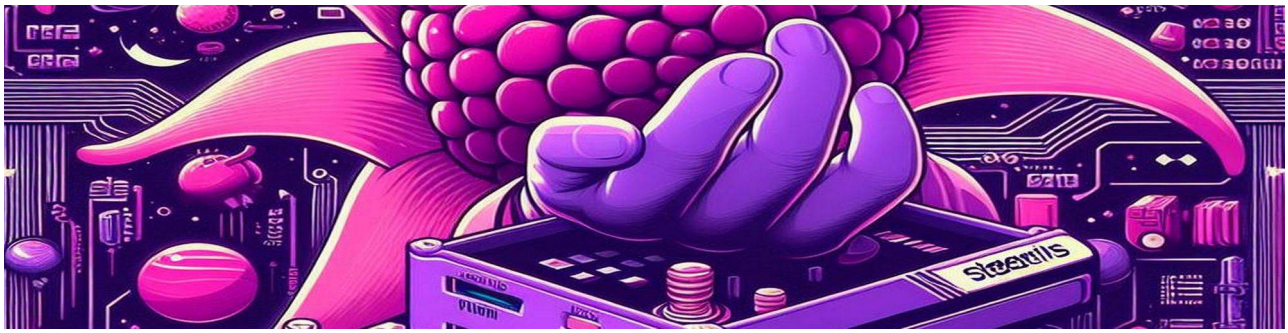
# Manuel de référence pour MECRISP Forth

**version 1.1 - 25 décembre 2023**



Auteur

- Marc PETREMANN

# Table des matières

# forth

## ! n addr --

Stocke une valeur entière n à l'adresse addr.

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

## # d1 -- d2

Effectue une division modulo la base numérique courante et transforme le reste de la division en chaîne de caractère. Le caractère est déposé dans le tampon définit à l'exécution de <#

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

## #> n -- addr len

Dépile n. Rend la chaîne de sortie numérique mise en forme sous forme de chaîne de caractères. *addr* et *len* spécifient la chaîne de caractères résultante.

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

## #s d1 -- d=0

Convertit le reste de d1 en chaîne de caractères dans la chaîne de caractères initiée par <#.

```
: 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

Recherche <name> et laisse son code d'exécution (adresse).

En interprétation, `' xyz EXECUTE` équivaut à `xyz`.

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

## * n1 n2 -- n3

Multiplication entière de deux nombres.

```
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

Multiplie n1 par n2 produisant le résultat intermédiaire à double précision d. Divise d par n3 en donnant le quotient entier n4.

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

## */mod n1 n2 n3 -- n4 n5

Multiplie n1 par n2 produisant le résultat intermédiaire à double précision d. Divise d par n3 produisant le reste entier n4 et le quotient entier n5.

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

## + n1 n2 -- n3

Laisse la somme de n1 et n2 sur la pile.

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

## +! n addr --

Incrémente le contenu de l'adresse mémoire pointé par addr.

```
0 variable valX
15 valX !
1 valX +!
valX ?   \ display 16
```

## +loop n --

Incrémente l'index de boucle de n.

Marque la fin d'une boucle `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 --

Ajoute x à la section de données actuelle.

## - n1 n2 -- n1-n2

Soustration de deux entiers.

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

## -rot n1 n2 n3 -- n3 n1 n2

Rotation inverse de la pile. Action similaire à `rot rot`

## . n --

Dépile la valeur au sommet de la pile et l'affiche en tant qu'entier simple précision signé.

```
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>

Le mot `."` est utilisable exclusivement dans une définition compilée.

A l'exécution, il affiche le texte compris entre ce mot et le caractère " délimitant la fin de chaîne de caractères.

```
: 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

Convertit un chiffre en caractère.

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

## .s --

Affiche le contenu de la pile de données, sans action sur le contenu de cette pile.

```
: 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

Division de deux entiers. Laisse le quotient entier sur la pile.

```
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

Divise n1 par n2, donnant le reste entier n3 et le quotient entier n4.

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

## 0< x1 --- fl

Teste si x1 est inférieur à zéro.

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

## 0<> n -- fl

Empile -1 si n <> 0

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

## 0= x -- fl

Teste si l'entier simple précision situé au sommet de la pile est nul.

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

## 0to1-atan x -- atanx

Calculer atan pour s31,32 x dans l'intervalle [0, 1].

```
0,3 0to1-atan f.   \ display:
0,291456793900579214096069335937500
```

## 0to1sqrt x -- sqrtx

Prenez la racine carrée du nombre s31.32 x avec x dans l'intervalle [0, 1].

```
0,22 0to1sqrt f.  \ display:
0,469041575910523533821105957031250
```

## 1+ n -- n+1

Incrémente la valeur située au sommet de la pile.

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

## 1- n -- n-1

Décrémente la valeur située au sommet de la pile.

## 2! d addr --

Stocke la valeur double précision d à l'adresse addr.

## 2* n -- n*2

Multiplie n par deux.

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

## 2+ n -- n+2

Incrémente n de deux unités.

## 2- n -- n-2

Soustrait deux.

## 2/ n -- n/2

Divise n par deux.

n/2 est le résultat du décalage de n d'un bit vers le bit le moins significatif, laissant le bit le plus significatif inchangé.

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

## 2@ addr -- d

Empile la valeur double précision d stockée à l'adresse addr.

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

Crée une constante double précision.

## 2drop n1 n2 n3 n4 -- n1 n2

Retire la valeur double précision du sommet de la pile de données.

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

## 2dup n1 n2 -- n1 n2 n1 n2

Duplique la valeur double précision n1 n2.

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

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

Ignore les délimiteurs d'espace de début. Analyse le nom délimité par un espace. Crée une définition pour le , appelée "définition deux-points". Entre dans l'état de compilation et démarre la définition actuelle.

L'exécution ultérieure de `NOM` réalise l'enchainement d'exécution des mots compilés dans sa définition "deux-points".

Après `: NOM`, l'interpréteur entre en mode compilation. Tous les mots non immédiats sont compilés dans la définition, les nombres sont compilés sous forme litérale. Seuls les mots immédiats ou placés entre crochets (mots `[` et `]`) sont exécutés pendant la compilation pour permettre de contrôler celle-ci.

Une définition "deux-points" reste invalide, c'est à dire non inscrite dans le vocabulaire courant, tant que l'interpréteur n'a pas exécuté `;` (point-virgule).

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

## ; --

Mot d'exécution immédiate terminant habituellement la compilation d'une définition "deux-points".

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

## < n1 n2 -- fl

Laisse fl vrai si n1 < n2

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

## <# n --

Marque le début de la conversion d'un nombre entier en chaîne de caractères.

```
\ 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

Laisse fl vrai si n1 <= n2

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

## <>  x1 x2 -- fl

Teste si l'entier simple précision x1 n'est pas égal à x2.

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

## =  n1 n2 -- fl

Laisse fl vrai si n1 = n2

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

## >  x1 x2 -- fl

Teste si x1 est supérieur à x2.

## >=  x1 x2 -- fl

Teste si l'entier simple précision x1 est égal à x2.

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

## >body  cfa -- pfa

convertit l'adresse cfa en adresse pfa (Parameter Fieds Address)

## >in -- addr

Nombre de caractères consommés depuis TIB

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

## >link cfa -- cfa2

Convertit l'adresse cfa du mot courant en adresse cfa du mot précédemment défini dans le dictionnaire.

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

## >name cfa -- nfa len

trouve l'adresse du champ de nom d'un mot à partir de son adresse de champ de code cfa.

```
' 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

Transfère n vers la pile de retour.

Cette opération doit toujours être équilibrée avec `r>`

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

## ? addr -- c

Affiche le contenu d'une variable ou d'une adresse quelconque.

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

## ?do n1 n2 --

Exécute une boucle `do loop` ou `do +loop` si n1 est strictement supérieur à 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

Duplique n si n n'est pas nul.

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

## @ addr -- n

Récupère la valeur entière n stockée à l'adresse addr.

```
TEMPERATURE @
```

## abort --

Génère une exception et interrompt l'exécution du mot et rend la main à l'interpréteur.

## abort" comp: --

Affiche un message d'erreur et interrompt toute exécution FORTH en cours.

```
: 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'

Renvoie la valeur absolue de n.

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

## accept addr n -- n

Accepte n caractères depuis le clavier (port série) et les stocke dans la zone mémoire pointée par addr.

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

## acos x -- acosx

Calcule acos pour s31,32 x dans l'intervalle [-1, 1].

Renvoie le résultat en degrés.

```
0,5 acos f>s .   \ display: 60
```

## again --

Marque la fin d'une boucle infinie de type `begin ... again`

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

## align --

Aligne le pointeur du dictionnaire de la section de données actuelle sur la limite de la cellule.

## aligned addr1 -- addr2

addr2 est la première adresse alignée plus grande ou égale à addr1.

## allot n --

Réserve n adresses dans l'espace de données.

```
create myDatas 128 allot
```

## and n1 n2 --- n3

Effectue un ET logique.

Les mots `AND`, `OR` et `XOR` effectuent des opérations logiques binaires **bit à bit** sur les entiers simple précision situés au sommet de la pile de données.

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

## arshift x1 u -- x2

Décalage arithmétique vers la droite de u bits.

## asin x -- asinx

Calcule asin pour s31,32 x dans l'intervalle [-1, 1].

Renvoie le résultat en degrés.

```
0,212 asin f>s .   \ display: 12
```

## atan x -- atanx

Calc atan pour s31,32 x, renvoie le résultat en degrés.

```
1,0 atan f.   \ display: 44,999999998137354850769042968750000
```

## base -- addr

Variable simple précision déterminant la base numérique courante.

La variable **BASE** contient la valeur 10 (décimal) au démarrage de FORTH.

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

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

## begin --

Marque le début d'une structure:

- begin..again

- begin..while..repeat

- begin..until

## beq --

Branchement si Z est défini

## bic x1 x2 -- x3

Met bits à zéro, identique à **not and**.

## bic! mask addr --

Efface le bit dans l'emplacement d'un mot.

## binary --

Sélectionne la base numérique 2.

## bis! mask addr --

Définit le bit dans l'emplacement d'un mot.

## bit@ mask addr -- flag

Bit de test dans l'emplacement d'un mot.

## bl -- 32

Dépose 32 sur la pile de données.

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

## bne --

Branchement si Z est "clear"

## c! c addr --

Stocke une valeur 8 bits c à l'adresse addr.

## c+! c c-addr --

Ajoute n à l'emplacement mémoire d'un octet.

## c, c --

Ajoute c à la section de données actuelle.

NON VALABLE POUR TOUTES LES CARTES

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

## c@ addr -- c

Récupère la valeur 8 bits c stockée à l'adresse addr.

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

## case --

Marque le début d'une structure `CASE OF ENDOF 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
  ;
```

## cbic! mask c-addr --

Effacer le bit dans l'emplacement de l'octet.

## cbis! mask c-addr --

Définit le bit dans l'emplacement de l'octet.

## cbit@ mask c-addr -- flag

Bit de test dans l'emplacement de l'octet.

## cell+ n -- n'

Incrémente n ...

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

## cells n -- n'

Multiplie n par la taille d'un entier (32 bits = 4 octets).

Permet de se positionner dans un tableau d'entiers.

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

## char -- <string>

Mot utilisable en interprétation seulement.

Empile le premier caractère de la chaîne qui suit ce mot.

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

## clz x1 -- u

Comptage des zéros restant.

## code -- <:name>

Définit un mot dont la définition est écrite en assembleur.

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

## compileonly --

Rend la définition actuelle utilisable en compilation uniquement.

## compiletoflash --

Sélectionne Mecrisp-Stellaris en mode «enregistrer les nouveaux mots en mode flash»

```
compiletoflash

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

: INIT
  closed.loop.program
;

compiletoram
```

## compiletoram --

Sélectionne la mémoire RAM comme cible de compilation.

## compiletoram? -- fl

Empile VRAI si compilation en RAM en cours.

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

Définition d'une constante.

```
binary  \ masks are 32 bit length
00000000000000000000000000000100 constant ledGREEN      \  green LED on G2
00000000010000000000000000000000 constant ledYELLOW     \ yellow LED on G21
00000000000000010000000000000000 constant ledRED        \    red LED on G17
decimal
```

## cos x -- cosx

x est n'importe quel angle s31,32 en degrés.

```
60,0 cos f.   \ display: 0,500000002561137080192565917968 75
```

## cr --

Affiche un retour à la ligne suivante.

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

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

Le mot **CREATE** peut être utilisé seul.

Le mot situé après **CREATE** est créé dans le dictionnaire, ici **DATAS**. L'exécution du mot ainsi créé dépose sur la pile de données l'adresse mémoire de la zone de paramètres. Dans cet exemple, nous avons compilé 4 valeurs 8 bits. Pour les récupérer, il faudra incrémenter l'adresse empilée avec la valeur de décalage de la donnée à récupérer.

```
\ 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 --

Basculer le bit dans l'emplacement de l'octet.

## cycles -- u

Empile le contenu pointé par `TIMERAWL`

## d+ d1 d2 -- d3

Ajoutez d2 à d1, ce qui donne la somme d3.

```
12456. 64251. d+ d.  \ display: 76707
```

## d- d1 d2 -- d3

Soustrayez d2 de d1, ce qui donne la différence d3.

```
45. 27. d- d.   \ display: 18
```

## d. d --

Affiche un nombre double précision.

## d0< d -- flag

flag est vrai si et seulement si d est inférieur à zéro.

```
-23. d0< .  \ display: -1
  0. d0< .  \ display: 0
 46. d0< .  \ display: 0
```

## d0= xd -- flag

flag est vrai si et seulement si xd est égal à zéro.

```
.55 d0= .   \ display:  0
 .0 d0= .   \ display: -1
```

## d2* xd1 -- xd2

xd2 est le résultat du décalage de xd1 d'un bit vers le bit de poids fort, en remplissant le bit de poids faible libéré avec zéro.

```
25. d2* d.   \ display:  50
```

## d2/ xd1 -- xd2

xd2 est le résultat du décalage de xd1 d'un bit vers le bit de poids faible, laissant le bit de poids fort inchangé.

```
47522. d2/ d.   \ display:  23761
```

## d< d1 d2 -- flag

flag is true if and only if d1 is less than d2.

## decimal --

Sélectionne la base numérique décimale. C'est la base numérique par défaut au démarrage de FORTH.

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

## deg-90to90 df1 -- df2

Convertit un angle s31.32 df1 en degrés en un angle df2 en [-90, 90) tel que df1 = df2 + n*180 où n est un nombre entier. (Pour tan uniquement.)

## deg2rad deg -- rad

Convertit s31,32 de degrés s31,32 en radians

```
45,0 deg2rad f.  \ display:  0,78539816779084503650665283203125
```

## depth -- n

n est le nombre de valeurs de cellule unique contenues dans la pile de données avant que n ne soit placé sur la pile.

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

## dictionarynext a-addr -- a-addr flag

Analyse la chaîne de dictionnaire et renvoie vrai si la fin est atteinte.

## dictionarystart -- addr

Point d'entrée courant pour une recherche dans le dictionnaire.

## digit char -- u t|f

Convertit un caractère en chiffre.

## dint --

Désactive les interruptions

## do n1 n2 --

Configure les paramètres de contrôle de boucle avec l'index n2 et la limite n1.

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

## does> comp: -- | exec: -- addr

Le mot **CREATE** peut être utilisé dans un nouveau mot de création de mots...

Associé à **DOES>**, on peut définir des mots qui disent comment un mot est créé puis exécuté.

## drop n --

Enlève du sommet de la pile de données le nombre entier simple précision qui s'y trouvait.

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

## dump a n --

Visualise une zone mémoire.

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

## dump-file-delete addr len addr2 len2 --

Transfère le contenu d'une chaîne texte addr len vers le fichier pointé par addr2 len2

## dup n -- n n

Duplique le nombre entier simple précision situé au sommet de la pile de données.

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

## eint --

Active les interruptions

## else --

Mot d'exécution immédiate et utilisé en compilation seulement. Marque une alternative dans une structure de contrôle du type **IF ... ELSE ... 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 --

Si x est un caractère graphique dans le jeu de caractères défini par l'implémentation, affiche x.

L'effet d'**EMIT** pour toutes les autres valeurs de x est défini par l'implémentation.

Lors du passage d'un caractère dont les bits de définition de caractère ont une valeur comprise entre hex 20 et 7F inclus, le caractère standard correspondant s'affiche. Étant donné que différents périphériques de sortie peuvent répondre différemment aux caractères de contrôle, les programmes qui utilisent des caractères de contrôle pour exécuter des fonctions spécifiques ont une dépendance environnementale. Chaque **EMIT** ne traite qu'avec un seul caractère.

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

## emit? -- fl

Prêt pour envoi d'un caractère.

## endcase --

Marque la fin d'une structure **CASE OF ENDOF 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
  ;
```

## endof --

Marque la fin d'un choix `OF .. ENDOF` dans la structure de contrôle entre `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 --

Efface tout. Efface Ram. Redémarre Forth.

## evaluate addr len --

Évalue le contenu d'une chaine de caractères.

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

## even u1|n1 -- u2|n2

Rend pair. Ajoute 1 si aimpair.

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

## execute addr --

Exécute le mot pointé par addr.

Prenez l'adresse d'exécution de la pile de données et exécute ce jeton. Ce mot puissant vous permet d'exécuter n'importe quel jeton qui ne fait pas partie d'une liste de jetons.

## exit --

Interrompt l'exécution d'un mot et rend la main au mot appelant.

Utilisation typique: `: X ... test IF ... EXIT THEN ... ;`

En exécution, le mot `EXIT` aura le même effet que le mot `;`

## extract n base -- n c

Extrait le digit de poids faible de n. Laisse sur la pile le quotient de n/base et le caractère ASCII de ce digit.

## false -- 0

Laisse 0 sur la pile.

## fill addr len c --

Si len est supérieur à zéro, range c dans toute la zone de longueur len à l'adresse mémoire commençant à addr.

## find addr len -- xt | 0

cherche un mot dans le dictionnaire.

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

## flashvar-here -- a-addr

Donne le pointeur de gestion actuelle de la RAM.

## floor r1 -- r2

Arrondi un réel à la valeur entière inférieure.

```
45,67 floor f.   \ display 45,000000000000000000000000000000
```

## for n --

Marque le début d'une boucle `for .. next`

ATTENTION: l'index de boucle sera traité dans l'intervalle [n..0], soit n+1 itérations, ce qui est contraire aux autres versions du langage FORTH implémentant FOR..NEXT (FlashForth).

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

```
    next
  ;
```

## h!  char c-addr --

stocke un demi mot en mémoire.

## h+!  u|n h-addr --

Ajoute n à l'emplacement mémoire d'un demi mot.

## h@  c-addr - - char

Empile un demi mot depuis la mémoire.

## hbic!  mask h-addr --

Effacer le bit dans l'emplacement de demi mot.

## hbis!  mask h-addr --

Définit le bit dans l'emplacement d'un demi mot.

## hbit@  mask h-addr -- flag

Bit de test dans l'emplacement d'un demi mot.

## here  -- addr

Restitue l'adresse courante du pointeur de dictionnaire.

Le pointeur de dictionnaire s'incrémente au fur et à mesure de la compilation de mots et définition des variables et tableaux de données.

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

## hex  --

Sélectionne la base numérique hexadécimale.

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

## hex.  u --

Imprime un bit entier non signé en base hexadécimale, doit uniquement être émis. Ceci est indépendant de la base numérique.

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

## hld -- addr

Pointeur vers le tampon de texte pour la sortie numérique.

## hold c --

Insère le code ASCII d'un caractère ASCII dans la chaîne de caractères initiée par <#.

## hxor! mask h-addr --

Basculer le bit dans l'emplacement de demi mot.

## i -- n

n est une copie de l'index de boucle actuel.

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

## idle task --

Désactive une tâche aléatoire (sauvegarde IRQ).

## if fl --

Le mot **IF** est d'exécution immédiate.

**IF** marque le début d'une structure de contrôle de type **IF..THEN** ou **IF..ELSE..THEN**.

Lors de l'exécution, la partie de définition située entre **IF** et **THEN** ou entre **IF** et **ELSE** est exécutée si le flag booléen situé au sommet de la pile de données est vrai (f<>0).

Dans le cas contraire, si le flag booléen est faux (f=0), c'est la partie de définition située entre **ELSE** et **THEN** qui sera exécutée. S'il n'y a pas de **ELSE**, l'exécution se poursuit après **THEN**.

```
: WEATHER? ( fl ---)
    IF
        ." Nice weather "
```

```
    ELSE
        ." Bad weather "
    THEN ;
1 WEATHER?    \ display: Nice weather
0 WEATHER?    \ display: Bad weather
```

## immediate --

Rend la définition la plus récente comme mot immédiat.

Définit le bit de lexique de compilation uniquement dans le champ de nom du nouveau mot compilé. Lorsque l'interpréteur rencontre un mot avec ce bit défini, il ne l'exécutera pas, mais transmet un message d'erreur. Ce bit empêche l'exécution des mots de structure en dehors d'une définition de mot.

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

## inline --

Rend la définition actuelle inlinable.

Pour l'espace mémoire flash, placez-le dans votre définition !

## ipsr -- ipsr

Registre d'état du programme d'interruption

## is --

Assigns the execution code of a word to a vectorized execution word.

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

## j -- n

n est une copie de l'index de boucle externe suivant.

```
: 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 est la copie en 3ème niveau dans une boucle do do..loop.

```
: 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

Attend l'appui sur une touche. L'appui sur une touche renvoie son code ASCII.

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

## key? -- fl

Renvoie *vrai* si une touche est appuyée.

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

## list --

Affiche tous les mots du dictionnaire FORTH.

```
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 --

Compile la valeur x comme valeur litérale.

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

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

## log10 fn -- log(fn)

Calcule le logarithme base 10 de fn.

```
100, log10 f.   \ display: 2,00000000000000000000000000000000
 10, log10 f.   \ display: 1,00000000000000000000000000000000
  2, log10 f.   \ display: 0,30102999554947018623352050781250
```

## loop --

Ajoute un à l'index de la boucle. Si l'index de boucle est alors égal à la limite de boucle, supprime les paramètres de boucle et poursuit l'exécution immédiatement après la boucle. Sinon, continue l'exécution au début de la boucle.

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

## lshift x1 n -- x2

Décalage de x1 vers la gauche de n bits.

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

## max n1 n2 -- n1|n2

Laisse le plus grand non signé de u1 et u2.

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

## min n1 n2 -- n1|n2

Laisse min de n1 et n2

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

## mod n1 n2 -- n3

Divise n1 par n2, laisse le reste simple précision n3.

La fonction modulo peut servir à déterminer la divisibilité d'un nombre par un autre.

```
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 --

Si u est supérieur à zéro, copier u caractères consécutifs de l'espace de données commençant à c-addr1 vers celui commençant à c-addr2, en procédant caractère par caractère des adresses inférieures aux adresses supérieures.

## ms n --

Attente en millisencondes.

Pour les attentes longues, définir un mot d'attente en secondes.

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

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

## n. n --

Affiche toute valeur n sous sa forme décimale.

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

## negate n -- -n'

Le complément à deux de n.

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

## new --

Efface la copie RAM actuelle du "dictionnaire flash" et redémarre Forth.

## nip n1 n2 -- n2

Enlève n1 de la pile.

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

## nl -- 10

Dépose 10 sur la pile de données.

`nl` est le code utilisé par eForth comme fin de ligne dans le code source Forth.

```
nl .   \ display 10
```

## nop --

Aucune action.

Ancrage pour manipulateurs inutilisés !

## normal --

Désactive les couleurs sélectionnées pour l'affichage.

## not x1 -- x2

Inverse tous les bits.

## of n --

Marque un choix **OF .. ENDOF** dans la structure de contrôle entre **CASE ENDCASE**

Si la valeur testée est égale à celle qui précède **OF**, la partie de code située entre **OF ENDOF** sera exécutée.

```
: 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

Effectue un OU logique.

Les mots **AND**, **OR** et **XOR** effectuent des opérations logiques binaires **bit à bit** sur les entiers simple précision situés au sommet de la pile de données.

```
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 une copie de n1 au sommet de la pile.

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

## parse c "string" -- addr count

Analyse le mot suivant dans le flux d'entrée, se terminant au caractère c. Laissez l'adresse et le nombre de caractères du mot. Si la zone d'analyse était vide, alors count=0.

## pause --

Passe la main aux autres tâches.

## pi/2 -- pi/2

Empile la valeur de pi/2, ce qui correspond à un angle de 90° exprimé en radians.

```
pi/2 f.   \ display: 1,57079632673412561416625976562500
```

## pi/4 -- pi/4

Empile la valeur de pi/4, ce qui correspond à un angle de 90° exprimé en radians.

```
pi/4 f.   \ display: 0,78539816336706280708312988281250
```

## pow10 fn -- 10exp-fn

Elève 10 à la puissance fn.

```
3, pow10 f.  \ display: 1000,00000000000000000000000000000000
2,4 pow10 f.  \ display: 251,18864816427230834960937500000000
```

## pow2 real1 -- real2

Éleve un nombre réel au carré.

```
4,0 pow2 f.
\ display: 16,00000000000000000000000000000000
5,1 pow2 f.
\ display: 34,29675079137086868286132812500000
```

## prompt --

Affiche un texte de disponibilité de l'interpréteur. Affiche par défaut:

ok

```
prompt   \ display: ok
```

## query --

Récupère l'entrée de l'utilisateur dans le tampon d'entrée.

## r> R: n -- S: n

Transfère n depuis la pile de retour.

Cette opération doit toujours être équilibrée avec >r

```
\ display n in binary format
```

```
: b. ( n -- )
    base @ >r
    binary .
    r> base !
  ;
```

## rad2deg rad -- deg

Convertit s31,32 de radians en s31,32 degrés.

```
pi/4 2,0 f/ rad2deg
f.   \ display: 22,4999999990686774253845214843750 0
```

## rdepth -- n

Donne le nombre d'éléments de pile de retour.

## rdrop --

Supprime la valeur qui est au sommet de la pile de retour.

## recurse --

Ajoute un lien d'exécution correspondant à la définition actuelle.

L'exemple habituel est le codage de la fonction factorielle.

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

## registerlist. --

Affiche la liste des registres.

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

## registerparser Stringadresse Länge -- Nummer

Mot utilisé par certains mots d'assemblage. Analyse si un registre r2..r15 est invoqué.

## repeat --

Achève une boucle indéfinie `begin.. while.. repeat`

## reset --

Réinitialisation au niveau matériel.

## rm -- "path"

Efface le fichier indiqué.

## rol x1 -- x2

Rotation logique de 1 bit vers la gauche.

## ror x1 -- x2

Rotation logique de 1 bit vers la droite.

## rot n1 n2 n3 -- n2 n3 n1

Rotation des trois valeurs au sommet de la pile.

## rp! addr --

Stocke le pointeur de pile de retour.

## rp@ -- addr

Récupère le pointeur de pile de retour.

## RSHIFT x1 u -- x2

Décalage vers la droite de u bits de la valeur x1.

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

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

En interprétation, laisse sur la pile de données la chaine délimitée par "

En compilation, compile la chaine délimitée par "

Lors de l'exécution du mot compilé, restitue l'adresse et la longueur de la chaîne...

```
\ 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
```

## s>d n -- d

Convertit un entier simple en double précision.

```
56 s>d d.   \ display: 56
```

### save ???

???

### SCR-delete -- addr

Variable pointant sur le bloc en cours d'édition.

### see -- name>

Décompile ou désassemble une définition FORTH.

### setsource c-addr len --

Change la source.

### shl x1 -- x2

Décalage logique de 1 bit vers la gauche.

### shr x1 -- x2

Décalage logique de 1 bit vers la droite.

### smudge --

Rend la définition actuelle visible, brûle les drapeaux collectés pour les faire clignoter et s'occupe de la fin appropriée.

### source -- c-addr len

Source courante.

### sp! addr --

Stocke le pointeur de pile de données.

### sp0 -- addr

pointe vers le bas de la pile de données de Forth (pile de données).

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

### sp@ -- addr

Dépose l'adresse du pointeur de pile sur la pile.

### space --

Affiche un caractère espace.

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

### spaces n --

Affiche n fois le caractère espace.

Défini depuis la version 7.071

### sqft fn -- sqrt(fn)

Calcule la racine carrée du nombre réel fn.

```
2, sqrt f.   \ display: 1,414213561918586492538452148437500
```

### stackspace -- 512

Constante. valeur 512, soit 128 éléments pour chaque tâche.

### state -- fl

Etat de compilation. L'état ne peut être modifié que par [ et ].

-1 pour compilateur, 0 pour interpréteur

### stop --

Arrête la tâche courante.

### swap n1 n2 -- n2 n1

Echange les valeurs situées au sommet de la pile.

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

### tan x -- tanx

x est n'importe quel angle s31,32 en degrés.

Déplacer x à la valeur équivalente dans [-90, 90)

```
45,0 tan f.   \ display: 1,000000000232830643653869628906250
```

### task: -- <name>

Crée une nouvelle tâche.

```
task: blinker

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

### tasks --

Affiche les tâches actuellement dans la liste circulaire.

### then --

Mot d'exécution immédiate utilisé en compilation seulement. Marque la fin d'une structure de contrôle de type `IF..THEN` ou `IF..ELSE..THEN`.

### tib -- addr

renvoie l'adresse du tampon d'entrée du terminal où la chaîne de texte d'entrée est conservée.

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

### TIMEHR -- $40054008

constante. Valeur $40054008.

### TIMEHW -- $40054000

constante. Valeur $40054000.

### TIMELR -- $4005400C

constante. Valeur $4005400C.

### TIMELW -- $40054004

constante. Valeur $40054004.

### TIMERAWH -- $40054024

constante. Valeur $40054024.

## TIMERAWL -- $40054028

constante. Valeur $40054028.

Les deux registres `TIMERAWH` et `TIMERAWH` pointent sur un contenu 64 bits du timer.

## true -- -1

Laisse -1 sur la pile.

## tuck x1 x2 -- x2 x1 x2

Copiez le premier élément (du haut) de la pile sous le deuxième élément de la pile.

## type addr c --

Affiche la chaine de caractères sur c octets.

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

## u. n --

Dépile la valeur au sommet de la pile et l'affiche en tant qu'entier simple précision non signé.

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

## u.2 u --

Affiche un nombre entier non signé sur deux caractères.

```
2 u.2   \ display 02
15 u.2   \ display 15
```

## u.4 u --

Affiche un nombre entier non signé sur quatre caractères.

```
2 u.4   \ display 0002
 15 u.4   \ display 0015
355 u.4   \ display 0355
```

## u.8 u --

Affiche un nombre entier non signé sur huit caractères.

```
2 u.8     \ display 00000002
 15 u.8     \ display 00000015
355 u.8     \ display 00000355
47521 u.8   \ display 00047521
```

## U/MOD  u1 u2 -- rem quot

division int/int->int non signée.

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

## u<  u1 u2 -- flag

flag est vrai si et seulement si u1 est inférieur à u2.

## u>  u1 u2 -- flag

flag est vrai si et seulement si u1 est supérieur à u2.

## ud*  ud1 ud2 -- ud3

64*64 = 64 Multiplication

```
12345. 54321. ud* d.  \ display: 670592745
```

## ud.  ud --

Affiche un nombre double précision non signé.

## um*  u1 u2 -- ud

multiplication de deux entiers non signés. Résultat en double précision.

```
3 5 um* d.   \ display: 15
```

## um/mod  ud u1 -- u2 u3

Divisez ud par u1, ce qui donne le quotient u3 et le reste u2. Toutes les valeurs et arithmétiques ne sont pas signées.

## unloop  --

Arrête une action do..loop. Utiliser `unloop` avant `exit` seulement dans une structure do..loop.

```
: example ( -- )
    100 0 do
        cr i .
        key bl = if
```

```
            unloop exit
        then
    loop
  ;
```

## until fl --

Ferme une structure **begin.. until**.

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

## unused -- free-mem

Affiche la mémoire en fonction du mode de compilation (Ram ou Flash).

## us u --

Génère une temporisation de n micro-secondes.

## use -- <name>

Utilise "name" comme fichier de blocs.

```
USE /spiffs/foo
```

## used -- n

Indique l'espace pris par les définitions utilisateur. Ceci inclue les mots déjà définis du dictionnaire FORTH.

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

Mot de création. Définit une variable simple précision.

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

## vorneabschneiden addr len -- addr' len'

Enlève le premier caractère.

## wake task --

Réveille une tâche aléatoire (sauvegarde IRQ)

## wfi --

Compilation seulement. Attend une interruption, entre en mode sommeil.

## while fl --

Marque la partie d'exécution conditionnelle d'une structure `begin..while..repeat`

## words --

Répertorie les noms de définition dans la première liste de mots de l'ordre de recherche. Le format de l'affichage dépend de l'implémentation.

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

## xor n1 n2 -- n3

Effectue un OU eXclusif logique.

Les mots `AND`, `OR` et `XOR` effectuent des opérations logiques binaires **bit à bit** sur les entiers simple précision situés au sommet de la pile de données.

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

## xor! mask addr --

Basculer le bit dans l'emplacement d'un mot.

## [ --

Entre en mode interprétation. `[` est un mot d'exécution immédiate.

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

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

Utilisable en compilation seulement. Exécution immédiate.

Compile le cfa de <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

En compilation, enregistre le code ASCII du caractère indiqué après ce mot.

En exécution, le code xchar est déposé sur la pile de données.

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

## ] --

Retour en mode compilation. `]` est un mot immédiat.

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