```
xVersion" 2011.11.5.1"

module forthInterpreter

  #document forthInterpreter
    //
    // ////////////////////////////////////////////////////////////////////////
    //
    // This is an internal forth engine, with some assembly level magic words
    // can be added to it from the cross compiler, but also when finished it
    // will be possible too feed in forth source code that will then be
    // compiled.
    //
    // Using direct threading according to the jump model from:
    //
    //   http://www.figuk.plus.com/build/heart.htm

    // also see:
    //
    //   http://www.bradrodriguez.com/papers/moving1.htm
    //
    // The computed jumps are being performed on registers that are then
    // pushed onto the processor hardware stack, this when followed by a return
    // instruction effectively performs an indirect jump. (GOTO/BRA)
    //
    // Where DATA was used in the first URL I'll use PFA. doVar will differ in
    // that it returns the RAM address that will need be compiled into PFA  by
    // Variable (as RAM is in another program space than FLASH) ... a Harvard
    // architectural quirk ... anyway, a variable is a constant that is used
    // as a pointer into RAM. The @ and ! operator will work on RAM and not on
    // FLASH. Latr on , and F, and such and FALLOT will work in code (FLASH)
    // space. VALLOT works in variable (RAM) space.
    //
    // This is a 32 bit forth using a cell size of four bytes although in FLASH
    // only 24 bits are valid, the upper 8 ones are zero by definition. And
    // for RAM pointers 17 bits are in use, so this should be able to execute
    // from any FLASH location and be able to use any RAM location. The return
    // and data stack pointers are only 16 bits wide, meaning return and data
    // stack will have to reside in the lower 64 k of RAM.
    //
    // Primaries are just machine code that end in "goto <*" t.next">".
    // Secondaries are pointer lists that start like a primary with code
    // handler bra <*" t.doCol">, variables are pointers to RAM that get
    // compiled into flash as a constant and constants are lteral values
    // compiled into FLASH, the value being pushed by bra <*" t.doCon">.
    //
    // Register allocations used :
    //
    //   TOS    w0, w1                          // Top Of Stack Cache Register
    //   IP     TBLPAG, w8                      // Instruction Pointer, walks over code pointer lists.
    //   W      w6, w7                          // W Register, used to compute PFA at run time
    //   PSP    w9                              // Parameter Stack Pointer - maps onto the compiled forth
    //   RSP    w10                             // Return    Stack Ponter  - maps onto the compiled forth
    //   UP     in memory                       // As a variable
    //   HERE   in memory                       // As a variable
    //
    //   TOS == Top  Of data/parameter Stack
    //   NOS == Next On data/parameter Stack , the 2nd stack element.
    //
    // Header layout used for compiled forth words
    //
    //  LFA : + 0 :: Link Field Address
    //
    //        points to the previous definition the last definition is pointed
    //        to through the variable LAST.
    //
    //  FFA : + 4 :: Flag Field Address
    //
    //        byte 3 : always zero
    //        byte 2 : always FF
    //        byte 1 : always FF
    //        byte 0 : bit 0 : immediate
    //                 bit 1 : hidden
    //
    //  NFA : + 8 :: Name Field Address
    //
    //        a variable lenght field, the low byte of the first pword
    //        holds the length, the remaining part holds the full name.
    //        When this does not completely fill out up to the next even
    //        flash address some alignment bytes are added.
    //
    //  CFA : + 8 + Length( Name) 4 bytes aligned :: Code Field Address
    //
    //        This field always holds code, which can be the start of
    //        a code block for a primary word, or a jump to a handler
    //        routine for secondary words. Primaries will end with
    //        a goto Next jump usually.
    //
    //  PFA : + 12 + Length( Name) 4 bytes aligned :: Parameter Field Address
    //
    //        For secondary words only, contains the data used by the handler
    //        compiled into the CFA. I.e. a list of pointers for a colon word
    //        (doCol handler), a value for a constant word (doCon) a pointer
    //        to RAM for a variable word (doVar) etc.
    //
    //
    //  Some code snippets do not have a header, the system will 'magically'
    //  know where they are, and they will be compiled by higher level words
    //  when they are needed. The word constant will compile doCon, Variable
```

```
    //  will compile doVar and : (colon) will compile doCol etc. etc.
    //
    //  The current headerless words are :
    //
    //    doCol (or ENTER) - pointer list interpreter
    //    doCon            - constant interpreter
    //    doVar            - variable interpreter
    //    doDefer          - defer interpreter
    //    doUser           - user variable interpreter
    //    doDoes>          - does> interpreter
    //
    // ////////////////////////////////////////////////////////////////////
    //
  #endDoc

private

  {
    //  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
    // Some meta magic
    //  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .


    2 constant THalfCell                            #// A Target HALFCELL is two  bytes (register size)
    4 constant TCell                                #// A Target CELL     is four bytes

    : TCells ( d -- d )

      TCell *
    ;

    : THalfCells ( d -- d )

      THalfCell *
    ;

    //  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

    variable DefinitionName                         #// Local  name for current definition
    variable TargetName                             #// Target name for current definition
    variable FlagsField                             #// Flags to be applied to current definition
    variable StackImage                             #// Target stack image for current definition
    variable LastLink         " 0" LastLink !       #// Last target NFA value
    variable TVoffset           0  TvOffset !       #// Current offset in target RAM "dictionary" space.


    //  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

    : NewTVoffset ( -- n )

      #// Make room for a new variable in target RAM and return it's offset

      TVoffset @
      dup TCell +
      TVoffset !
    ;

    //  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

    : TVCells ( -- n )

      // Amount of target data cells currently allocated

      TVOffset @
      TCell /
    ;


    //  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

    : HeaderName ( -- str )

      // Return the target header name from the definition name
      // The definition name is the target name prefixed with t.

      DefinitionName @ " .h" +
    ;

    //  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

    : registerHeader ( -- )

      // Parse some tokens from the input and remember those.

      bl token TargetName !                           // Get and store the target name
      bl Token FlagsField !                           // Get and store the flags
      bl token StackImage !                           // Get and store stack image
      " t." TargetName @ + DefinitionName !           // Make a local name by prefixing t.
    ;

    //  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

    : definition ( -- )

      // Start a target definition by buildin a header into a string variable
      // codeBuffer (as source code to be intepreted later on).

      registerHeader
```

```
  " " codeBuffer !
  "   code " HeaderName + "  " + StackImage @ +                                  cb+cr+
  "  "                                                                          cb+cr+
  "     .pword    " LastLink @ +                                               cb+cr+
  "     .pword    " FlagsField @ +                                             cb+cr+
  "     .pascii   <" TargetName @ strLen >$ + " >,\"" + TargetName @ + " \"" +  cb+cr+
  "     .palign   2"                                                           cb+cr+
  "  "                                                                         cb+cr+

  "   .ifdef    __C30ELF"                                                      cb+cr+
  "   .type     <!\" " DefinitionName @ + " \" mangle>" + " , @function" +     cb+cr+
  "   .endif"                                                                  cb+cr+
  "  "                                                                         cb+cr+

  " <!\" " DefinitionName @ + " \" mangle>:" +                                 cb+cr+

  " <*\" " HeaderName + " \">" + LastLink !
;

//  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

: endDefinition ( -- )

  // End a target definition and compile it by interpreting the contents
  // from the string variable codeBuffer

  "   endCode used"                                                          cb+cr+
  CodeBuffer @
  // dup message                              // debug code to dump code created into console
  evaluate
;

//  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

: scanForCode ( str -- )

  // Scan the input line by line until a line is found that contains
  // str, just add the lines to the codeBuffer for later interpretation.

  begin
    readLine                                // ( str line )
    over over                               // ( str line str line )
    strPos 0 =
  while
    cb+cr+
  repeat
  skipToEOL
  "  " cb+
  drop
  drop
;

//  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

: t.code ( -- )

  // Start a target code word (primary) to be ended with t.endCode

  definition
    " t.endCode" scanForCode
  endDefinition
;

//  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

: t.: ( -- )

  // Start a target colon word (secondary) to be ended with t.;

  definition
    "     bra       <*\" t.doCol\">                       ; // DOCOL :: Execute following pointer list" cb+cr+

    " t.;" scanForCode

    " "                                                                                                  cb+cr+
    "     .pword    <*\" t.semi\">                        ; // SEMI :: return to calling pointer list"  cb+cr+
  endDefinition
;

//  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

: t.constant ( value -- )

  // Define a target constant

  definition
    "     bra       <*\" t.doCon\">                       ; // DOCON :: push following pword" cb+cr+
    "     .pword    " swap >$ +                                                                  cb+cr+
  endDefinition
;

//  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

: t.variable ( -- )

  // Define a target variable

  NewTVoffset                               // Make room in target RAM, return offset for it
  definition
```

```
      "     bra       <*\" t.doVar\">                         ; // DOVAR :: push following pword" cb+cr+
      "     .pword    <@\" VBank\"> + " swap >$ +                                                 cb+cr+
    endDefinition
  ;

  //  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

  : t.defer ( -- )

    // Define a target deferred word (code pointer).

    NewTVoffset                                 // Make room in target RAM, return offset for it
    definition
      "     bra       <*\" t.doDefer\">                       ; // DODEFER :: execute following pword" cb+cr+
      "     .pword    <@\" VBank\"> + " swap >$ +                                                 cb+cr+
    endDefinition
  ;

  //  . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
}


// ////////////////////////////////////////////////////////////////////////

20 16 + THalfCells  constant  RBankSize             #// Register  Bank  Size - in bytes (keep a few spares to allow for PS
underflow)
   128 TCells       constant  PSSize                #// Parameter Stack Size - in bytes
   128 TCells       constant  RSSize                #// Return    Stack Size - in bytes
  4096 TCells       constant  VBankSize             #// Variable  Bank  Size - in bytes

       RBankSize    array     RBank                 #// Register bank storage area
       PSSize       array     PS                    #// Parameter Stack - TOS is at the lowest  address
       RSSize       array     RS                    #// Return    Stack - TOS is at the highest address
       VBankSize    array     VBank                 #// Variable bank storage area

                    2variable TVector               #// To set a target forth xt from native 4th - not implemented yet


// ////////////////////////////////////////////////////////////////////////

code forth.start ( -- )

  ; // Start the forth engine by jumping into COLD.
  ; //
  ; // Save registers

  breakpoint  forth.start
                                                    ; // forth.start
  mov       w0, <@" RBank">                         ; //  0 ::  0 : old w0
  mov       #<@" RBank"> + 2, w0
  mov       w1 , [w0++]                             ; //  2 ::  1
  mov       w2 , [w0++]                             ; //  4 ::  2
  mov       w3 , [w0++]                             ; //  6 ::  3
  mov       w4 , [w0++]                             ; //  8 ::  4
  mov       w5 , [w0++]                             ; // 10 ::  5
  mov       w6 , [w0++]                             ; // 12 ::  6
  mov       w7 , [w0++]                             ; // 14 ::  7
  mov       w8 , [w0++]                             ; // 16 ::  8
  mov       w9 , [w0++]                             ; // 18 ::  9
  mov       w10, [w0++]                             ; // 20 :: 10
  mov       w11, [w0++]                             ; // 22 :: 11
  mov       w12, [w0++]                             ; // 24 :: 12
  mov       w13, [w0++]                             ; // 26 :: 13
  mov       w14, [w0++]                             ; // 28 :: 14
  mov       w15, [w0++]                             ; // 30 :: 15
  mov       TBLPAG, w2
  mov       w2 , [w0++]                             ; // 32 :: 16
  mov       RCOUNT, w2
  mov       w2 , [w0++]                             ; // 34 :: 17
  mov       SR, w2
  mov       w2 , [w0]                               ; // 36 :: 18

  ; // Initialize forth interpreter

  mov       #<@" PS">, w9                           ; // Set PSP to lowest  address in PS

  mov       #<@" RS">, w3                           ; // Set RSP one beyond highest address in RS
  mov       #<@" RSSize">, w10
  add       w3, w10, w10

  ; // Jump into COLD, setting up initial W contents on the fly

  mov       #tblpage  ( <*" t.cold">), w2           ; // Set up t.cold as the entry point
  mov       w2, TBLPAG                              ; // Which needs be a forth secondary word
  mov       #tbloffset( <*" t.cold">), w8

  mov       w8    , w6                              ; // Set up initial W to point to t.cold
  mov       TBLPAG, w7

  push      w6                                      ; // JUMP IP (into t.cold)
  push      w7

  return
endcode used

#document forth.start
  // Start the forth engine by jumping into COLD.
  // COLD will run some code and when finished it
  // will return control to the caller of START
  // trhough the word BYE.
```

```
  #endDoc

  // ////////////////////////////////////////////////////////////////////////

  t.code bye 0xffff00 <( -- )>
                                                    ; // bye
<*" FIRST_FLASH">:

    ; // Restore registers

    mov       #<@" RBank"> + 36, w0
    mov       [w0--], w2                            ; // 36 :: 18
    mov       w2, SR
    mov       [w0--], w2                            ; // 34 :: 17
    mov       w2, RCOUNT
    mov       [w0--], w2                            ; // 32 :: 16
    mov       w2, TBLPAG
    mov       [w0--], w15                           ; // 20 :: 15
    mov       [w0--], w14                           ; // 28 :: 14
    mov       [w0--], w13                           ; // 26 :: 13
    mov       [w0--], w12                           ; // 24 :: 12
    mov       [w0--], w11                           ; // 22 :: 11
    mov       [w0--], w10                           ; // 20 :: 10
    mov       [w0--], w9                            ; // 18 ::  9
    mov       [w0--], w8                            ; // 16 ::  8
    mov       [w0--], w7                            ; // 14 ::  7
    mov       [w0--], w6                            ; // 12 ::  6
    mov       [w0--], w5                            ; // 10 ::  5
    mov       [w0--], w4                            ; //  8 ::  4
    mov       [w0--], w3                            ; //  6 ::  3
    mov       [w0--], w2                            ; //  4 ::  2
    mov       [w0--], w1                            ; //  2 ::  1
    mov       [w0]  , w0                            ; //  0 ::  0

    ; // Return to caller of forth.start

    breakpoint  t.bye

    return
  t.endCode

  #document t.bye.h
    // CODE BYE ( -- )
    //
    // Returns to the caller of START
  #endDoc


  // ////////////////////////////////////////////////////////////////////////////
  // ////////////////////////////////////////////////////////////////////////////
  //
  // Headerless words for inner interpreter

  code t.doCol ( -- ) // [ -- d ] (or ENTER)
                                                    ; // doCol
    mov       w8, [--w10]                           ; // IP PUSH RSP
    mov       TBLPAG, w2
    mov       w2, [--w10]

    inc2      w6, w8                                ; // PFA TO IP
    addc      w7, #0, w2
    mov       w2, TBLPAG

<*" t.next">:

    tblrdl.w  [w8  ], w6                            ; // [IP] TO W   CELL +TO IP
    tblrdh.w  [w8++], w7

    push      w6                                    ; // W JUMP
    push      w7
    return
  endCode used

  #document t.doCol
    // DOCOL :: A code fragment, not a forth primary
    //   IP PUSH RSP
    //   PFA TO IP
    //   NEXT
    //
    // NEXT :: A code fragment, not a forth primary
    //   [IP] TO W
    //   CELL +TO IP
    //   W JUMP
  #endDoc


  // ////////////////////////////////////////////////////////////////////////////

  code t.doCon ( -- d )
                                                    ; // doCon
    mov       w0, [w9++]                            ; // [PFA] PUSH PSP
    mov       w1, [w9++]

    inc2      w6, w6                                ; // PFA
    addc      w7, #0, w7
    mov       w7, TBLPAG

    tblrdl.w  [w6], w0                              ; // [PFA]
    tblrdh.w  [w6], w1
```

```
  goto      <*" t.next">                        ; // NEXT
endCode used

#document t.doCon
  // A code fragment, not a forth primary
  //   [PFA] PUSH PSP
  //   NEXT
#endDoc


// ////////////////////////////////////////////////////////////////////////

code t.doVar ( -- d )
                                                ; // doVar
  mov       w0, [w9++]                          ; // [PFA] PUSH PSP
  mov       w1, [w9++]

  inc2      w6, w6                              ; // PFA
  addc      w7, #0, w7
  mov       w7, TBLPAG

  tblrdl.w  [w6], w0                            ; // [PFA]
  tblrdh.w  [w6], w1

  goto      <*" t.next">                        ; // NEXT
endCode used

#document t.doVar
  // A code fragment, not a forth primary
  //   [PFA] PUSH PSP
  //   NEXT
#endDoc


// ////////////////////////////////////////////////////////////////////////

code t.doDefer ( -- )
                                                ; // doDefer
  mov       w0, [w9++]                          ; // [PFA] PUSH PSP
  mov       w1, [w9++]

  inc2      w6, w6                              ; // PFA
  addc      w7, #0, w7
  mov       w7, TBLPAG

  tblrdl.w  [w6], w0                            ; // [PFA]
  tblrdh.w  [w6], w1

  mov       [w0++], w2                          ; // @
  mov       [w0  ], w1
  mov       w2, w0

  goto      <*" t.execute">                     ; // EXECUTE
endCode used

#document t.doDefer
  // A code fragment, not a forth primary
  //   [PFA] PUSH PSP
  //   @
  //   EXECUTE
#endDoc


// ////////////////////////////////////////////////////////////////////////

code t.doUser ( -- d )
                                                ; // doUser
  breakpoint UNTESTED                           // !!! This code was not tested yet !!!

  mov       w0, [w9++]                          ; // [PFA] PUSH PSP
  mov       w1, [w9++]

  inc2      w6, w6                              ; // PFA
  addc      w7, #0, w7
  mov       w7, TBLPAG

  tblrdl.w  [w6], w0                            ; // [PFA]
  tblrdh.w  [w6], w1

  mov       <@" VBank"> + 4, w2                 ; // UP
  mov       <@" VBank"> + 6, w3

  add       w1, w3, w1                          ; // +
  add       w0, w2, w0
  addc      w1, #0, w1

  goto      <*" t.next">                        ; // NEXT
endCode used

#document t.doUser
  // A code fragment, not a forth primary
  //   [PFA] UP + PUSH PSP
  //   NEXT
#endDoc


// ////////////////////////////////////////////////////////////////////////

code t.doDoes> ( -- d )
```

```
                                                ; // doDoes>
  breakpoint UNTESTED                           // !!! This code was not tested yet !!!

  mov       w8, [--w10]                         ; // IP PUSH RSP
  mov       TBLPAG, w2
  mov       w2, [--w10]

  pop       TBLPAG                              ; // POP hardware call stack TO IP
  pop       w8

  mov       w0, [w9++]                          ; // [PFA] PUSH PSP
  mov       w1, [w9++]

  inc2      w6, w6                              ; // PFA
  addc      w7, #0, w7
  mov       w7, TBLPAG

  tblrdl.w  [w6], w0
  tblrdh.w  [w6], w1

  goto      <*" t.next">                        ; // NEXT
endCode used

#document t.doDoes>
  // A code fragment, not a forth primary
  //   IP PUSH RSP
  //   POP hardware call stack TO IP
  //   [PFA] PUSH PSP
  //   NEXT
#endDoc


// ////////////////////////////////////////////////////////////////////////////
// ////////////////////////////////////////////////////////////////////////////
//
// Inner interpreter words with headers

t.code execute 0xffff00 <( d -- )>
                                                ; // execute
  mov       w0, w6                              ; // POP PSP TO W
  mov       w1, w7
  mov       [--w9], w1
  mov       [--w9], w0

  push      w6                                  ; // W JUMP
  push      w7
  return
t.endCode

#document t.execute.h
  // CODE EXECUTE
  //   POP PSP TO W
  //   W JUMP
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.code semi 0xffff00 <( -- ) // [ d -- ] or EXIT>
                                                ; // semi
  mov       [w10++], w2                         ; // POP RSP TO IP
  mov       w2, TBLPAG
  mov       [w10++], w8

  goto      <*" t.next">                        ; // NEXT
t.endCode

#document t.semi.h
  // CODE SEMI
  //   POP RSP TO IP
  //   NEXT
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.code doLit 0xffff00 <( -- d )>
                                                ; // doLit
  mov       w0, [w9++]
  mov       w1, [w9++]
  tblrdl.w  [w8  ], w0
  tblrdh.w  [w8++], w1

  goto      <*" t.next">                        ; // NEXT
t.endCode

#document t.doLit.h
  // CODE DOLIT
  //   push inline value
  //   NEXT
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.code breakpoint 0xffff00 <( -- )>
                                                ; // breakpoint
  breakpoint  t.breakpoint

  goto      <*" t.next">                        ; // NEXT
```

```
  t.endCode

  #document t.breakpoint.h
    // CODE BREAKPOINT
    //   stop execution here in the debugger
    //   NEXT
  #endDoc


  // ////////////////////////////////////////////////////////////////////////

  t.code cfa2pfa 0xffff00 <( d -- d )>
                                                  ; // cfa2pfa
    add       w0, #2, w0
    addc      w1, #0, w1

    goto      <*" t.next">                        ; // NEXT
  t.endCode

  #document t.cfa2pfa.h
    // : cfa2pfa ( d -- d )
    //   calculate PFA from CFA
    // ;
  #endDoc


  // ////////////////////////////////////////////////////////////////////////

  t.code 0= 0xffff00 <( d -- df )>
                                                  ; // 0=
    ior       w0, w1, w0                          ; // Or low and high word
    sub       #1, w0                              ; // Subtract one, borrow set if 0= holds
    subb      w0, w0, w0                          ; // when borrow set $ffffffff, 0 otherwise
    mov       w0, w1

    goto      <*" t.next">                        ; // NEXT
  t.endCode

  #document t.0=.h
    // code 0= ( d -- df )
    //   return $ffffffff when d = 0, and 0 otherwise
  #endDoc


  // ////////////////////////////////////////////////////////////////////////

  t.code branch 0xffff00 <( -- )>
                                                  ; // branch
    tblrdl.w  [w8], w2                            ; // [IP] -> IP
    tblrdh.w  [w8], w3
    mov       w2, w8
    mov       w3, TBLPAG

    goto      <*" t.next">                        ; // NEXT
  t.endCode

  #document t.branch.h
    // code branch ( -- )
    //   branch to the following inline absolute address
  #endDoc


  // ////////////////////////////////////////////////////////////////////////

  t.code fbranch 0xffff00 <( df -- )>
                                                  ; // fbranch
    ior       w0, w1, w0                          ; // Or low and high word, branch if zero

    bra       z, 1f                               ; // B/ calculate new IP (branch)
    bra       2f                                  ; // B/ Skip over current IP

1:

    tblrdl.w  [w8], w2                            ; // [IP] -> IP // Branch
    tblrdh.w  [w8], w3
    mov       w2, w8
    mov       w3, TBLPAG
    bra       3f

2:

    mov       TBLPAG, w2                          ; // IP++       // No branch
    add       w8, #2, w8
    addc      w2, #0, w2
    mov       w2, TBLPAG

3:

    mov       [--w9], w1                          ; // Drop df
    mov       [--w9], w0

    goto      <*" t.next">                        ; // NEXT
  t.endCode

  #document t.fbranch.h
    // code fbranch
    //   branch to the following inline absolute address
    //   when TOS = 0, skip that address otherwise (and
    //   do not branch then).
  #endDoc
```

```
// ////////////////////////////////////////////////////////////////////////////

t.code drop 0xffff00 <( d -- )>
                                                ; // drop
  mov       [--w9], w1
  mov       [--w9], w0

  goto      <*" t.next">                        ; // NEXT
t.endCode

#document t.drop.h
  // CODE DROP
  //   drop TOS
  //   NEXT
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.code dup 0xffff00 <( d -- d d )>
                                                ; // dup
  mov       w0, [w9++]
  mov       w1, [w9++]

  goto      <*" t.next">                        ; // NEXT
t.endCode

#document t.dup.h
  // CODE DUP
  //   dup TOS so NOS and TOS are equal
  //   NEXT
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.code @ 0xffff00 <( dAddress -- dData )>
                                                ; // @
  mov       [w0++], w2                          ; // Hmm this supports only 64 k of RAM
  mov       [w0  ], w1                          ; // ... anyway ... move contents of dAddress to dData
  mov       w2, w0

  goto      <*" t.next">                        ; // NEXT
t.endCode

#document t.@.h
  // CODE @ ( d -- d ) // fetch
  //  get dData from dAddress
  //  NEXT
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.code ! 0xffff00 <( dData dAddress -- )>
                                                ; // !
  mov       [--w9], w3                          ; // dData -> w3,w2
  mov       [--w9], w2

  mov       w2, [w0++]                          ; // Hmm this supports only 64 k of RAM
  mov       w3, [w0  ]                          ; // ... anyway ... move dData to dAddress

  mov       [--w9], w1                          ; // And drop an item
  mov       [--w9], w0

  goto      <*" t.next">                        ; // NEXT
t.endCode

#document t.!.h
  // CODE ! ( d d -- ) // store
  //   store dData at dAddress
  //   NEXT
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.code f@ 0xffff00 <( d -- d )>
                                                ; // f@
  mov       TBLPAG, w3                          ; // Save TBLPAG
  mov       w1, TBLPAG                          ; // Set up flash address into TBLPAG, w2
  mov       w0, w2
  tblrdl.w  [w2], w0                            ; // Read two words from flash
  tblrdh.w  [w2], w1
  mov       w3, TBLPAG                          ; // Restore TBLPAG

  goto      <*" t.next">                        ; // NEXT
t.endCode

#document t.f@.h
  // code f@ ( d -- d )
  //   fetch a value from code space (flash)
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.code + 0xffff00 <( d1 d2 -- d )>
```

```
                                                        ; // +
  add       w1, [--w9], w1
  add       w0, [--w9], w0
  addc      w1, #0, w1

  goto      <*" t.next">                                ; // NEXT
t.endCode

#document t.+.h
  // CODE +
  //   add d1 and d2
  //   NEXT
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.code negate 0xffff00 <( d -- d )>
                                                        ; // negate
  neg     w1, w1
  neg     w0, w0
  subb    #0, w1

  goto      <*" t.next">                                ; // NEXT
t.endCode

#document t.negate.h
  // CODE negate
  //   negate TOS
  //   NEXT
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.: - 0xffff00 <( d1 d2 -- d )>
                                                        ; // -
  .pword    <*" t.negate">                              ; // negate
  .pword    <*" t.+">                                   ; // +
t.;

#document t.-.h
  // : - ( d1 d2 -- d ) negate + ;
  //   subtract d2 from d1
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.code and 0xffff00 <( d1 d2 -- d )>
                                                        ; // and
  and       w1, [--w9], w1
  and       w0, [--w9], w0

  goto      <*" t.next">                                ; // NEXT
t.endCode

#document t.and.h
  // CODE and
  //   and d1 and d2
  //   NEXT
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.code or 0xffff00 <( d1 d2 -- d )>
                                                        ; // or
  ior       w1, [--w9], w1
  ior       w0, [--w9], w0

  goto      <*" t.next">                                ; // NEXT
t.endCode

#document t.or.h
  // CODE or
  //   or d1 and d2
  //   NEXT
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.code xor 0xffff00 <( d1 d2 -- d )>
                                                        ; // xor
  xor       w1, [--w9], w1
  xor       w0, [--w9], w0

  goto      <*" t.next">                                ; // NEXT
t.endCode

#document t.xor.h
  // CODE xor
  //   or d1 and d2
  //   NEXT
#endDoc


// ////////////////////////////////////////////////////////////////////////////
```

```
t.code not 0xffff00 <( d -- d )>
                                                    ; // not
  com       w0, w0
  com       w1, w1

  goto      <*" t.next">                            ; // NEXT
t.endCode

#document t.xor.h
  // CODE xor
  //   or d1 and d2
  //   NEXT
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.code cells 0xffff00 <( d -- d )>
                                                    ; // cells
  sl        w0, w0                                  ; // Shift left low word,
                                                    ; // shift a zero to bit 0
                                                    ; // shift bit 15 into carry
  rlc       w1, w1                                  ; // rotate left high word
                                                    ; // carry into bit 0, bit 15 into carry

  sl        w0, w0                                  ; // And again
  rlc       w1, w1

  goto      <*" t.next">                            ; // NEXT
t.endCode

#document t.cells.h
  // : CELLS ( d -- d ) 2* 2* ;
  //   multiply d by CELL
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.: = 0xffff00 <( d1 d2 -- df )>
                                                    ; // =
  .pword    <*" t.-">                               ; // - 0=
  .pword    <*" t.0=">
t.;

#document t.=.h
  // : = - 0= ; // return true when d1 = d2, false otherwise
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.: =marker 0xffff00 <( d -- df )>
                                                    ; // =marker
  .pword    <*" t.marker.mask">                     ; // marker.mask and marker.mask =
  .pword    <*" t.and">
  .pword    <*" t.marker.mask">
  .pword    <*" t.=">
t.;

#document t.=marker.h
  // : =marker marker.mask and marker.mask = ; // return true when d is a header marker, false otherwise
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.code 2+ 0xffff00 <( d -- d )>
                                                    ; // 2+
  add       w0, #2, w0
  addc      w1, #0, w1

  goto      <*" t.next">                            ; // NEXT
t.endCode

#document t.2+.h
  // code 2_ ( d - d )
  //   add 2 to TOS
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.code 2- 0xffff00 <( d -- d )>
                                                    ; // 2-
  sub       w0, #2, w0
  subb      w1, #0, w1

  goto      <*" t.next">                            ; // NEXT
t.endCode

#document t.2-.h
  // code 2_ ( d - d )
  //   subtract 2 from TOS
#endDoc


// ////////////////////////////////////////////////////////////////////////////

t.: cfa2ffa 0xffff00 <( d -- d )>
```

```
                                                    ; // cfa2ffa
<*" cfa2ffa_loop">:                                 ; // begin

    .pword    <*" t.2-">                            ; //   2-
    .pword    <*" t.dup">                           ; //   dup
    .pword    <*" t.f@">                            ; //   f@
    .pword    <*" t.=marker">                       ; //   =marker
    .pword    <*" t.fbranch">                       ; // until
    .pword    <*" cfa2ffa_loop">                    ; //
  t.;

  #document t.cfa2ffa.h
    : cfa2ffa ( d -- d )                            // Determine Flag Field Address (FFA) from CFA
      begin
        2- dup f@ =marker                           // Serarch backwards from CFA till marker found
      until
    ;
  #endDoc


  // ////////////////////////////////////////////////////////////////////////

  t.: cfa2lfa 0xffff00 <( d -- d )>
                                                    ; // cfa2lfa
    .pword    <*" t.cfa2ffa">                       ; // cfa2ffa 2-
    .pword    <*" t.2-">
  t.;

  #document t.cfa2lfa.h
    // : cfa2lfa ( d -- d ) cfa2ffa 2- ; // Get Link Field Address from CFA
  #endDoc


  // ////////////////////////////////////////////////////////////////////////

  t.: cfa2nfa 0xffff00 <( d -- d )>
                                                    ; // cfa2nfa
    .pword    <*" t.cfa2ffa">                       ; // cfa2ffa 2+
    .pword    <*" t.2+">
  t.;

  #document t.cfa2nfa.h
    // : cfa2nfa ( d -- d ) cfa2ffa 2+ ; // Get Name Field Address from CFA
  #endDoc


  // ////////////////////////////////////////////////////////////////////////

  t.: vallot 0xffff00 <( d -- )>
                                                    ; // vallot
    .pword    <*" t.cells">                         ; // CELLS -> bytes
    .pword    <*" t.vhere">                         ; // vhere
    .pword    <*" t.@">                             ; // @
    .pword    <*" t.+">                             ; // +
    .pword    <*" t.vhere">                         ; // vhere
    .pword    <*" t.!">                             ; // !
  t.;

  #document t.vallot.h
    // : VALLOT ( d -- ) CELLS VHERE @ + VHERE ! ; // allot d cells to variable space
  #endDoc


  // ////////////////////////////////////////////////////////////////////////

  t.: (is) 0xffff00 <( dtoken ddefer -- )>
                                                    ; // (is)
    .pword    <*" t.cfa2pfa">                       ; // cfa2pfa f@ !
    .pword    <*" t.f@">
    .pword    <*" t.!">
  t.;

  #document t.(is).h
    // : (is) ( dtoken ddefer -- ) cfa2pfa f@ ! ;
    // Resolve a deferred word with execution token ddefer to dtokenq
    // i.e. set the deferred word to execute dtoken
  #endDoc


  // ////////////////////////////////////////////////////////////////////////

  t.code noop 0xffff00 <( -- )>
                                                    ; // noop
    goto      <*" t.next">                          ; // NEXT
  t.endCode

  #document t.noop.h
    // code noop ( -- )
    //   wastes a few cycles
  #endDoc


  // ////////////////////////////////////////////////////////////////////////

  t.: definitions 0xffff00 <( -- )>
                                                    ; // definitions
    .pword    <*" t.context">                       ; // context @ current!
    .pword    <*" t.@">
    .pword    <*" t.current">
    .pword    <*" t.!">
```

```
t.;

#document t.definitions.h
  // : definitions ( -- ) context @ current ! ;
  // make compiler vocabulary (current) to the currently topmost
  // search vocabulary (context)
#endDoc


// ////////////////////////////////////////////////////////////////////////

t.variable  vhere   0xffff00 <( -- d )>
t.variable  chere   0xffff00 <( -- d )>
t.variable  last    0xffff00 <( -- d )>
t.variable  current 0xffff00 <( -- d )>
t.variable  context 0xffff00 <( -- d )>
t.variable  up      0xffff00 <( -- d )>
t.defer     (main)  0xffff00 <( -- d )>


// ////////////////////////////////////////////////////////////////////////

        4 t.constant  cell            0xffff00 <( -- d )>  // Target CELL size
  TVCells t.constant  initial.vallot  0xffff00 <( -- d )>  // Define this after all t.variable and t.defer definitions
$00ffff00 t.constant  marker.mask     0xffff00 <( -- d )>


// ////////////////////////////////////////////////////////////////////////

t.: cold 0xffff00 <( -- )>
                                                ; // cold
  .pword    <*" t.doLit">                       ; // VBANK VHERE !
  .pword    <*" VBank">
  .pword    <*" t.vhere">
  .pword    <*" t.!">

  ; // VALLOT all defined target variables
  ; // NOTE :: use this after VHERE was setup

  .pword    <*" t.initial.vallot">              ; // INITIAL.VALLOT VALLOT
  .pword    <*" t.vallot">

  ; // Set CHERE to point to first free FLASH location

  .pword    <*" t.doLit">                       ; // LAST_FLASH CHERE !
  .pword    <*" LAST_FLASH">
  .pword    <*" t.chere">
  .pword    <*" t.!">

  ; // Set LAST to point to last header definition (this one)

  .pword    <*" t.doLit">                       ; // ' COLD LAST !
  .pword    <*" t.cold.h">
  .pword    <*" t.last">
  .pword    <*" t.!">

  .pword    <*" t.last">                        ; // LAST @ CONTEXT !
  .pword    <*" t.@">
  .pword    <*" t.context">
  .pword    <*" t.!">

  .pword    <*" t.definitions">                 ; // DEFINITIONS ( context @ current ! )

  .pword    <*" t.doLit">                       ; // 0 UP !
  .pword    0
  .pword    <*" t.up">
  .pword    <*" t.!">

  .pword    <*" t.doLit">                       ; // ' (MAIN) IS NOOP
  .pword    <*" t.noop">
  .pword    <*" t.doLit">
  .pword    <*" t.(main)">
  .pword    <*" t.(is)">


  .pword    <*" t.doLit">                       ; // ' noop cfa2ffa drop
  .pword    <*" t.noop">
  .pword    <*" t.cfa2ffa">
  .pword    <*" t.drop">


  .pword    <*" t.(main)">                      ; // (MAIN)

  .pword    <*" t.bye">                         ; // BYE :: Return to caller of forth.start
t.;

#document t.cold.h
  // : COLD ( -- )
  //
  //   VBANK VHERE !
  //   0     UP    !
  //   BYE
  // ;
#endDoc


// ////////////////////////////////////////////////////////////////////////

code t.lastdef ( -- )
                                                ; // lastdef
```

```
    .pascii       "This is the end."

<*" LAST_FLASH">:
  endCode used

  // ////////////////////////////////////////////////////////////////////////

public

  export  forth.start ( -- )

endModule
```