MECRISP Forth Reference Manual

version 1.0 - 21 décembre 2023

Author

- Marc PETREMANN

Content

forth

## ! n addr --

Store n to address.

```
0 variable temperature
32 temperature !
```

# d1 -- d2

Perform a division modulo the current numeric base and transform the rest of the division into a string of characters. The character is dropped in the buffer set to running <#

```
: hh ( c -- adr len)
    base @ >r  hex
    s>d <# # # #>
    r> base !
  ;
 3 hh type  \ display 03
26 hh type  \ display 1a
```

#> n -- addr len

Drop n. Make the pictured numeric output string available as a character string. *addr* and *len* specify the resulting character string.

```
\ display address in format: NNNN-NNNN
: DUMPaddr ( n -- )
    <# # # # #  [char] - hold # # # # #>
    type
  ;
```

#s d1 -- d=0

Converts the rest of d1 to a string in the character string initiated by <#.

```
: EUROS ( d1 --- str len)
    <#
    # #             \ convert € cents
    [char] , hold   \ add char "," to str buffer
    #s #>           \ convert rest after ","
  ;
15630. EUROS type   \ display 156,30 ok
```

'   exec: <space>name -- xt

Skip leading space delimiters. Parse name delimited by a space. Find name and return xt, the execution token for name.

When interpreting, `' xyz EXECUTE` is equivalent to `xyz`.

```
defer xEmit
: vxEmit ( c ---)
    1+ emit ;
' vxEmit is xEmit
```

*   n1 n2 -- n3

Integer multiplication of two numbers.

```
6 3   *    \ push 18 operation 6*3
7 3   *    \ push 21 operation 7*3
-7 3 *     \ push -21
7 -3 *     \ push -21
-7 -3 *    \ push 21
```

*/   n1 n2 n3 -- n4

Multiply n1 by n2 producing the intermediate double-cell result d. Divide d by n3 giving the single-cell quotient n4.

```
5000 1000 4000 */ .     \ display    1250
```

*/MOD   n1 n2 n3 -- n4 n5

Multiply n1 by n2 producing the intermediate double-cell result d. Divide d by n3 producing the single-cell remainder n4 and the single-cell quotient n5.

```
50000 10 4001 */MOD .     \ display    124 3876
```

+   n1 n2 -- n3

Leave sum of n1 n2 on stack.

```
7 15 +     \ leave 22 on stack
```

+!   n addr --

Increments the contents of the memory address pointed to by addr.

```
0 variable valX
15 valX !
1 valX +!
```

```
valX ?   \ display 16
```

+loop   n --

Increment index loop with value n.

Mark the end of a loop `n1 0 do ... n2 +loop`.

```
: loopTest
   100 0 do
       i .
   5 +loop
  ;
loopTest  \ display 0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90
95
```

,   x --

Append x to the current data section.

-   n1 n2 -- n1-n2

Subtract two integers.

```
6 3  - .  \ display 3
-6 3 - .   \ display -9
```

-rot   n1 n2 n3 -- n3 n1 n2

Inverse stack rotation. Same action than `rot rot`

.   n --

Remove the value at the top of the stack and display it as a signed single precision integer.

```
1 .                         \ display 1
1 2 .                       \ display 2  leave 1 on stack
1 2 + .                     \ display 3  addition 1 and 2, leave nothing on the
stack
6 3  * .                    \ display 18
7 3  * 6 3 * + .            \ display 39 operation (7*3)+(6*3)
```

."   -- <string>

The word `."` can only be used in a compiled definition.

At runtime, it displays the text between this word and the delimiting **"** character end of string.

```
: TITLE
    ."      GENERAL MENU" CR
    ."      ============" ;
: line1
    ." 1.. Enter datas" ;
: line2
    ." 2.. Display datas" ;
: last-line
    ." F.. end program" ;
: MENU ( ---)
    title cr cr cr
    line1 cr cr
    line2 cr cr
    last-line ;
```

.digit   u -- char

Converts a digit to a char.

```
1 .digit emit
1 .digit .   \ display: 49
9 .digit .   \ display: 57
hex
a .digit .   \ display: 41
A .digit .   \ display: 41
decimal
```

.s   --

Displays the content of the data stack, with no action on the content of this stack.

```
: myLoopTest
    10 0 do
       i cr .s
    loop
  ;
\ display:
Stack: [1 ] 42  TOS: 0  *>
Stack: [2 ] 42 0  TOS: 1  *>
Stack: [3 ] 42 0 1  TOS: 2  *>
Stack: [4 ] 42 0 1 2  TOS: 3  *>
...
Stack: [10 ] 42 0 1 2 3 4 5 6 7 8  TOS: 9  *>
```

/   n1 n2 -- n3

Divide n1 by n2, giving the single-cell quotient n3.

```
6 3  / .  \ display 2 opération 6/3
7 3  / .  \ display 2 opération 7/3
8 3  / .  \ display 2 opération 8/3
9 3  / .  \ display 3 opération 9/3
```

/mod   n1 n2 -- n3 n4

Divide n1 by n2, giving the single-cell remainder n3 and the single-cell quotient n4.

```
22 7 /MOD . .    \ display    3 1
```

0<   x1 --- fl

Test if x1 is less than zero.

```
3 0< .    \ display: 0
-5 0< .   \ display: -1
```

0<>   n -- fl

Leave -1 if n <> 0

```
4 0<> .      \ display: -1
3 3 - 0<> .  \ display: 0
```

0=   x -- fl

flag is true if and only if x is equal to zero.

```
5 0=      \ push  FALSE on stack
0 0=      \ push  TRUE  on stack
```

1+   n -- n+1

Increments the value at the top of the stack.

```
25 1+ .    \ display 26
-34 1+ .   \ display -33
```

1-   n -- n-1

Decrements the value at the top of the stack.

2!   d addr --

Store double precision value in memory address addr.

2*   n -- n*2

Multiply n by two.

```
4 2* .      \ display: 8
7 2* .      \ display: 14
-15 2* .    \ display: -30
```

2+   n -- n+2

Increments n by two units.

2-   n -- n-2

Subtracts two.

2/   n -- n/2

Divide n by two.

n/2 is the result of shifting n one bit toward the least-significant bit, leaving the most-significant bit unchanged

```
24 2/ .     \ display    12
25 2/ .     \ display    12
26 2/ .     \ display    13
```

2@   addr -- d

Leave on stack double precision value d stored at address addr.

2constant   comp: d -- <name> | -- d

Makes a double constant.

2drop   n1 n2 n3 n4 -- n1 n2

Removes the double-precision value from the top of the data stack.

```
1 2 3 4 2drop    \ leave 1 2 on top of stack
```

2dup   n1 n2 -- n1 n2 n1 n2

Duplicates the double precision value n1 n2.

```
1 2 2dup   \ leave 1 2 1 2 on stack
```

:   comp: -- <word> | exec: --

Skip leading space delimiters. Parse name delimited by a space. Create a definition for name, called a "colon definition". Enter compilation state and start the current definition.

Subsequent execution of `NOM` performs the execution sequence words compiled in his "colon" definition.

After `: NOM`, the interpreter enters compile mode. All non-immediate words are compiled in the definition, the numbers are compiled in literal form. Only immediate words or placed in square brackets (words `[` and `]`) are executed during compilation to help control it.

A "colon" definition remains invalid, ie not inscribed in the current vocabulary, as long as the interpreter did not execute `;` (semi-colon).

```
: NAME   nomex1 nomex2 ... nomexn ;
NAME   \ execute NAME
```

;   --

Immediate execution word usually ending the compilation of a "colon" definition.

```
: NAME
    nomex1 nomex2
    nomexn ;
```

<   n1 n2 -- fl

Leave fl true if n1 < n2

```
4 10 <=    \ leave -1 on stack
4 4  <=    \ leave  0 on stack
4 3  <=    \ leave  0 on stack
```

<#   n --

Marks the start of converting a integer number to a string of characters.

```
\ display address in format: NNNN-NNNN
: DUMPaddr ( n -- )
    s>d <# # # # #  [char] - hold # # # # #>
    type
  ;

\ display byte in format: NN
: DUMPbyte ( c -- )
    s>d <# # # #>
    type
  ;
```

<=   n1 n2 -- fl

Leave fl true if n1 <= n2

```
4 10 <=   \ leave -1 on stack
4 4  <=   \ leave -1 on stack
4 3  <=   \ leave  0 on stack
```

<>   x1 x2 -- fl

flag is true if and only if x1 is different x2.

```
5 5 <>      \ push  FALSE on stack
5 4 <>      \ push  TRUE  on stack
```

=   n1 n2 -- fl

Leave fl true if n1 = n2

```
4 10 =   \ leave  0 on stack
4 4  =   \ leave -1 on stack
```

>   x1 x2 -- fl

Test if x1 is greater than x2.

>=   x1 x2 -- fl

flag is true if and only if x1 is equal x2.

```
5 5 >=     \ push  FALSE on stack
5 4 >=     \ push  TRUE  on stack
```

>body   cfa -- pfa

pfa is the data-field address corresponding to cfa.

>in   -- addr

Number of characters consumed from TIB

```
tib >in @ type
\ display:
tib >in @
```

>link   cfa -- cfa2

Converts the cfa address of the current word into the cfa address of the word previously defined in the dictionary.

```
' dup >link    \ get cfa from word defined before dup
>name type     \ display "XOR"
```

>name   cfa -- nfa len

finds the name field address of a token from its code field address.

```
' words        \ push cfa of 'words' on stack
>name          \ convert cfa of 'words' in nfa field followed by len
type           \ display 'words'
```

>r   S: n -- R: n

Transfers n to the return stack.

This operation must always be balanced with `r>`

```
\ display n in binary format
: b. ( n -- )
    base @ >r
    binary .
    r> base !
  ;
```

?   addr -- c

Displays the content of any variable or address.

```
0 variable score
25 score !
score ?       \ display: 25
```

?do   n1 n2 --

Executes a `do loop` or `do +loop` loop if n1 is strictly greater than n2.

```
DECIMAL
: qd ?DO I LOOP ;
   789   789 qd \
 -9876 -9876 qd \
     5     0 qd \  display: 0 1 2 3 4
```

?dup   n -- n | n n

Duplicate n if n is not nul.

```
55 ?dup     \ copy 55 on stack
 0 ?dup     \ do nothing
```

@ addr -- n

Retrieves the integer value n stored at address addr.

```
TEMPERATURE @
```

abort --

Raises an exception and interrupts the execution of the word and returns control to the interpreter.

abort" comp: --

Displays an error message and aborts any FORTH execution in progress.

```
: abort-test
    if
        abort" stop program"
    then
    ." continue program"
  ;

0 abort-test   \ display: continue program
1 abort-test   \ display: stop program ERROR
```

abs n -- n'

Return the absolute value of n.

```
-7 abs .    \ display: 7
 7 abs .    \ display: 7
```

accept addr n -- n

Accepts n characters from the keyboard (serial port) and stores them in the memory area pointed to by addr.

```
create myBuffer 100 allot
myBuffer 100 accept     \ on prompt, enter: This is an example
myBuffer swap type      \ display: This is an example
```

again --

Mark the end on an infinit loop of type `begin ... again`

```
: test ( -- )
    begin
        ." Diamonds are forever" cr
    again
```

```
 ;
```

align   --

Align the current data section dictionary pointer to cell boundary.

aligned   addr1 -- addr2

addr2 is the first aligned address greater than or equal to addr1.

allot   n --

Reserve n address units of data space.

```
create myDatas 128 allot
```

and   n1 n2 --- n3

Execute logic AND.

The words `AND`, `OR`, and `XOR` perform operations binary **bitwise** logic on single-precision integers at the top of the data stack.

```
0   0 and .      \ display 0
0   -1  and .    \ display 0
-1   0 and .     \ display 0
-1   -1  and .   \ display -1
```

arshift   x1 u -- x2

Arithmetic right-shift of u bit-places.

base   -- addr

Single precision variable determining the current numerical base.

The `BASE` variable contains the value 10 (decimal) when FORTH starts.

```
DECIMAL      \ select decimal base
2 BASE !     \ selevt binary base

\ other example
: GN2 \ ( -- 16 10 )
   BASE @ >R HEX BASE @ DECIMAL BASE @ R> BASE !
  ;
```

begin   --

Marks the start of a structure:

- begin..again

- begin..while..repeat

- begin..until

bic   x1 x2 -- x3

Bit clear, identical to `not and`.

bic!   mask addr --

Clear bit in word-location.

binary   --

Select current base to 2.

bis!   mask addr --

Set bit in word-location.

bit@   mask addr -- flag

Test bit in word-location.

bl   -- 32

Value 32 on stack.

```
\ definition of bl
: bl   ( -- 32 )
    32
  ;
```

c!   c addr --

Stores an 8-bit c value at address addr.

c+!   c c-addr --

Add to byte memory location

c,   c --

Append c to the current data section.

NOT AVAILABLE ON ALL MCU'S

```
create myDatas
    36 c,   42 c,  24 c,  12 c,
myDatas 1+ c@    \ push 42 on stack
```

c@ addr -- c

Retrieves the 8-bit c value stored at address addr.

```
35 constant PINB   \ adresse registre données PIN de PORT B sur Arduino
PINB c@            \ empile contenu registre pointé par PINB
```

case --

```
: day ( n -- addr len )
    CASE
        0 OF s" Sunday"     ENDOF
        1 OF s" Monday"     ENDOF
        2 OF s" Tuesday"    ENDOF
        3 OF s" Wednesday"  ENDOF
        4 OF s" Thursday"   ENDOF
        5 OF s" Friday"     ENDOF
        6 OF s" Saturday"   ENDOF
    ENDCASE
  ;
```

cbic! mask c-addr --

Clear bit in byte-location.

cbis! mask c-addr --

Set bit in byte-location.

cbit@ mask c-addr -- flag

Test bit in byte-location.

cell+ n -- n'

Increment n...

```
10 cell+ .     \ display: 14
```

cells n -- n'

Multiply n by the size of an integer (32 bits = 4 bytes).

Allows you to position yourself in an array of integers.

```
1 cells .      \ display 4
6 cells .      \ display 24
```

char -- <string>

Word used in interpretation only.

Leave the first character of the string following this word.

```
char v .            \ display: 118 (ascii code for "v")
char house .        \ display: 104 - code for "h"
```

clz x1 -- u

Count leading zeros.

code -- <:name>

Defines a word whose definition is written in assembly language.

```
code my2*
  a1 32 ENTRY,
  a8 a2 0 L32I.N,
  a8 a8 1 SLLI,
  a8 a2 0 S32I.N,
  RETW.N,
end-code
```

compiletoflash --

Put Mecrisp-Stellaris into “save any new words to flash mode”

```
compiletoflash

: closed.loop.program      \ runs forever
  begin
  ." This MCU is running a TURNKEY application" cr
  again
;

: INIT
  closed.loop.program
;

compiletoram
```

compiletoram --

makes ram the target for compiling

compiletoram? -- fl

Stack TRUE if currently compile into ram.

constant comp: n -- <name> | exec: -- n

Define a constant.

```
binary  \ masks are 32 bit length
00000000000000000000000000000100 constant ledGREEN      \  green LED on G2
00000000001000000000000000000000 constant ledYELLOW     \ yellow LED on G21
00000000000000100000000000000000 constant ledRED        \    red LED on G17
decimal
```

cr --

Show a new line return.

```
: .result ( ---)
    ." Port analys result" cr
    . "pool detectors" cr ;
```

create comp: -- <name> | exec: -- addr

The word **CREATE** can be used alone.

The word after **CREATE** is created in the dictionary, here **DATAS**. The execution of the word thus created deposits on the data stack the memory address of the parameter zone. In this example, we have compiled 4 8-bit values. To recover them, it will be necessary to increment the address stacked with the value shifting the data to be recovered.

```
\ Peripherals accessed by the CPU via 0x3FF40000 ~ 0x3FF7FFFF address space
\ (DPORT address) can also be accessed via 0x60000000 ~ 0x6003FFFF
\ (AHB address). (0x3FF40000 + n) address and (0x60000000 + n)
\ address access the same content, where n = 0 ~ 0x3FFFF.
create uartAhbBase
    $60000000 ,
    $60010000 ,
    $6002E000 ,

: REG_UART_AHB_BASE { idx -- addr }     \ id=[0,1,2]
    uartAhbBase  idx cell *  + @
  ;
```

cxor! mask c-addr --

Toggle bit in byte-location.

cycles -- u

Stack the content pointed by **TIMERAWL**

d. d --

Print double number.

decimal --

Selects the decimal number base. It is the default digital base when FORTH starts.

```
HEX
FF DECIMAL .   \ display 255
```

depth -- n

n is the number of single-cell values contained in the data stack before n was placed on the stack.

```
\ test this after reset:
depth       \ leave 0 on stack
10 32 25
depth       \ leave 3 on stack
```

dictionarystart -- addr

Current entry point for dictionary search.

digit char -- u t|f

Converts a char to a digit.

dint --

Disables Interrupts

do n1 n2 --

Set up loop control parameters with index n2 and limit n1.

```
: testLoop
    256 32 do
        I emit
    loop
  ;
```

DOES> comp: -- | exec: -- addr

The word `CREATE` can be used in a new word creation word...

Associated with `DOES>`, we can define words that say how a word is created then executed.

drop n --

Removes the single-precision integer that was there from the top of the data stack.

```
2 5 8  drop   \ leave 2 and 5 on stack
```

dump a n --

Dump a memory region

```
here 32 - 32 dump
\ dump memory between here-20 --> here
```

dump-file-delete addr len addr2 len2 --

Transfers the contents of a text string addr len to a file pointed by addr2 len2

dup n -- n n

Duplicates the single-precision integer at the top of the data stack.

```
: SQUARE ( n --- nE2)
    DUP * ;
5 SQUARE .     \ display 25
10 SQUARE .    \ display 100
```

eint --

Enables Interrupts

else --

Word of immediate execution and used in compilation only. Mark a alternative in a control structure of the type `IF ... ELSE ... THEN`

At runtime, if the condition on the stack before `IF` is false, there is a break in sequence with a jump following `ELSE`, then resumed in sequence after `THEN`.

```
: TEST ( ---)
    CR ." Press a key " KEY
    DUP 65 122 BETWEEN
    IF
        CR 3 SPACES ." is a letter "
    ELSE
        DUP 48 57 BETWEEN
        IF
            CR 3 SPACES ." is a digit "
        ELSE
            CR 3 SPACES ." is a special character "
        THEN
    THEN
    DROP ;
```

emit   x --

If x is a graphic character in the implementation-defined character set, display x.

The effect of `EMIT` for all other values of x is implementation-defined.

When passed a character whose character-defining bits have a value between hex 20 and 7F inclusive, the corresponding standard character is displayed. Because different output devices can respond differently to control characters, programs that use control characters to perform specific functions have an environmental dependency. Each `EMIT` deals with only one character.

```
65 emit    \ display A
66 emit    \ display B
```

emit?   -- fl

Ready to send a character.

endcase   --

Marks the end of a `CASE OF ENDOF ENDCASE` structure

```
: day ( n -- addr len )
    CASE
        0 OF s" Sunday"     ENDOF
        1 OF s" Monday"     ENDOF
        2 OF s" Tuesday"    ENDOF
        3 OF s" Wednesday"  ENDOF
        4 OF s" Thursday"   ENDOF
        5 OF s" Friday"     ENDOF
        6 OF s" Saturday"   ENDOF
    ENDCASE
  ;
```

endof   --

Marks the end of a `OF .. ENDOF` choice in the control structure between `CASE ENDCASE`.

```
: day ( n -- addr len )
    CASE
        0 OF s" Sunday"     ENDOF
        1 OF s" Monday"     ENDOF
        2 OF s" Tuesday"    ENDOF
        3 OF s" Wednesday"  ENDOF
        4 OF s" Thursday"   ENDOF
        5 OF s" Friday"     ENDOF
        6 OF s" Saturday"   ENDOF
    ENDCASE
  ;
```

eraseflash --

Erases everything. Clears Ram. Restarts Forth.

evaluate addr len --

Evaluate the content of a string.

```
s" words"
evaluate   \ execute the content of the string, here: words
```

even u1|n1 -- u2|n2

Makes even. Adds one if uneven.

```
3 even .   \ display: 4
6 even .   \ display: 6
```

EXECUTE addr --

Execute word at addr.

Take the execution address from the data stack and executes that token. This powerful word allows you to execute any token which is not a part of a token list.

exit --

Aborts the execution of a word and gives back to the calling word.

Typical use: `: X ... test IF ... EXIT THEN ... ;`

At run time, the word `EXIT` will have the same effect as the word `;`

extract n base -- n c

Extract the least significant digit of n. Leave on the stack the quotient of n/base and the ASCII character of this digit.

false -- 0

Leave 0 on stack.

fill   addr len c --

If len is greater than zero, store c in each of len consecutive characters of memory beginning at addr.

find   addr len -- xt | 0

Find a word in dictionnary.

```
32 string t$
s" vlist" t$ $!
t$ find    \ push cfa of VLIST on stack
```

floor   r1 -- r2

Rounds a real down to the integer value.

```
45,67 floor f.   \ display 45,00000000000000000000000000000000
```

for   n --

Marks the start of a loop `for .. next`

WARNING: the loop index will be processed in the interval [n..0], i.e. n+1 iterations, which is contrary to the other versions of the FORTH language implementing FOR..NEXT (FlashForth).

```
: myLoop ( ---)
    10 for
        r@ . cr   \ display loop index
    next
  ;
```

h!   char c-addr --

Stores halfword in memory

h+!   u|n h-addr --

Add n to halfword memory location.

h@   c-addr - - char

Fetches halfword from memory

hbic!   mask h-addr --

Clear bit in halfword-location.

hbis! mask h-addr --

Set bit in halfword-location.

hbit@ mask h-addr -- flag

Test bit in halfword-location.

here -- addr

Leave the current data section dictionary pointer.

The dictionary pointer is incremented as the words are compiled and variables and data tables are defined.

```
here u.        \ display 1073709120
: null ;
here u.        \ display 1073709144
```

hex --

Selects the hexadecimal digital base.

```
255 HEX .    \ display FF
DECIMAL      \ return to decimal base
```

hex. u --

Prints integer bit unsigned in hex base, needs emit only. This is independent of number base.

```
3 hex.    \ display: 00000003
 12 hex.    \ display: 0000000C
259 hex.    \ display: 00000103
```

hld -- addr

Pointer to text buffer for number output.

hold c --

Inserts the ASCII code of an ASCII character into the character string initiated by <#.

hxor! mask h-addr --

Toggle bit in halfword-location.

i -- n

n is a copy of the current loop index.

```
: mySingleLoop ( -- )
    cr
    10 0 do
        i .
    loop
  ;
mySingleLoop
\ display 0 1 2 3 4 5 6 7 8 9
```

idle   task --

Idle a random task (IRQ save).

if   fl --

The word `IF` is executed immediately.

`IF` marks the start of a control structure for type `IF..THEN` or `IF..ELSE..THEN`.

```
: WEATHER? ( fl ---)
    IF
        ." Nice weather "
    ELSE
        ." Bad weather "
    THEN ;
1 WEATHER?    \ display: Nice weather
0 WEATHER?    \ display: Bad weather
```

immediate   --

Make the most recent definition an immediate word.

Sets the compile-only lexicon bit in the name field of the new word just compiled. When the interpreter encounters a word with this bit set, it will not execute this word, but spit out an error message. This bit prevents structure words to be executed accidentally outside of a compound word.

```
: myWord
    ." *** HELLO ***"  ; immediate
: myTest
    myWord
    ." *** TEST *** ;
\ display: *** HELLO ***  during compilation of myTest
myTest     \ display: *** TEST ***
```

ipsr   -- ipsr

Interrupt Program Status Register

is --

Affecte le code d'exécution d'un mot à un mot d'exécution vectorisée.

```
defer xEmit
: vxEmit ( c ---)
    1+ emit ;
' vxEmit is xEmit
```

j -- n

n is a copy of the next-outer loop index.

```
: myDoubleLoop ( -- )
    cr
    10 0 do
        cr
        10 0 do
            i 1+  j 1+  * .
        loop
    loop
  ;
myDoubleLoop
\ display:
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
4 8 12 16 20 24 28 32 36 40
5 10 15 20 25 30 35 40 45 50
6 12 18 24 30 36 42 48 54 60
7 14 21 28 35 42 49 56 63 70
8 16 24 32 40 48 56 64 72 80
9 18 27 36 45 54 63 72 81 90
10 20 30 40 50 60 70 80 90 100
```

k -- n

n is a copy of the next-next-outer loop index.

```
: myTripleLoop ( -- )
    cr
    5 0 do
        cr
        5 0 do
            cr
            5 0 do
                i 1+  j 1+  k 1+  * * .
            loop
        loop
    loop
  ;
myTripleLoop
```

key -- char

Waits for a key to be pressed. Pressing a key returns its ASCII code.

```
key .    \ display 97 if key "a" is active
key .    \ affiche 65 if key "A" is active
```

key? -- fl

Returns *true* if a key is pressed.

```
: keyLoop
    begin
    key? until
  ;
```

list --

Shows all the words in the FORTH dictionary.

```
list   \ display:
--- Mecrisp-Stellaris RA 2.6.5 --- 2dup 2drop 2swap 2nip 2over 2tuck 2rot
2-rot 2>r 2r> 2r@ 2rdrop d2/ d2* dshr dshl dabs dnegate d- d+ s>d um*.....
```

literal x --

Compiles the value x as a literal value.

```
: valueReg ( --- n)
   [ 36 2 * ] literal ;

\ equivalent to:
: valueReg ( --- n)
   72 ;
```

log10 fn -- log(fn)

Calculates the base 10 logarithm of fn.

```
100, log10 f.   \ display: 2,0000000000000000000000000000000
 10, log10 f.   \ display: 1,0000000000000000000000000000000
  2, log10 f.   \ display: 0,30102999554947018623352050781250
```

loop   --

Add one to the loop index. If the loop index is then equal to the loop limit, discard the loop parameters and continue execution immediately following the loop. Otherwise continue execution at the beginning of the loop.

```
: ascii-chars ( -- )
    128 32 do
        i emit
    loop
  ;
```

lshift   x1 n -- x2

Shift x1 left by n bits.

```
2 3 lshift .    \ display: 16
```

max   n1 n2 -- n1|n2

Leave the unsigned larger of u1 and u2.

```
3  10 max .     \ display 10
-3 -10 max .     \ display -3
```

min   n1 n2 -- n1|n2

Leave min of n1 and n2

```
10 2 min .     \ display: 2
2 10 min .     \ display: 2
2 -10 min .    \ display: -10
```

mod   n1 n2 -- n3

Divide n1 by n2, giving the single-cell remainder n3.

The modulo function can be used to determine the divisibility of one number by another.

```
21 7 mod . \ display 0
22 7 mod . \ display 1
23 7 mod . \ display 2
24 7 mod . \ display 3

: DIV? ( n1 n2 ---)
    OVER OVER MOD CR
    IF
        SWAP . ." is not "
    ELSE
```

```
        SWAP . ." is "
    THEN
    ." divisible by " .
  ;
```

move   c-addr1 c-addr2 u --

If u is greater than zero, copy u consecutive characters from the data space starting at c-addr1 to that starting at c-addr2, proceeding character-by-character from lower addresses to higher addresses.

ms   n --

Waiting in millisencondes.

For long waits, set a wait word in seconds.

```
500 ms \ detay for 1/2 second

: seconds ( n --)
    0
    for
        1000 ms
    next
  ;
12 seconds \ delay for 12 seconds
```

n.   n --

Display any value n in decimal format.

```
hex 3F n.   \ display: 63
```

negate   n -- -n'

Two's complement of n.

```
7 negate .    \ display: -7
-8 negate .   \ display: 8
```

nip   n1 n2 -- n2

Remove n1 from the stack.

```
10 25 nip .   \ display 10
```

nl -- 10

Value 10 on stack.

`nl` is the code used by eForth as a line ending in the Forth source code.

```
nl .   \ display 10
```

nop --

No operation.

Hook for unused handlers !

normal --

Disables selected colors for display.

not x1 -- x2

Invert all bits.

of n --

Marks a `OF .. ENDOF` choice in the control structure between `CASE ENDCASE`

If the tested value is equal to the one preceding `OF`, the part of code located between `OF ENDOF` will be executed.

```
: day ( n -- addr len )
    CASE
        0 OF s" Sunday"     ENDOF
        1 OF s" Monday"     ENDOF
        2 OF s" Tuesday"    ENDOF
        3 OF s" Wednesday"  ENDOF
        4 OF s" Thursday"   ENDOF
        5 OF s" Friday"     ENDOF
        6 OF s" Saturday"   ENDOF
    ENDCASE
  ;
```

or n1 n2 -- n3

Execute logic OR.

The words `AND`, `OR`, and `XOR` perform operations binary **bitwise** logic on single-precision integers at the top of the data stack.

```
0  -1    or  .  \ display 0
0  -1    or  .  \ display -1
-1   0   or  .  \ display -1
```

```
-1   -1  or  .  \ display -1
```

over   n1 n2 -- n1 n2 n1

Place a copy of n1 on top of the stack.

```
2 5 OVER  \ duplicate 2 on top of the stack
```

PARSE   c "string" -- addr count

Parse the next word in the input stream, terminating on character c. Leave the address and character count of word. If the parse area was empty then count=0.

pause   --

Yield to other tasks.

pi/2   -- pi/2

Stacks the value of pi/2, which corresponds to a 90° angle expressed in radians.

```
pi/2 f.   \ display: 1,570796326734125614166259765625OO
```

pi/4   -- pi/4

Stacks the value of pi/4, which corresponds to a 45° angle expressed in radians.

```
pi/2 f.   \ display: 1,570796326734125614166259765625OO
```

pow10   fn -- 10exp-fn

Raise 10 to the power fn.

```
3, pow10 f.   \ display: 1000,000000000000000000000000000000
2,4 pow10 f.  \ display: 251,188648164272308349609375000000000
```

pow2   real1 -- real2

Square a real number.

```
4,0 pow2 f.
\ display: 16,000000000000000000000000000000
5,1 pow2 f.
\ display: 34,296750791370868682861328125000000
```

prompt   --

Displays an interpreter availability text. Default poster:

ok

```
prompt   \ display: ok
```

r>   R: n -- S: n

Transfers n from the return stack.

This operation must always be balanced with >r

```
\ display n in binary format
: b. ( n -- )
    base @ >r
    binary .
    r> base !
  ;
```

rdepth   -- n

Gives number of return stack items.

rdrop   --

Removes the value that is at the top of the return stack.

recurse   --

Append the execution semantics of the current definition to the current definition.

The usual example is the coding of the factorial function.

```
: FACTORIAL ( +n1 -- +n2)
   DUP 2 < IF DROP 1 EXIT THEN
   DUP 1- RECURSE *
;
```

registerlist.   --

Display registers list.

```
registerlist.    \ display:  r1  r3  r5
```

repeat   --

End a indefinite loop `begin.. while.. repeat`

reset   --

Reset on hardware level.

rm -- "path"

Delete the file designed in file path.

rol x1 -- x2

Logical left-rotation of one bit-place.

ror x1 -- x2

Logical right-rotation of one bit-place.

rot n1 n2 n3 -- n2 n3 n1

Rotate three values on top of stack.

rp! addr --

Store return stack pointer.

rp@ -- addr

Fetch return stack pointer.

RSHIFT x1 u -- x2

Right shift of the value x1 by u bits.

```
64 2 rshift .  \ display 16
```

s" comp: -- <string> | exec: addr len

In interpretation, leaves on the data stack the string delimited by "

In compilation, compiles the string delimited by "

When executing the compiled word, returns the address and length of the string...

```
\ header for DUMP
: headDump
    s" --addr----- 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F"
  ;
headDump        \ push addr len on stack
headDump type   \ display: --addr----- 00 01 02 03 04 05 06 07 08 09 0A 0B
0C 0D 0E 0F
```

save ???

???

SCR-delete -- addr

Variable pointing to the block being edited.

see -- name>

Decompile or disassemble a FORTH definition.

shl x1 -- x2

Logical left-shift of one bit-place.

shr x1 -- x2

Logical right-shift of one bit-place.

sp! addr --

Store data stack pointer.

sp0 -- addr

Points to the bottom of Forth's parameter stack (data stack).

```
sp0 .    \ display addr of stack first addr
```

sp@ -- addr

Push on stack the address of data stack.

space --

Display one space.

```
\ definition of space
: space   ( -- )
    bl emit
  ;
```

spaces n --

Displays the space character n times.

Defined since version 7.071

sqft fn -- sqrt(fn)

Calculate the square root of the real number fn.

```
2, sqrt f.   \ display: 1,41421356191858649253845214843750
```

stackspace -- 512

Constant. value 512, or 128 elements for each task.

state -- fl

Compilation state. State can only be changed by `[` and `]`.

-1 for compiling, 0 for interpreting

stop --

Stop current task.

swap n1 n2 -- n2 n1

Swaps values at the top of the stack.

```
2 5 SWAP
.   \ display 2
.   \ display 5
```

task: -- <name>

Create a new task.

```
task: blinker

: blink& ( -- )
  blinker activate
  begin
    LED1 iox!    \ toggle LED1
    200 ms       \ wait 200 ms
  again ;
```

tasks --

Show tasks currently in round-robin list.

then --

Immediate execution word used in compilation only. Mark the end a control structure of type `IF..THEN` or `IF..ELSE..THEN` .

tib -- addr

returns the address of the the terminal input buffer where input text string is held.

```
tib >in @ type
\ display:
```

```
tib >in @
```

TIMEHR -- $40054008

Constant. Value $40054008.

TIMEHW -- $40054000

Constant. Value $40054000.

TIMELR -- $4005400C

Constant. Value $4005400C.

TIMELW -- $40054004

Constant. Value $40054004.

TIMERAWH -- $40054024

Constant. Value $40054024.

TIMERAWL -- $40054028

Constant. Value $40054028.

The two registers `TIMERAWH` and `TIMERAWH` point to 64-bit content of the timer.

true -- -1

Leave -1 on stack.

type addr c --

Display the string characters over c bytes.

```
: hello ( --- addr c)
s" Hello world" ;
hello type               \ display: Hello world
hello drop 5 type        \ display: Hello
```

u. n --

Removes the value from the top of the stack and displays it as an unsigned single precision integer.

```
1 U.      \ display 1
-1 U.     \ display 65535
```

U/MOD   u1 u2 -- rem quot

Unsigned int/int->int division.

```
-25 3 U/MOD  \ leave on stack: 0 6148914691236517197
```

ud.   ud --

Print unsigned double number.

unloop   --

Stop a do..loop action. Using `unloop` before `exit` only in a do..loop structure.

```
: example ( -- )
    100 0 do
        cr i .
        key bl = if
            unloop exit
        then
    loop
  ;
```

until   fl --

End of `begin.. until` structure.

```
: myTestLoop ( -- )
    begin
        key dup .
        [char] A =
    until
  ;
myTestLoop \ end loop if key A pressed
```

unused   -- free-mem

Displays memory depending on compile mode (Ram or Flash).

us   u --

Generates a timeout of u microseconds.

use   -- <name>

Use "name" as the blockfile.

```
USE /spiffs/foo
```

used -- n

Specifies the space taken up by user definitions. This includes already defined words from the FORTH dictionary.

variable comp: -- <name> | exec: -- addr

Creation word. Defines a simple precision variable.

```
0 variable speed
75 speed !    \ store 75 in speed
speed @ .     \ display 75
```

wake task --

Wake a random task (IRQ save).

wfi --

Compile only. Wait For Interrupt, enters sleep mode.

while fl --

Mark the conditionnal part execution of a structure `begin..while..repeat`

words --

List the definition names in the first word list of the search order. The format of the display is implementation-dependent.

```
words
\ display content of FORTH dictionnary
```

xor n1 n2 -- n3

Execute logic eXclusif OR.

The words `AND`, `OR`, and `XOR` perform operations binary **bitwise** logic on single-precision integers at the top of the data stack.

```
0 -1 xor .       \ display 0
0 -1  xor .      \ display -1
-1  0 xor .      \ display -1
-1  0  xor .     \ display 0
```

xor! mask addr --

Toggle bit in word-location.

[--

Enter interpretation state. [is an immediate word.

```
\ source for [
: [
    0 state !
    ; immediate
```

['] comp: -- <name> | exec: -- addr

Use in compilation only. Immediate execution.

Compile the cfa of <name>

```
serial \ Select Serial vocabulary

: serial2-type ( a n -- )
    Serial2.write drop ;

: typeToLoRa ( -- )
    0 echo !     \ disable display echo from terminal
    ['] serial2-type is type
  ;

: typeToTerm ( -- )
    ['] default-type is type
    -1 echo !    \ enable display echo from terminal
  ;
```

[char] comp: -- <spaces>name | exec: -- xchar

Place xchar, the value of the first xchar of name, on the stack.

```
: GC1 [CHAR] X     ;
: GC2 [CHAR] HELLO ;
GC1 \ empile 58
GC2 \ empile 48
```

] --

Return to compilation.] is an immediate word.

With FlashForth, the words [and] allow you to use assembly code, subject to first compiling an assembler.

```
\ Load constant $1234 to top of stack
: a-number ( -- 1234 )
    dup                    \ Make space for new TOS value
    [ R24 $34 ldi, ]
    [ R25 $12 ldi, ]
;
```