ,  \ CFA for 0BRANCH
...0branch-code... next end-code immediate
```

The body of IF now contains the 0BRANCH-code as a noname-primitive. Later on the codefield of IF is overwritten with the address of the compile action (as a DOES> routine) for IF:

```
TELL if TO-DO: compile, ... ;
```

Remember that this IF (when executed in noForth) finds the token of 0BRANCH on the stack.

Similarly: put the (DO)-code in the body of DO and

```
TELL do TO-DO: compile, ... ;
```

etc.

This means that also auxiliary words for compiler words need no headers. The decompiler nicely shows those auxiliary words as the defining word or compiler word in question within 'airy' parenthesis. Example of decompiler output:

```
see m,
CDEE f   C166 --- M, ( : )
CDF0     CB8E BOUNDS
CDF2     C2AC ( ?DO )
CDF4     CDFE
CDF6     C30C I
CDF8 V   C556 C@
CDFA     CDDA C,
CDFC     C2E6 ( LOOP )
CDFE B   C142 ( ; )
```

Remember that in the decompiler `( ccc )` can be read as: this is the body address of 'ccc'.

*