

Le manuel pour Z79FORTH

version 1.2 - mardi 6 août 2024



Auteur

- François LAAGEL
- Marc PETREMANN

Collaborateur

-

Contents

Auteur.....	1
Collaborateur.....	1
Introduction.....	4
Objet.....	4
Déclinaisons.....	5
La carte Z79Forth.....	6
Alimentation de la carte Z79Forth.....	8
Le câble USB.....	8
Communiquer avec la carte Z79Forth.....	9
Utilisation d'un terminal.....	9
Utiliser les nombres avec Z79Forth.....	11
Les nombres avec l'interpréteur FORTH.....	11
Saisie des nombres avec différentes base numérique.....	11
Changement de base numérique.....	12
Binaire et hexadécimal.....	13
Taille des nombres sur la pile de données FORTH.....	15
Accès mémoire et opérations logiques.....	16
Un vrai FORTH 16 bits avec Z79Forth.....	18
Les valeurs sur la pile de données.....	18
Les valeurs en mémoire.....	18
Traitement par mots selon taille ou type des données.....	19
Conclusion.....	20
Affichage des nombres et chaînes de caractères.....	22
Changement de base numérique.....	22
Préfixer les nombres entiers.....	23
Définition de nouveaux formats d'affichage.....	23
Affichage des caractères et chaînes de caractères.....	26
Commentaires et mise au point.....	28
Ecrire un code FORTH lisible.....	28
Indentation du code source.....	29
Les commentaires.....	30
Les commentaires de pile.....	30
Signification des paramètres de pile en commentaires.....	30
Commentaires des mots de définition de mots.....	31
Les commentaires textuels.....	32
Commentaire en début de code source.....	32
Moniteur de pile.....	33
Dictionnaire / Pile / Variables / Constantes.....	35
Étendre le dictionnaire.....	35
Piles et notation polonaise inversée.....	35
Manipulation de la pile de paramètres.....	36

La pile de retour et ses utilisations.....	37
Utilisation de la mémoire.....	38
Variables.....	38
Constantes.....	38
Valeurs pseudo-constantes.....	39
Outils de base pour l'allocation de mémoire.....	39
Les mots de création de mots.....	40
Utilisation de does>.....	40
Exemple de gestion de couleur.....	41
Exemple, écrire en pinyin.....	42
Ressources.....	44
GitHub.....	44
Youtube.....	44

Introduction

Il fut un temps, sur cette planète, où l'on pouvait comprendre le fonctionnement interne d'un ordinateur presque jusqu'au niveau du bit. Le matériel était fourni avec des schémas et le code source du logiciel était livré. Ces jours sont-ils derrière nous et à jamais perdus dans la mémoire des *anciens* ?

Z79Forth est un effort pour prouver que ce n'est pas le cas et que le principe KISS (Keep it Simple Stupid) peut encore produire un ordinateur parfaitement compréhensible. Il est proposé aux personnes qui n'ont pas peur de l'assemblage électronique et qui souhaitent apprendre le langage de programmation **Forth**.

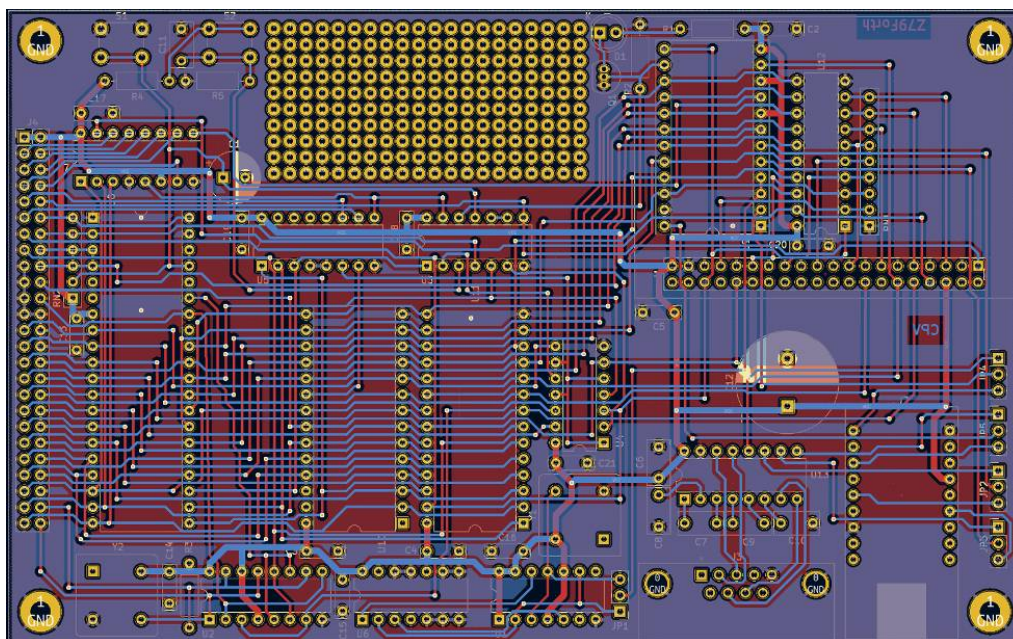
Objet

Le livrable de ce projet est un kit électronique à assembler par soi-même. Il s'agit d'un ordinateur mono-carte très simple et qui laisse ouverte la possibilité de développements matériels ultérieurs. Ses principales caractéristiques sont :

- processeur Hitachi HD63C09E (compatible avec le Motorola MC6809) à 4 megahertz.
- micrologiciel Forth resident en EEPROM de 8 kilo octets.
- 32 kilo octets de RAM statique dont 25 disponibles pour les applications.
- 64 mega octets de stockage de masse sur medium CompatFlash.
- communication par voie série sur USB ou RS232 (115200 ou 38400 bits par seconde).
- un connecteur d'extension pour l'accès à des fonctionnalités supplémentaires.
- alimentation par chargeur de téléphone mobile fourni (prise USB mini B).
- très basse consommation : inférieure à 1 watt dans la plupart des cas.

Certains des composants utilisés sont des pièces de collection disponibles uniquement sur Ebay. La qualité de l'ensemble proposé ici est une préoccupation majeure. Le micrologiciel intégré dans cette plateforme est Open Source ; il est le résultat de quatre années de développement.

Une fois assemblé, l'heureux propriétaire de cette plateforme pourra légitimement se considérer comme détenteur d'un ordinateur uniquement disponible en **édition limitée**.



Déclinaisons

Selon votre propension à la nostalgie, vous pouvez sélectionner l'une des implémentations conformes aux normes suivantes du langage **Forth** :

- la spécification ANS94 qui permet de porter facilement du code contemporain.
- la spécification 79-STANDARD, considérée par certains comme n'ayant qu'une valeur historique mais les véritables nostalgiques auront une autre opinion.

La carte Z79Forth

Comme si on était à nouveau en 1979 !

Cette plate-forme est conçue comme une base pour l'auto-apprentissage et le développement ultérieur du matériel. La variante Forth ciblée est le 79-STANDARD, une référence historique. L'ensemble de la conception est basé sur le Hitachi HD63C09E, une implémentation grandement améliorée du Motorola MC6809.

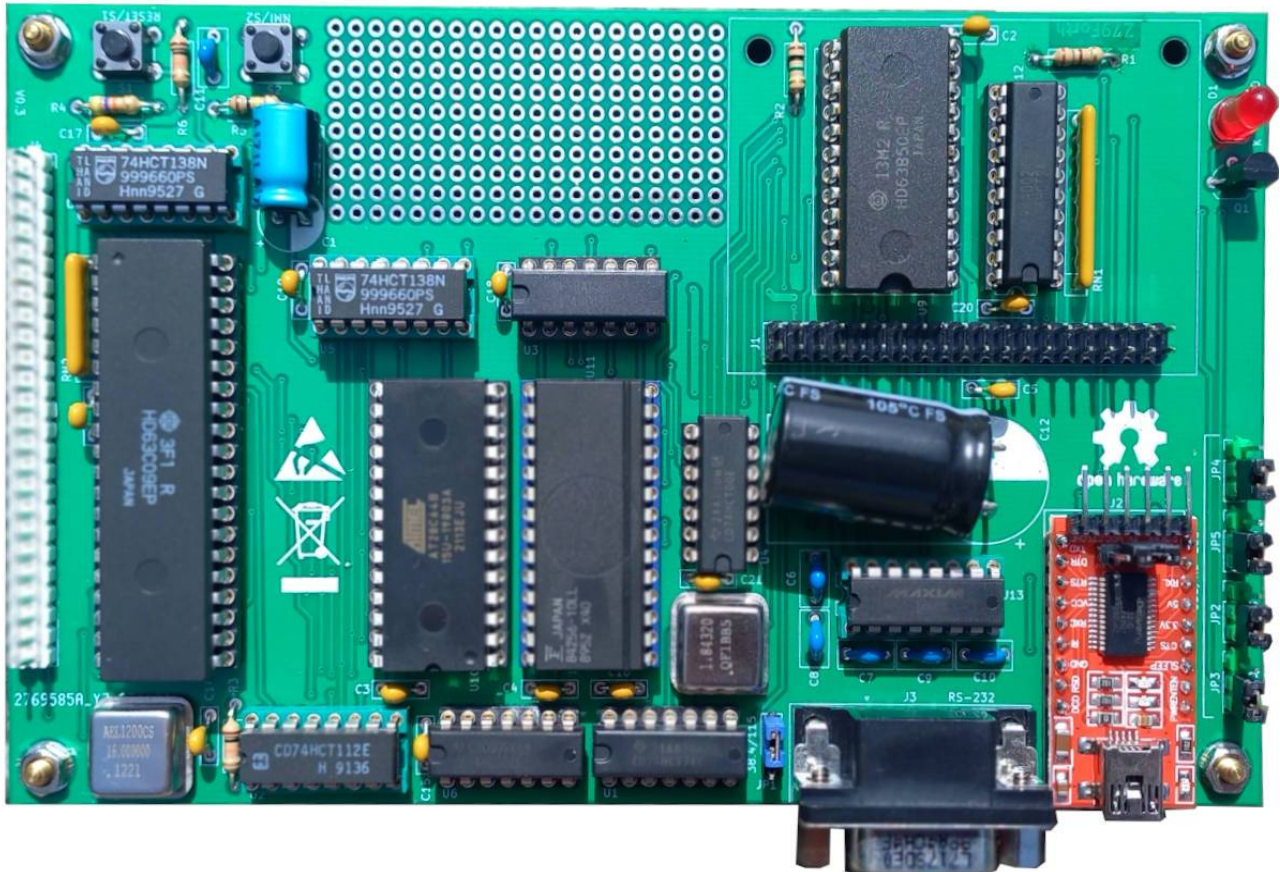


Figure 1: La carte Z79Forth

Les principales caractéristiques sont :

- Fonctionnement du processeur à 5 MHz.
- RAM statique de 32 Ko. Peut être étendue à 48 Ko.
- EEPROM de 8 Ko exécutant une implémentation native du sous-ensemble Forth 79-STANDARD.
- 6 lignes de périphériques d'E/S de rechange sont décodées et disponibles pour des développements ultérieurs.

- Alimentation par USB. La consommation de courant est comprise entre 56 et 150 mA.
- Console de ligne série fonctionnant à 115 200 bps.
- Prise en charge du stockage de masse sur SanDisk CompactFlash (jusqu'à 64 Mo).
- Conception sans interruption.

Le logiciel est sous licence GNU General Public License version 3 et est disponible à l'adresse <https://github.com/frenchie68/Z79Forth>

Des schémas Kicad y sont également fournis.

État du projet : carte fonctionnelle sur circuit imprimé. Le logiciel fonctionne conformément aux spécifications. Il n'y a aucun bug connu à ce jour.

Alimentation de la carte Z79Forth

La carte Z79Forth peut être alimentée de deux manières :

- avec un bloc d'alimentation 230V-5V et un câble USB-A 2.0 vers mâle Mini USB ;
- depuis un ordinateur via le câble USB-A 2.0 vers mâle Mini USB.

Le câble USB

Dans les deux cas, il faut utiliser un câble USB-A 2.0 vers mâle Mini USB :



Figure 2: câble USB 2.0 et alimentation 230V->5V

Vous pouvez alimenter la carte Z79Forth depuis le port série d'un ordinateur. Dans ce cas, seul le câble USB est nécessaire. Cette solution est à utiliser seulement si vous avez peu de périphériques USB connectés simultanément sur l'ordinateur.

Si vous êtes amené à tester des composants sur la carte Z79Forth, utilisez un hub USB. En cas d'incident, cette solution évitera d'endommager le port USB de l'ordinateur.

Si vous voulez communiquer avec la carte Z79Forth et l'alimenter en même temps via le port USB, il faudra régler les connecteurs **JP1** à **JP5** sur la carte. Voir chapitre *Communiquer avec la carte Z79Forth*.

Communiquer avec la carte Z79Forth

Il y a deux moyens de communiquer avec la carte Z79Forth :

- par liaison série ↔ USB, via un câble USB-A 2.0 vers mâle Mini USB, voir *Alimentation de la carte Z79Forth*.
- par câble USB ↔ RS232 DB9.

Si vous utilisez le câble USB-A 2.0 vers mâle Mini USB, vérifiez la position des cavaliers JP2 à JP5 :

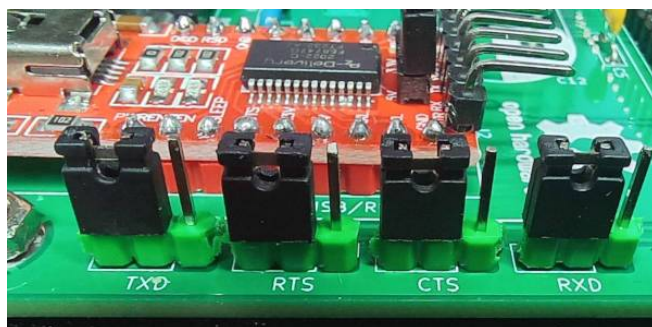


Figure 3: position des cavaliers JP2 à JP5

Pour la liaison via un câble USB-A 2.0 mâle Mini USB, les cavaliers doivent être dans la position montrée sur la photo ci-dessus.

Le cavalier JP1 se trouve à gauche du connecteur RS232 DB9. Ce cavalier règle la vitesse de transmission série :

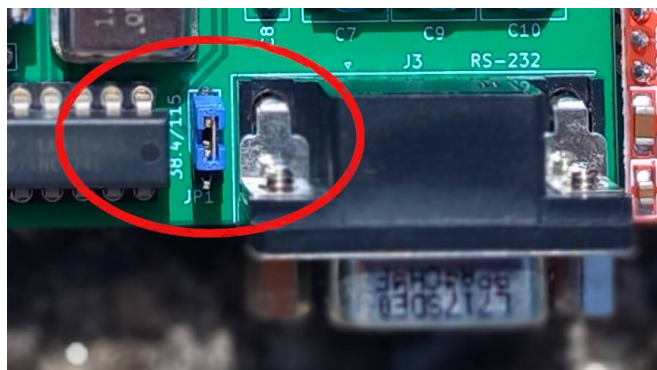


Figure 4: position du cavalier JP1

Dans la position 2-3 du cavalier JP1, telle que visible sur la photo ci-dessus, la liaison série est établie à 115200 bps. Si vous utilisez un terminal ne communiquant pas à cette vitesse, mettez ce cavalier dans la position 1-2. Dans la position JP1 1-2, la vitesse de transmission série sera ramenée à 38400 bps.

Utilisation d'un terminal

Il existe de nombreux logiciels permettant d'émuler un terminal . Citons les plus connus :

- **Putty** sous Windows ou Linux ;
- **Tera Term** sous Windows ;
- **XXXXX**... @TODO à compléter

Le terminal permet de communiquer avec l'interpréteur FORTH implanté sur la carte Z79Forth.

Utiliser les nombres avec Z79Forth

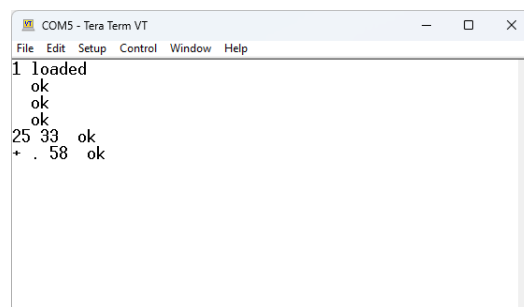
Nous avons démarré Z79Forth sans souci. Nous allons maintenant approfondir quelques manipulations sur les nombres pour comprendre comment maîtriser 79Forth.

Nous allons commencer par aborder ces notions élémentaires en vous invitant à effectuer des manipulations simples.

Les nombres avec l'interpréteur FORTH

Au démarrage de Z79Forth, la fenêtre du terminal TERA TERM (ou tout autre programme de terminal de votre choix) doit indiquer la disponibilité de Z79Forth. Appuyez une ou deux fois sur la touche *ENTER* du clavier. Z79Forth répond avec la confirmation de bonne exécution **ok..**

On va tester l'entrée de deux nombres, ici **25** et **33**. Tapez ces nombres, puis *ENTER* au clavier. Z79Forth répond toujours par **ok..** Vous venez d'empiler deux nombres sur la pile du langage FORTH. Entrez maintenant **+** puis sur la touche *ENTER*. Z79Forth affiche le résultat :



```
COM5 - Tera Term VT
File Edit Setup Control Window Help
1 loaded
ok
ok
ok
25 33 ok
+ . 58 ok
```

Figure 5: première opération avec Z79Forth

Cette opération a été traitée par l'interpréteur FORTH.

Z79Forth, comme toutes les versions du langage FORTH a deux états :

- **interpréteur** : l'état que vous venez de tester en effectuant une simple somme de deux nombres ;
- **compilateur** : un état qui permet de définir de nouveaux mots. Cet aspect sera approfondi ultérieurement.

Saisie des nombres avec différentes base numérique

Afin de bien assimiler les explications, vous êtes invité à tester tous les exemples via la fenêtre du terminal TERA TERM ou tout autre programme terminal de votre choix.

Les nombres peuvent être saisis de manière naturelle. En décimal, ce sera TOUJOURS une séquence de chiffres, exemple :

```
-1234 5678 + .
```

Le résultat de cet exemple affichera **4444**. Les nombres et mots FORTH doivent être séparés par au moins un caractère *espace*. L'exemple fonctionne parfaitement si on tape un nombre ou mot par ligne :

```
-1234
5678
+
.
```

Les nombres peuvent être préfixés si on souhaite saisir des valeurs autrement que sous leur forme décimale :

- le signe **\$** pour indiquer que le nombre est une valeur hexadécimale ;
- le signe **%** pour indiquer que le nombre est une valeur binaire;

Exemple :

```
255 .      \ display 255
$ff .      \ display 255
```

L'intérêt de ces préfixes est d'éviter toute erreur d'interprétation en cas de valeurs similaires :

```
$0305
0305
```

ne sont **pas** des nombres **égaux** si la base numérique hexadécimale n'est pas explicitement définie !

Changement de base numérique

Z79Forth dispose de mots permettant de changer de base numérique :

- **hex** pour sélectionner la base numérique hexadécimale ;
- **decimal** pour sélectionner la base numérique décimale.

Tout nombre saisi dans une base numérique doit respecter la syntaxe des nombres dans cette base :

```
3E7F
```

provoquera une erreur si vous êtes en base décimale.

```
hex 3e7f
```

fonctionnera parfaitement en base hexadécimale. La nouvelle base numérique reste valable tant qu'on ne sélectionne pas une autre base numérique :

```
hex
$0305
```

sont des nombres **égaux**!

Une fois un nombre déposé sur la pile de données dans une base numérique, sa valeur ne change plus. Par exemple, si vous déposez la valeur **\$ff** sur la pile de données, cette valeur qui est **255** en décimal, ou **11111111** en binaire, ne changera pas si on revient en décimal :

```
hex ff decimal . \ display: 255
```

Au risque d'insister, **255** en décimal est **la même valeur** que **\$ff** en hexadécimal !

Dans l'exemple donné en début de chapitre, on définit une constante en hexadécimal :

```
25 constant myScore
```

Si on tape :

```
hex myScore .
```

Ceci affichera le contenu de cette constante sous sa forme hexadécimale. Le changement de base n'a **aucune conséquence** sur le fonctionnement final du programme FORTH.

Binaire et hexadécimal

Le système de numération binaire moderne, base du code binaire, a été inventé par Gottfried Leibniz en 1689 et apparaît dans son article Explication de l'Arithmétique Binaire en 1703.

Dans son article, LEIBNITZ se sert des seuls caractères **0** et **1** pour décrire tous les nombres :

```
: bin0to15 ( -- )
  2 base !
  $10 0 do
    cr i .
  loop
  cr decimal ;
bin0to15 \ display:
0
1
10
11
100
101
110
111
1000
1001
1010
1011
```

```
1100
1101
1110
1111
```

Est-ce nécessaire de comprendre le codage binaire ? Je dirai oui et non. **Non** pour les usages de la vie courante. **Oui** pour comprendre la programmation des micro-contrôleurs et la maîtrise des opérateurs logiques.

C'est Georges Boole qui a décrit de manière formelle la logique. Ses travaux ont été oubliés jusqu'à l'apparition des premiers ordinateurs. C'est Claude Shannon qui se rend compte qu'on peut appliquer cet algèbre dans la conception et l'analyse de circuits électriques.

L'algèbre de Boole manipule exclusivement des **0** et des **1**.

Les composants fondamentaux de tous nos ordinateurs et mémoires numériques utilisent le codage binaire et l'algèbre de Boole.

La plus petite unité de stockage est l'octet. C'est un espace constitué de 8 bits. Un bit ne peut avoir que deux états : **0** ou **1**. La valeur la plus petite pouvant être stockée dans un octet est **00000000**, la plus grande étant **11111111**. Si on coupe en deux un octet, on aura :

- quatre bits de poids faible, pouvant prendre les valeurs **0000** à **1111** ;
- quatre bits de poids fort pouvant prendre une de ces mêmes valeurs.

Si on numérote toutes les combinaisons entre 0000 et 1111, en partant de 0, on arrive à 15 :

```
: binary ( -- )
  2 base !
;
: bin0to15 ( -- )
  binary
  $10 0 do
    cr i .
    i hex . binary
  loop
  cr decimal ;
bin0to15 \ display:
0 0
1 1
10 2
11 3
100 4
101 5
110 6
```



```

111 7
1000 8
1001 9
1010 A
1011 B
1100 C
1101 D
1110 E
1111 F

```

Dans la partie droite de chaque ligne, on affiche la même valeur que dans la partie gauche, mais en hexadécimal : **1101** et **D** sont les mêmes valeurs !

La représentation hexadécimale a été choisie pour représenter des nombres en informatique pour des raisons pratiques. Pour la partie de poids fort ou faible d'un octet, sur 4 bits, les seuls combinaisons de représentation hexadécimale seront comprises entre **0** et **F**. Ici, les lettres A à F **sont des chiffres** hexadécimaux !

```
$3E \ is more readable as 00111110
```

La représentation hexadécimale offre donc l'avantage de représenter le contenu d'un octet dans un format fixe, de **00** à **FF**. En décimal, il aurait fallu utiliser 0 à 255.

Taille des nombres sur la pile de données FORTH

Z79Forth utilise une pile de données de 16 bits de taille mémoire, soit 2 octets (8 bits x 2 = 16 bits). La plus petite valeur hexadécimale pouvant être empilée sur la pile FORTH sera **0000**, la plus grande sera **FFFF**. Toute tentative d'empiler une valeur de taille supérieure se solde par une erreur :

```

abcdefabcdefabcdef \ display:
OoR error (0000/0000)
F58F >NUMBER+0081

```

Empilons la plus grande valeur possible au format hexadécimal sur 16 bits (2 octets) :

```

decimal
$ffff . \ display: -1

```

Je vous voit surpris, mais ce résultat est **normal** ! Le mot **.** affiche la valeur qui est au sommet de la pile de données sous sa forme signée. Pour afficher la même valeur non signée, il faut utiliser le mot **u.** :

```
$ffff u. \ display: 65535
```

C'est parce que sur les 16 bits utilisés par FORTH pour représenter un nombre entier, le bit de poids fort est utilisé comme signe :

- si le bit de poids fort est à **0**, le nombre est positif ;
- si le bit de poids fort est à **1**, le nombre est négatif.

Donc, si vous avez bien suivi, nos valeurs décimales 1 et -1 sont représentées sur la pile, au format binaire sous cette forme :

```
2 base !
00000000000000001 \ push 1 on stack
1111111111111111 \ push -1 on stack
```

Et c'est là qu'on va faire appel à notre mathématicien, Mr LEIBNITZ, pour additionner en binaire ces deux nombres. Si on fait comme à l'école, en commençant par la droite, il faudra simplement respecter cette règle : $1 + 1 = 10$ en binaire. On met les résultats sur une troisième ligne :

```
00000000000000001
11111111111111111
      10
```

Etape suivante :

```
00000000000000001
11111111111111111
      10
     100
```

Arrivé à la fin, on aura comme résultat :

```
00000000000000001
11111111111111111
10000000000000000
```

Mais comme ce résultat a un 17ème bit de poids fort à 1, sachant que le format des entiers est strictement limité à 16 bits, le résultat final est 0. C'est surprenant ? C'est pourtant ce que fait toute horloge digitale. Masquez les heures. Arrivé à 59, rajoutez 1, l'horloge affichera 0.

Les règles de l'arithmétique décimale, à savoir $-1 + 1 = 0$ ont été parfaitement respectées en logique binaire !

Accès mémoire et opérations logiques

La pile de données n'est en aucun cas un espace de stockage de données. Sa taille est d'ailleurs très limitée. Et la pile est partagée par beaucoup de mots. L'ordre des paramètres est fondamental. Une erreur peut générer des dysfonctionnements :

```
variable score
```

Stockons une valeur quelconque dans **score** :

```
decimal
1900 score !
```

Récupérons le contenu de **score** :

```
score @ . \ display 1900
```

Pour isoler l'octet de poids faible. Plusieurs solutions s'offrent à nous. Une solution exploite le masquage binaire avec l'opérateur logique **and** :

```
hex
score @ .          \ display:  76C
score @
$00FF and .        \ display:  6C
```

Pour isoler le second octet en partant de la droite :

```
score @
$FF00 and .        \ display:  700
```

Ici, nous nous sommes amusés avec le contenu d'une variable.

Pour conclure ce chapitre, il y a encore beaucoup à apprendre sur la logique binaire et les différents codages numériques possibles. Si vous avez testé les quelques exemples donnés ici, vous comprenez certainement que FORTH est un langage intéressant :

- grâce à son interpréteur qui permet d'effectuer de nombreux tests, ce de manière interactive sans nécessiter de recompilation en transmission de code ;
- un dictionnaire dont la plupart des mots sont accessibles depuis l'interpréteur ;
- un compilateur permettant de rajouter de nouveaux mots *à la volée*, puis les tester immédiatement.

Enfin, ce qui ne gâche rien, le code FORTH, une fois compilé, est certainement aussi performant que son équivalent en langage C.

Un vrai FORTH 16 bits avec Z79Forth

Z79Forth est un vrai FORTH 16 bits. Qu'est-ce que ça signifie ?

Le langage FORTH privilégie la manipulation de valeurs entières. Ces valeurs peuvent être des valeurs littérales, des adresses mémoires, des contenus de registres...

Les valeurs sur la pile de données

Au démarrage de Z79Forth, l'interpréteur FORTH est disponible. Si vous entrez n'importe quel nombre, il sera déposé sur la pile sous sa forme d'entier 16 bits :

```
35
```

Si on empile une autre valeur, elle sera également empilée. La valeur précédente sera repoussée vers le bas d'une position :

```
45
```

Pour faire la somme de ces deux valeurs, on utilise un mot, ici **+** :

```
+
```

Nos deux valeurs entières 16 bits sont additionnées et le résultat est déposé sur la pile. Pour afficher ce résultat, on utilisera le mot **.** :

```
. \ affiche 80
```

En langage FORTH, on peut concentrer toutes ces opérations en une seule ligne :

```
35 45 + . \ affiche 80
```

Contrairement au langage C, on ne définit pas de type **int8** ou **int16** ou **int32**.

Avec Z79Forth, un caractère ASCII sera désigné par un entier 16 bits, mais dont la valeur sera bornée [32..256[. Exemple :

```
decimal  
67 emit \ display C
```

Avec Z79Forth, un entier 16 bits signé sera défini dans l'intervalle -32768 à 32767.

Parfois, on parle de demi-mot. Un demi-mot numérique concerne la partie 8 bits de poids fort ou poids faible d'un entier 16 bits. Un demi-mot sera défini dans l'intervalle 0 à 256.

Les valeurs en mémoire

Z79Forth permet de définir des constantes, des variables. Leur contenu sera toujours au format 16 bits. Mais il est des situations où ça ne nous arrange pas forcément. Prenons un exemple simple, définir un alphabet morse. Nous n'avons besoin que de quelques octets :

- un pour définir le nombre de signes du code morse

- un ou plusieurs octets pour chaque lettre du code morse

```
create mA ( -- addr )
  2 c,
  char . c,   char - c,

create mB ( -- addr )
  4 c,
  char - c,   char . c,   char . c,   char . c,

create mC ( -- addr )
  4 c,
  char - c,   char . c,   char - c,   char . c,
```

Ici, nous définissons seulement 3 mots, **mA**, **mB** et **mC**. Dans chaque mot, on stocke plusieurs octets. La question est: comment va-t-on récupérer les informations dans ces mots?

L'exécution d'un de ces mots dépose une valeur 16 bits, valeur qui correspond à l'adresse mémoire où on a stocké nos informations morse. C'est le mot **c@** qui va nous servir à extraire le code morse de chaque lettre :

```
mA c@ . \ affiche 2
mB c@ . \ affiche 4
```

Le premier octet extrait ainsi va nous servir à gérer une boucle pour afficher le code morse d'une lettre :

```
: .morse ( addr -- )
  dup 1+ swap c@ 0 do
    dup i + c@ emit
  loop
  drop
;

mA .morse \ affiche .-
mB .morse \ affiche -...
mC .morse \ affiche -.-.
```

Il existe plein d'exemples certainement plus élégants. Ici, c'est pour montrer une manière de manipuler des valeurs 8 bits, nos octets, alors qu'on exploite ces octets sur une pile 16 bits.

Traitement par mots selon taille ou type des données

Dans tous les autres langages, on a un mot générique, genre **echo** (en PHP) qui affiche n'importe quel type de donnée. Que ce soit entier, réel, chaîne de caractères, on utilise toujours le même mot. Exemple en langage PHP :

```
$bread = "Pain cuit";
$price = 2.30;
echo $bread . " : " . $price;
```

```
// affiche   Pain cuit: 2.30
```

Pour tous les programmeurs, cette manière de faire est LA NORME! Alors comment ferait FORTH pour cet exemple en PHP?

```
: pain s" Pain cuit" ;  
: prix s" 2.30" ;  
pain type    s" : " type    prix type  
\ affiche   Pain cuit: 2.30
```

Ici, le mot **type** nous indique qu'on vient de traiter une chaîne de caractères.

Là où PHP (ou n'importe quel autre langage) a une fonction générique et un analyseur syntaxique, FORTH compense avec un type de donnée unique, mais des méthodes de traitement adaptées qui nous informent sur la nature des données traitées.

Voici un cas absolument trivial pour FORTH, afficher un nombre de secondes au format HH:MM:SS:

```
: :##  
  # 6 base !  
  # decimal  
  [char] : hold  
;  
: .hms ( n -- )  
  0 <# :## :## # # #> type  
;  
4225 .hms \ affiche: 01:10:25
```

J'adore cet exemple, car, à ce jour, **AUCUN AUTRE LANGAGE DE PROGRAMMATION** n'est capable de réaliser cette conversion HH:MM:SS de manière aussi élégante et concise.

Vous l'avez compris, le secret de FORTH est dans son vocabulaire.

Conclusion

FORTH n'a pas de typage de données. Toutes les données transitent par une pile de données. Chaque position dans la pile Z79Forth est TOUJOURS un entier 16 bits !

C'est tout ce qu'il y a à savoir.

Les puristes de langages hyper structurés et verbeux, tels C ou Java, crieront certainement à l'hérésie. Et là, je me permettrai de leur répondre : pourquoi avez-vous besoin de typer vos données ?

Car, c'est dans cette simplicité que réside la puissance de FORTH: une seule pile de données avec un format non typé et des opérations très simples.

Et je vais vous montrer ce que bien d'autres langages de programmation ne savent pas faire, définir de nouveaux mots de définition :

```
: morse: ( comp: c -- | exec -- )
```



```

create
,
does>
  dup 1 cells + swap @ 0 do
    dup i + c@ emit
  loop
  drop space
;
2 morse: mA      char . c,   char - c,
4 morse: mB      char - c,   char . c,   char . c,   char . c,
4 morse: mC      char - c,   char . c,   char - c,   char . c,
mA mB mC        \ affiche  .- ... -.-.

```

Ici, le mot **morse:** est devenu un mot de définition, au même titre que **constant** ou **variable**...

Car FORTH est plus qu'un langage de programmation. C'est un méta-langage, c'est à dire un langage pour construire **votre propre langage** de programmation....

Affichage des nombres et chaînes de caractères

Changement de base numérique

FORTH ne traite pas n'importe quels nombres. Ceux que vous avez utilisés en essayant les précédents exemples sont des entiers signés simple précision. Le domaine de définition des entiers 16 bits signés est compris entre -32768 à 32767. Exemple :

```
32767 .      \ affiche 32767
32767 1+ .    \ affiche -32768
-1 u.        \ affiche 65535
```

Ces nombres peuvent être traités dans n'importe quelle base numérique, toutes les bases numériques situées entre 2 et 36 étant valides :

```
255 HEX . DECIMAL \ affiche FF
```

On peut choisir une base numérique encore plus grande, mais les symboles disponibles sortiront de l'ensemble alpha-numérique [0..9,A..Z] et risquent de devenir incohérents.

La base numérique courante est contrôlée par une variable nommée **BASE** et dont le contenu peut être modifié. Ainsi, pour passer en binaire, il suffit de stocker la valeur **2** dans **BASE**. Exemple:

```
2 BASE !
```

et de taper **DECIMAL** pour revenir à la base numérique décimale.

Z79Forth dispose de deux mots pré-définis permettant de sélectionner différentes bases numériques :

- **DECIMAL** pour sélectionner la base numérique décimale. C'est la base numérique prise par défaut au démarrage de Z79Forth ;
- **HEX** pour sélectionner la base numérique hexadécimale.

Dès sélection d'une de ces bases numériques, les nombres littéraux seront interprétés, affichés ou traités dans cette base. Tout nombre entré précédemment dans une base numérique différente de la base numérique courante est automatiquement converti dans la base numérique actuelle. Exemple :

```
DECIMAL \ base en decimal
255     \ empile 255
HEX     \ sélectionne base hexadécimale
1+      \ incrémente 255 devient 256
.       \ affiche 100
```

On peut définir sa propre base numérique en définissant le mot approprié ou en stockant cette base dans **BASE**. Exemple :

```
: BINARY ( -- )      \ sélectionne la base numérique binaire
    2 BASE ! ;
DECIMAL 255 BINARY .  \ affiche      11111111
```

Le contenu de **BASE** peut être empilé comme le contenu de n'importe quelle autre variable :

```
VARIABLE RANGE_BASE  \ définition de variable RANGE_BASE
BASE @ RANGE_BASE !   \ stockage contenu BASE dans RANGE_BASE
HEX FF 10 + .         \ affiche 10F
RANGE_BASE @ BASE !   \ restaure BASE avec contenu de RANGE_BASE
```

Dans une définition **:**, le contenu de **BASE** peut transiter par la pile de retour :

```
: OPERATION ( ---)
    BASE @ >R          \ stocke BASE sur pile de retour
    HEX $FF 10 + .     \ opération du précédent exemple
    R> BASE ! ;        \ restaure valeur initiale de BASE
```

ATTENTION: les mots **>R** et **R>** ne sont pas exploitables en mode interprété. Vous ne pouvez utiliser ces mots que dans une définition qui sera compilée.

Préfixer les nombres entiers

Z79Forth autorise la saisie de valeurs numériques dans une des bases suivantes en faisant préfixer ces valeurs ainsi :

- préfixe **\$** pour marquer un entier en base hexadécimale ;
- préfixe **%** pour marquer un entier en base binaire ;
- préfixe **&** ou **#** pour marquer un entier en base décimale ;
- préfixe **@** pour marquer un entier en base octale.

Exemple :

```
hex F7 2 base ! 00001111 and decimal .
```

Peut être réécrit avec les préfixes numériques :

```
$F7 %00001111 and .
```

Ces préfixes offrent l'avantage de ne pas provoquer d'erreur si on sélectionne accidentellement une mauvaise base numérique :

```
#101 .      \ affiche 101
%101 .      \ affiche 5
$101 .      \ affiche 257
```

Définition de nouveaux formats d'affichage

Forth dispose de primitives permettant d'adapter l'affichage d'un nombre à un format quelconque. Avec Z79Forth, ces primitives traitent les nombres entiers :

- **<#** débute une séquence de définition de format ;
- **#** insère un digit dans une séquence de définition de format ;
- **#S** équivaut à une succession de **#** ;
- **HOLD** insère un caractère dans une définition de format ;
- **#>** achève une définition de format et laisse sur la pile l'adresse et la longueur de la chaîne contenant le nombre à afficher.

Ces mots ne sont utilisables qu'au sein d'une définition. Exemple, soit à afficher un nombre exprimant un montant libellé en euros avec la virgule comme séparateur décimal :

```
: .EUROS ( n ---)
s>d
<# # # [char] , hold #S #>
type space ." EUR" ;
1245 .euros
```

Exemples d'exécution :

35 .EUROS	\ affiche	0,35 EUR
3575 .EUROS	\ affiche	35,75 EUR
1015 3575 + .EUROS	\ affiche	45,90 EUR

Dans la définition de **EUROS** :

- le mot **s>d** convertit un entier 16 bits en un entier 32 bits,
- le mot **<#** débute la séquence de définition de format d'affichage ;
- les deux mots **#** placent les chiffres des unités et des dizaines dans la chaîne de caractère ;
- le mot **HOLD** place le caractère **,** (virgule) à la suite des deux chiffres de droite ;
- le mot **#S** complète le format d'affichage avec les chiffres non nuls à la suite de **,**
- le mot **#>** ferme la définition de format et dépose sur la pile l'adresse et la longueur de la chaîne contenant les digits du nombre à afficher.

Le mot **TYPE** affiche cette chaîne de caractères.

En exécution, une séquence de format d'affichage traite exclusivement des nombres entiers 32 bits signés ou non signés. La concaténation des différents éléments de la chaîne se fait de droite à gauche, c'est à dire en commençant par les chiffres les moins significatifs.

Le traitement d'un nombre par une séquence de format d'affichage est exécutée en fonction de la base numérique courante. La base numérique peut être modifiée entre deux digits.

Voici un exemple plus complexe démontrant la compacité du FORTH. Il s'agit d'écrire un programme convertissant un nombre quelconque de secondes au format HH:MM:SS:

```
: :00 ( ---)
  DECIMAL #          \ insertion digit unité en décimal
  6 BASE !           \ sélection base 6
  #                  \ insertion digit dizaine
  [char] : HOLD      \ insertion caractère :
  DECIMAL ;          \ retour base décimale
: HMS ( n ---)       \ affiche nombre secondes format HH:MM:SS
  s>d <# :00 :00 #S #> TYPE SPACE ;
```

Exemples d'exécution:

```
59 HMS      \ affiche      0:00:59
60 HMS      \ affiche      0:01:00
4500 HMS     \ affiche      1:15:00
```

Explication : le système d'affichage des secondes et des minutes est appelé système sexagésimal. Les **unités** sont exprimées dans la base numérique décimale, les **dizaines** sont exprimées dans la base six. Le mot **:00** gère la conversion des unités et des dizaines dans ces deux bases pour la mise au format des chiffres correspondants aux secondes et aux minutes. Pour les heures, les chiffres sont tous décimaux.

Autre exemple, soit à définir un programme convertissant un nombre entier simple précision décimal en binaire et l'affichant au format bbbb bbbb bbbb bbbb:

```
: FOUR-DIGITS ( ---)
  # # # # 32 HOLD ;
: AFB ( d ---)          \ format 4 digits and a space
  BASE @ >R             \ Current database backup
  2 BASE !              \ Binary digital base selection
  s>d
  <#
  4 0 DO                \ Format Loop
    FOUR-DIGITS
  LOOP
  #> TYPE SPACE         \ Binary display
  R> BASE ! ;           \ Initial digital base restoration
```

Exemple d'exécution :

```
#12 AFB      \ affiche      0000 0000 0000 0110
$3FC5 AFB    \ affiche      0011 1111 1100 0101
```

Encore un exemple, soit à créer un agenda téléphonique où l'on associe à un patronyme un ou plusieurs numéros de téléphone. On définit un mot par patronyme :

```
: .## ( -- )
  # # [char] . HOLD ;
: .TEL ( d -- )
  CR <# .## .## .## .## # # #> TYPE CR ;
```

```
: DUGENOU ( -- )
    0618051254. .TEL ;
dugenou \ affiche : 06.18.05.12.54
```

Ce répertoire téléphonique, qui peut être compilé depuis un fichier source, est facilement modifiable, et bien que les noms ne soient pas classés, la recherche y est extrêmement rapide.

Affichage des caractères et chaînes de caractères

L'affichage d'un caractère est réalisé par le mot **EMIT**:

```
65 EMIT \ affiche A
```

Les caractères affichables sont compris dans l'intervalle 32..255. Les codes compris entre 0 et 31 seront également affichés, sous réserve de certains caractères exécutés comme des codes de contrôle. Voici une définition affichant tout le jeu de caractères de la table ASCII :

```
variable #out
: #out+! ( n -- )
    #out +! \ incrémente #out
;
: (.) ( n -- a l )
    DUP ABS s>d <# #S SWAP SIGN #>
;
: .R ( n l -- )
    >R (.) R> OVER - SPACES TYPE
;
: JEU-ASCII ( ---)
    cr 0 #out !
    128 32
    DO
        I 3 .R SPACE \ affiche code du caractère
        4 #out+!
        I EMIT 2 SPACES \ affiche caractère
        3 #out+!
        #out @ 77 =
        IF
            CR 0 #out !
        THEN
    LOOP ;
```

L'exécution de **JEU-ASCII** affiche les codes ASCII et les caractères dont le code est compris entre 32 et 127. Pour afficher la table équivalente avec les codes ASCII en hexadécimal, taper **HEX JEU-ASCII** :

```
hex jeu-ascii
20 21 ! 22 " 23 # 24 $ 25 % 26 & 27 ' 28 ( 29 ) 2A *
2B + 2C , 2D - 2E . 2F / 30 0 31 1 32 2 33 3 34 4 35 5
```


36 6	37 7	38 8	39 9	3A :	3B ;	3C <	3D =	3E >	3F ?	40 @
41 A	42 B	43 C	44 D	45 E	46 F	47 G	48 H	49 I	4A J	4B K
4C L	4D M	4E N	4F O	50 P	51 Q	52 R	53 S	54 T	55 U	56 V
57 W	58 X	59 Y	5A Z	5B [5C \	5D]	5E ^	5F _	60 `	61 a
62 b	63 c	64 d	65 e	66 f	67 g	68 h	69 i	6A j	6B k	6C l
6D m	6E n	6F o	70 p	71 q	72 r	73 s	74 t	75 u	76 v	77 w
78 x	79 y	7A z	7B {	7C	7D }	7E ~	7F	ok		

Les chaînes de caractères sont affichées de diverses manières. La première, utilisable en compilation seulement, affiche une chaîne de caractères délimitée par le caractère " (guillemet) :

```
: TITRE ." MENU GENERAL" ;
      TITRE      \ affiche      MENU GENERAL
```

La chaîne est séparée du mot "." par au moins un caractère espace.

Une chaîne de caractères peut aussi être compilée par le mot **s** et délimitée par le caractère " (guillemet) :

```
: LIGNE1 ( --- adr len)
      S" E..Enregistrement de donnees" ;
```

L'exécution de **LIGNE1** dépose sur la pile de données l'adresse et la longueur de la chaîne compilée dans la définition. L'affichage est réalisé par le mot **TYPE** :

```
LIGNE1 TYPE      \ affiche E..Enregistrement de donnees
```

En fin d'affichage d'une chaîne de caractères, le retour à la ligne doit être provoqué s'il est souhaité :

```
CR TITRE CR CR LIGNE1 TYPE CR
\ affiche
\ MENU GENERAL
\
\ E..Enregistrement de données
```

Un ou plusieurs espaces peuvent être ajoutés en début ou fin d'affichage d'une chaîne alphanumérique :

```
SPACE          \ affiche un caractère espace
10 SPACES      \ affiche 10 caractères espace
```

Commentaires et mise au point

Il n'existe pas d'IDE¹ pour gérer et présenter le code écrit en langage FORTH de manière structurée. Au pire, vous utilisez un éditeur de texte ASCII, au mieux un vrai IDE et des fichiers texte :

- **edit** ou **wordpad** sous Windows
- **edit** sous Linux
- **PsPad** sous windows
- **Netbeans** sous Windows ou Linux...

Voici un extrait de code qui pourrait être écrit par un débutant :

```
: cycle.stop -1 +to MAX_LIGHT_TIME MAX_LIGHT_TIME 0 = if
0 myLIGHTS ! then ;
```

Ce code sera parfaitement compilé par Z79Forth. Mais restera-t-il compréhensible dans le futur s'il faut le modifier ou le réutiliser dans une autre application ?

Ecrire un code FORTH lisible

Commençons par le nomage du mot à définir, ici **cycle.stop**. Z79Forth permet d'écrire des noms de mots de taille limitée. La taille des mots définis dans cette limite n'a aucune influence sur les performances de l'application finale. On dispose donc d'une certaine liberté pour écrire ces mots :

- à la manière de la programmation objet en javascript: **cycle.stop**
- à la manière CamelCoding **cycleStop**
- pour programmeur voulant un code très compréhensible **cycle-stop-lights**
- programmeur qui aime le code concis **cs1**

Z79Forth transformera tous les caractères d'un mot en caractères majuscule :

CYCLE.STOP, **CYCLESTOP**, **CYCLE-STOP-LIGHT**, **CSL**.

Il n'y a pas de règle de nommage. L'essentiel est que vous puissiez facilement relire votre code FORTH. Cependant, les programmeurs informatique en langage FORTH ont certaines habitudes :

- constantes en caractères majuscules **MAX_LIGHT_TIME**
- mot de définition d'autres mots **defColor:**, c'est à dire mot suivi de deux points ;

1 Integrated Development Environment = Environnement de Développement Intégré

- mot de transformation d'adresse **>date**, ici le paramètre sera incrémenté d'une certaine valeur pour pointer sur la donnée adéquate ;
- mot de stockage mémoire **date@** ou **date!**
- Mot d'affichage de donnée **date.**

Et qu'en est-il du nommage des mots FORTH dans une langue autre qu'en anglais ? Là encore, une seule règle : **liberté totale** ! Attention cependant, Z79Forth n'accepte pas les noms écrits dans des alphabets différents de l'alphabet latin. Vous pouvez cependant utiliser ces alphabets pour les commentaires :

```
: .date      \ Плакат сегодняшней даты
...code...  ;
```

OU

```
: .date      \ 海報今天的日期
...code...  ;
```

Les caractères provenant d'un codage linguistiques ne seront utilisables que dans votre code source.

Indentation du code source

Que le code soit sur deux lignes, dix lignes ou plus, ça n'a aucun effet sur les performances du code une fois compilé. Donc, autant indenter son code de manière structurée :

- une ligne par mot de structure de contrôle **if else then, begin while repeat...**
Pour le mot if, on peut de faire précéder du test logique qu'il traitera ;
- une ligne par exécution d'un mot prédéfini, précédé le cas échéant des paramètres de ce mot.

Exemple :

```
variable MAX_LIGHT_TIME
: cycle.stop
  -1 MAX_LIGHT_TIME +!
  MAX_LIGHT_TIME 0 =
  if
    0 myLIGHTS !
  then
;
;
```

Si le code traité dans une structure de contrôle est peu fourni, le code FORTH peut être compacté :

```
: cycle.stop
  -1 MAX_LIGHT_TIME +!
  MAX_LIGHT_TIME 0 =
```

```
if 0 myLIGHTS ! then
;
```

Les commentaires

Comme tout langage de programmation, le langage FORTH permet le rajout de commentaires dans le code source. Le rajout de commentaires n'a aucune conséquence sur les performances de l'application après compilation du code source.

En langage FORTH, nous disposons de deux mots pour délimiter des commentaires :

- le mot **(** suivi impérativement d'au moins un caractère espace. Ce commentaire est achevé par le caractère **)** ;
- le mot **** suivi impérativement d'au moins un caractère espace. Ce mot est suivi d'un commentaire de taille quelconque entre ce mot et la fin de la ligne.

Le mot **(** est largement utilisé pour les commentaires de pile. Exemples :

```
dup  ( n - n n )
swap ( n1 n2 - n2 n1 )
drop ( n -- )
emit ( c -- )
```

Les commentaires de pile

Comme nous venons de le voir, ils sont marqués par **(** et **)**. Leur contenu n'a aucune action sur le code FORTH en compilation ou en exécution. On peut donc mettre n'importe quoi entre **(** et **)**. Pour ce qui concerne les commentaires de pile, on restera très concis. Le signe **--** symbolise l'action d'un mot FORTH. Les indications figurant avant **--** correspondent aux données déposées sur la pile de données avant l'exécution du mot. Les indications figurant après **--** correspondent aux données laissées sur la pile de données après exécution du mot. Exemples :

- **words (--)** signifie que ce mot ne traite aucune donnée sur la pile de données ;
- **emit (c --)** signifie que ce mot traite une donnée en entrée et ne laisse rien sur la pile de données ;
- **bl (-- 32)** signifie que ce mot ne traite pas de donnée en entrée et laisse la valeur décimale 32 sur la pile de données ;

Il n'y a aucune limitation sur le nombre de données traitées avant ou après exécution du mot. Pour rappel, les indications entre **(** et **)** sont seulement là pour information.

Signification des paramètres de pile en commentaires

Pour commencer, une petite mise au point très importante s'impose. Il s'agit de la taille des données en pile. Avec Z79Forth, les données de pile occupent 2 octets. Ce sont donc

des entiers au format 16 bits. Cependant, certains mots traitent des données au format 8 bits. Alors on met quoi sur la pile de données ? Avec Z79Forth, ce seront **TOUJOURS DES DONNEES 16 BITS** ! Un exemple avec le mot **c** ! :

```
variable myDelemiter
64 myDelimiter c!    ( c addr -- )
```

Ici, le paramètre **c** indique qu'on empile une valeur entière au format 16 bits, mais dont la valeur sera toujours comprise dans l'intervalle [0..255].

Le paramètre standard est toujours **n**. S'il y a plusieurs entiers, on les numérote : **n1 n2 n3**, etc.

On aurait donc pu écrire l'exemple précédent comme ceci :

```
variable myDelemiter
64 myDelimiter c!    ( n1 n2 -- )
```

Mais c'est nettement moins explicite que la version précédente. Voici quelques symboles que vous serez amené à voir au fil des codes sources :

- **addr** indique une adresse mémoire littérale ou délivrée par une variable ;
- **c** indique une valeur 8 bits dans l'intervalle [0..255]
- **d** indique une valeur double précision.
Avec Z79Forth , c'est au format 32 bits ;
- **fl** indique une valeur booléenne, 0 ou non zéro ;
- **n** indique un entier. Entier signé 16 bits pour Z79Forth ;
- **str** indique une chaîne de caractère. Equivaut à **addr len --**
- **u** indique un entier non signé

Rien n'interdit d'être un peu plus explicite :

```
: SQUARE ( n -- n-exp2 )
  dup *
;
```

Commentaires des mots de définition de mots

Les mots de définition utilisent **create** et **does>**. Pour ces mots, il est conseillé d'écrire les commentaires de pile de cette manière :

```
\ define a command or data stream for SSD1306
: streamCreate: ( comp: <name> | exec: -- addr len )
  create
    here    \ leave current dictionnary pointer on stack
    0 c,    \ initial lenght data is 0
  does>
```

```

dup 1+ swap c@
\ send a data array
;

```

Ici, le commentaire est partagé en deux parties par le caractère | :

- à gauche, la partie action quand le mot de définition est exécuté, préfixé par **comp:**
- à droite la partie action du mot qui sera défini, préfixé par **exec:**

Au risque d'insister, ceci n'est pas un standard. Ce sont seulement des recommandations.

Les commentaires textuels

Ils sont inqués par le mot \ suivi obligatoirement par au moins un caractère espace et du texte explicatif :

```

\ store at <WORD> addr length of datas compiled beetween
\ <WORD> and here
: ;endStream ( addr-var len ---)
  dup 1+ here
  swap -      \ calculate cdata length
  \ store c in first byte of word defined by streamCreate:
  swap c!
;

```

Ces commentaires peuvent être écrits dans n'importe quel alphabet supporté par votre éditeur de code source :

```

\ 儲存在 <WORD> addr 之間編譯的資料長度
\ <WORD> 和這裡
: ;endStream ( addr-var len ---)
  dup 1+ here
  swap -      \ 計算 cdata 長度
  \ 將 c 儲存在由 StreamCreate 定義的字的的第一個位元組中:
  swap c!
;

```

Commentaire en début de code source

Avec une pratique de programmation intensive, on se retrouve rapidement avec des centaines, voire des milliers de fichiers source. Pour éviter des erreurs de choix de fichiers, il est fortement conseillé de marquer le début de chaque fichier source avec un commentaire :

```

\ *****
\ Manage commands for display
\   Filename:      commands.fs
\   Date:          21 may 2023
\   Updated:       21 may 2023

```



```

\   File Version:  1.0
\   Forth:         Z79Forth all versions 7.x++
\   Copyright:     Marc PETREMANN
\   Author:        Marc PETREMANN
\   GNU General Public License
\   *****

```

Toutes ces informations sont à votre libre choix. Elles peuvent devenir très utiles quand on revient des mois ou des années plus tard sur le contenu d'un fichier.

Pour conclure, n'hésitez pas à commenter et indenter vos fichiers sources en langage FORTH.

Moniteur de pile

Le contenu de la pile de données peut être affiché à tout moment grâce au mot **.s**. Voici la définition du mot **.DEBUG** qui exploite **.s** :

```

variable debugStack

: debugOn ( -- )
  -1 debugStack !
;

: debugOff ( -- )
  0 debugStack !
;

: .DEBUG
  debugStack @
  if
    cr ." STACK: " .s
    key drop
  then
;

```

Pour exploiter **.DEBUG**, il suffit de l'insérer dans un endroit stratégique du mot à mettre au point :

```

\ example of use:
: myTEST
  128 32 do
    i .DEBUG
    emit
  loop
;

```

Ici, on va afficher le contenu de la pile de données après exécution du mot **i** dans notre boucle **do loop**. On active la mise au point et on exécute **myTEST** :

```
debugOn
```

```
myTest
\ displays:
\ STACK: <1> 32
\ 2
\ STACK: <1> 33
\ 3
\ STACK: <1> 34
\ 4
\ STACK: <1> 35
\ 5
\ STACK: <1> 36
\ 6
\ STACK: <1> 37
\ 7
\ STACK: <1> 38
```

Quand la mise au point est activée par **debugOn**, chaque affichage du contenu de la pile de données met en pause notre boucle **do loop**. Exécuter **debugOff** pour que le mot **myTEST** s'exécute normalement.

Une fois le mot parfaitement mis au point, on peut enlever notre mot de traçage.

```
: myTEST
  128 32 do i emit
  loop
;
```

Dictionnaire / Pile / Variables / Constantes

Étendre le dictionnaire

Forth appartient à la classe des langages d'interprétation tissés. Cela signifie qu'il peut interpréter les commandes tapées sur la console, ainsi que compiler de nouveaux sous-programmes et programmes.

Le compilateur Forth fait partie du langage et des mots spéciaux sont utilisés pour créer de nouvelles entrées de dictionnaire (c'est-à-dire des mots). Les plus importants sont `:` (commencer une nouvelle définition) et `;` (termine la définition). Essayons ceci en tapant :

```
: *+ * + ;
```

Ce qui s'est passé? L'action de `:` est de créer une nouvelle entrée de dictionnaire nommée `*+` et passer du mode interprétation au mode compilation. En mode compilation, l'interpréteur recherche les mots et, plutôt que de les exécuter, installe des pointeurs vers leur code. Si le texte est un nombre, au lieu de le pousser sur la pile, Z79Forth construit le nombre dans le dictionnaire. L'action d'exécution de `*+` est donc d'exécuter séquentiellement les mots définis précédemment `*` et `+`.

Le mot `;` est spécial. C'est un mot immédiat et il est toujours exécuté, même si le système est en mode compilation. Ce que fait `;` est double. Tout d'abord, il installe le code qui renvoie le contrôle au niveau externe suivant de l'interpréteur et, deuxièmement, il revient du mode compilation au mode interprétation.

Maintenant, essayez votre nouveau mot :

```
decimal 5 6 7 *+ . \ affiche 47
```

Cet exemple illustre deux activités principales de travail dans Forth : : ajouter un nouveau mot au dictionnaire, et l'essayer dès qu'il a été défini.

Piles et notation polonaise inversée

Forth a une pile explicitement visible qui est utilisée pour passer des nombres entre les mots (commandes). Utiliser Forth efficacement vous oblige à penser en termes de pile. Cela peut être difficile au début, mais comme pour tout, cela devient beaucoup plus facile avec la pratique.

En FORTH, La pile est analogue à une pile de cartes avec des nombres écrits dessus. Les nombres sont toujours ajoutés au sommet de la pile et retirés du sommet de la pile. Z79Forth intègre deux piles: la pile de paramètres et la pile de retour, chacune composée d'un certain nombre de cellules pouvant contenir des nombres de 16 bits.

La ligne d'entrée FORTH:

```
decimal 2 5 73 -16
```

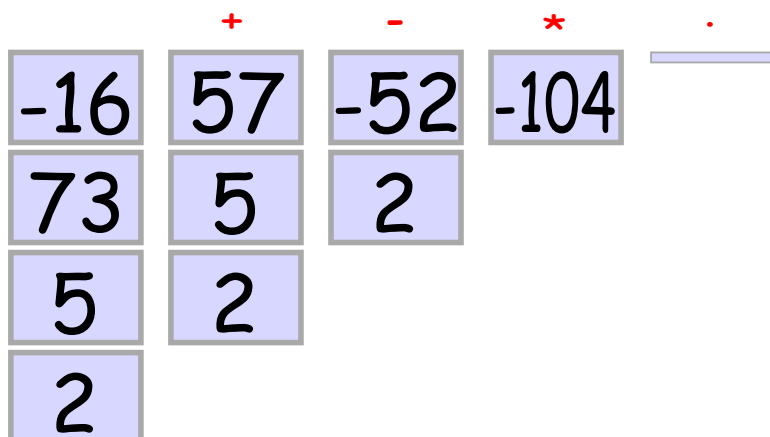
laisse la pile de paramètres dans l'état

Cellule	contenu	commentaire
0	-16	(TOS) Sommet pile
1	73	(NOS) Suivant dans la pile
2	5	
3	2	

Nous utiliserons généralement une numérotation relative à base zéro dans les structures de données Forth telles que piles, tableaux et tables. Notez que, lorsqu'une séquence de nombres est saisie comme celle-ci, le nombre le plus à droite devient *TOS* et le nombre le plus à gauche se trouve au bas de la pile.

Supposons que nous suivions la ligne d'entrée d'origine avec la ligne

```
+ - * .
```



Les opérations produiraient les opérations de pile successives:

Après les deux lignes, la console affiche :

```
decimal 2 5 73 -16
+ - * .           \ affiche: -104
```

Notez que Z79Forth affiche commodément les éléments de la pile lors de l'interprétation de chaque ligne et que la valeur de -16 est affichée sous la forme d'entier non signé 16 bits. Le mot `.` consomme la valeur de données -104, laissant la pile vide. Si nous exécutons `.` sur la pile maintenant vide, l'interpréteur externe abandonne avec une erreur de pointeur de pile **Data stack underflow**.

La notation de programmation où les opérandes apparaissent en premier, suivis du ou des opérateurs est appelée *Notation polonaise inverse* (RPN).

Manipulation de la pile de paramètres

Étant un système basé sur la pile, Z79Forth doit fournir des moyens de mettre des nombres sur la pile, pour les supprimer et réorganiser leur ordre. On a déjà vu qu'on peut

mettre des nombres sur la pile simplement en les tapant. Nous pouvons également intégrer les nombres dans la définition d'un mot FORTH.

Le mot **drop** supprime un numéro du sommet de la pile mettant ainsi le suivant au sommet. Le mot **swap** échange les 2 premiers numéros. **dup** copie le nombre au sommet, poussant tout les autres numéros vers le bas. **rot** fait pivoter les 3 premiers nombres. Ces

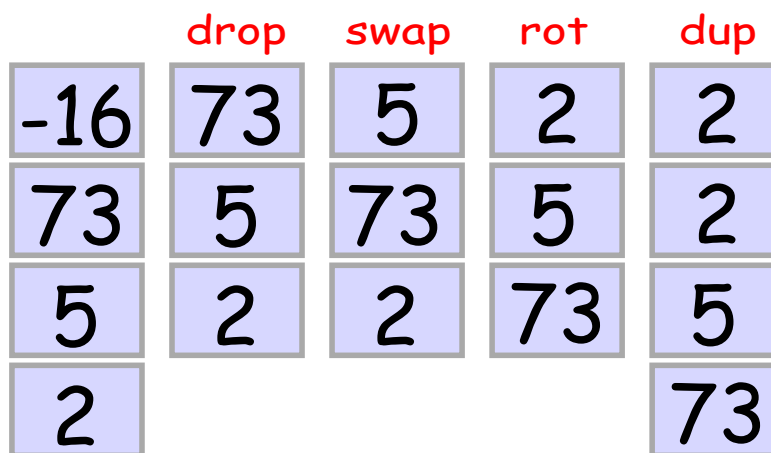


Figure 6: manipulation de la pile de données

actions sont présentées ci-dessous.

La pile de retour et ses utilisations

Lors de la compilation d'un nouveau mot, Z79Forth établit des liens entre le mot appelant et les mots définis précédemment qui doivent être invoqués par l'exécution du nouveau mot. Ce mécanisme de liaison, lors de l'exécution, utilise la pile de retour (rstack). L'adresse du mot suivant à invoquer est placée sur la pile de retour de sorte que, lorsque le mot courant est terminé en cours d'exécution, le système sait où passer au mot suivant. Comme les mots peuvent être imbriqués, il doit y avoir une pile de ces adresses de retour.

En plus de servir de réservoir d'adresses de retour, l'utilisateur peut également stocker et récupérer à partir de la pile de retour, mais cela doit être fait avec soin car la pile de retour est essentielle à l'exécution du programme. Si vous utilisez la pile de retour pour le stockage temporaire, vous devez la remettre dans son état d'origine, sinon vous ferez probablement planter le système Z79Forth. Malgré le danger, il y a des moments où l'utilisation de pile de retour comme stockage temporaire peut rendre votre code moins complexe.

Pour stocker dans la pile, utilisez **>r** pour déplacer le sommet de la pile de paramètres vers le haut de la pile de retour. Pour récupérer une valeur, **r>** déplace la valeur supérieure de la pile de retour vers le sommet de la pile de paramètres. Le mot **r@** copie le haut de la pile de retour dans la pile de paramètres.

Les mots **>r** et **r>** ne peuvent être utilisés que dans une définition FORTH.

Utilisation de la mémoire

Dans Z79Forth, les nombres 16 bits sont extraits de la mémoire vers la pile par le mot **@** (fetch) et stocké du sommet à la mémoire par le mot **!** (store). **@** attend une adresse sur la pile et remplace l'adresse par son contenu. **!** attend un nombre et une adresse pour le stocker. Il place le numéro dans l'emplacement de mémoire référencé par l'adresse, consommant les deux paramètres dans le processus.

Les nombres non signés qui représentent des valeurs de 8 bits (octets) peuvent être placés dans des caractères de la taille d'un caractère, en utilisant **c@** et **c!**.

```
create testVar
  cell allot
$F7 testVar c!
testVar c@ . \ affiche 247
```

Variables

Une variable est un emplacement nommé en mémoire qui peut stocker un nombre, tel que le résultat intermédiaire d'un calcul, hors de la pile. Par exemple :

```
variable x
```

crée un emplacement de stockage nommé, **x**, qui s'exécute en laissant l'adresse de son emplacement de stockage au sommet de la pile:

```
x . \ affiche l'adresse du contenu de x
```

Nous pouvons alors aller chercher ou stocker à cette adresse :

```
variable x
3 x !
x @ . \ affiche: 3
```

Constantes

Une constante est un nombre que vous ne voudriez pas changer pendant l'exécution d'un programme. Le résultat de l'exécution du mot associé à une constante est la valeur des données restant sur la pile.

```
\ définit une matrice X Y
64 constant matrix-width
16 constant matrix-height

create my-matrix
  matrix-width matrix-height * allot
```

Valeurs pseudo-constantes

Une valeur définie avec **value** est un type hybride de variable et constante. Nous définissons et initialisons une valeur. Cette valeur est invoquée comme nous le ferions pour une constante. On peut aussi changer une valeur comme on peut changer une variable.

```
decimal
13 value thirteen
thirteen .      \ display: 13
47 to thirteen
thirteen .      \ display: 47
```

Le mot **to** fonctionne également dans les définitions de mots, en remplaçant la valeur qui le suit par tout ce qui est actuellement au sommet de la pile. Vous devez faire attention à ce que **to** soit suivi d'une valeur définie par **value**.

Outils de base pour l'allocation de mémoire

Les mots **create** et **allot** sont les outils de base pour réserver un espace mémoire et y attacher une étiquette. Par exemple, la transcription suivante montre une nouvelle entrée de dictionnaire **graphic-array** :

```
create graphic-array ( --- addr )
  %00000000 c,
  %00000010 c,
  %00000100 c,
  %00001000 c,
  %00010000 c,
  %00100000 c,
  %01000000 c,
  %10000000 c,
```

Lorsqu'il est exécuté, le mot **graphic-array** empilera l'adresse de la première entrée.

Nous pouvons maintenant accéder à la mémoire allouée à **graphic-array** en utilisant les mots de récupération et de stockage expliqués plus tôt. Pour calculer l'adresse du troisième octet attribué à **graphic-array** on peut écrire **graphic-array 2 +**, en se rappelant que les indices commencent à 0.

```
30 graphic-array 2 + c!
graphic-array 2 + c@ .      \ affiche 30
```

Les mots de création de mots

FORTH est plus qu'un langage de programmation. C'est un méta-langage. Un méta-langage est un langage utilisé pour décrire, spécifier ou manipuler d'autres langages.

Avec Z79Forth, on peut définir la syntaxe et la sémantique de mots de programmation au-delà du cadre formel des définitions de base.

On a déjà vu les mots définis par **constant**, **variable**, **value**. Ces mots servent à gérer des données numériques.

On a également utilisé le mot **create**. Ce mot crée un en-tête permettant d'accéder à une zone de données mis en mémoire. Exemple :

```
create temperatures
  34 , 37 , 42 , 36 , 25 , 12 ,
```

Ici, chaque valeur est stockée dans la zone des paramètres du mot **temperatures** avec le mot **,**.

Avec Z79Forth, on va voir comment personnaliser l'exécution des mots définis par **create**.

Utilisation de does>

Il y a une combinaison de mots-clés "**CREATE**" et "**DOES>**", souvent utilisée ensemble pour créer des mots (mots de vocabulaire) personnalisés avec des comportements spécifiques.

Voici comment cela fonctionne généralement en Forth :

- **CREATE** : ce mot-clé est utilisé pour créer un nouvel espace de données dans le dictionnaire Z79Forth. Il prend en charge un argument, qui est le nom que vous donnez à votre nouveau mot ;
- **DOES>** : ce mot-clé est utilisé pour définir le comportement du mot que vous venez de créer avec **CREATE**. Il est suivi d'un bloc de code qui spécifie ce que le mot devrait faire lorsqu'il est rencontré pendant l'exécution du programme.

Ensemble, cela ressemble à quelque chose comme ceci :

```
forth
CREATE mon-nouveau-mot
  \ code à exécuter lorsqu'on rencontre mon-nouveau-mot
DOES>
;
```

Lorsque le mot **mon-nouveau-mot** est rencontré dans le programme FORTH, le code spécifié dans la partie **does> ... ;** sera exécuté.

```
\ define a color, similar as constant
```



```

: defCOLOR:
  create ( addr1 -- <name> )
  ,
  does> ( -- regAddr )
  @
;

```

Ici, on définit le mot de définition **defCOLOR:** qui a exactement la même action que **constant**. Mais pourquoi créer un mot qui recrée l'action d'un mot qui existe déjà ?

```
$0000FF constant BLUE
```

OU

```
$0000FF defCOLOR: BLUE
```

sont semblables. Cependant, en créant nos couleurs avec **defCOLOR:** on a les avantages suivants :

- un code Z79Forth source plus lisible. On détecte facilement toutes les constantes nommant une couleur ;
- on se laisse la possibilité de modifier la partie **does>** de **defCOLOR:** sans avoir ensuite à réécrire les lignes de code qui n'utiliseraient pas **defCOLOR:**

Voici un cas classique, le traitement d'un tableau de données :

```

\ mot de définition pour tableau à une dimension
: array ( comp: -- <name> | exec: index <name> -- addr )
  create
  does>
    swap cell * +
;
array temperatures
  21 ,    32 ,    45 ,    44 ,    28 ,    12 ,
0 temperatures @ . \ display 21
5 temperatures @ . \ display 12

```

L'exécution de **temperatures** doit être précédé de la position de la valeur à extraire dans ce tableau. Ici nous récupérons seulement l'adresse contenant la valeur à extraire.

Exemple de gestion de couleur

Dans ce premier exemple, on définit le mot **color:** qui va récupérer la couleur à sélectionner et la stocker dans une variable :

```

0 value currentCOLOR

\ define word as COLOR constant
: color: ( n -- <name> )
  create
  ,

```

```

does>
    @ to currentCOLOR
;

$00 color: setBLACK
$ff color: setWHITE

```

L'exécution du mot **setBLACK** ou **setWHITE** simplifie considérablement le code Z79Forth. Sans ce mécanisme, il aurait fallu répéter régulièrement une de ces lignes :

```
$00 currentCOLOR !
```

Ou

```

$00 constant BLACK
BLACK currentCOLOR !

```

Exemple, écrire en pinyin

Le pinyin est couramment utilisé dans le monde entier pour enseigner la prononciation du chinois mandarin, et il est également utilisé dans divers contextes officiels en Chine, comme les panneaux de signalisation, les dictionnaires et les manuels d'apprentissage. Il facilite l'apprentissage du chinois pour les personnes dont la langue maternelle utilise l'alphabet latin.

Pour écrire en chinois sur un clavier QWERTY, les Chinois utilisent généralement un système appelé "pinyin input" ou "saisie pinyin". Pinyin est un système de romanisation du chinois mandarin, qui utilise l'alphabet latin pour représenter les sons du mandarin.

Sur un clavier QWERTY, les utilisateurs tapent les sons du mandarin en utilisant la romanisation pinyin. Par exemple, si quelqu'un veut écrire le caractère "你" ("nǐ" signifiant "tu" ou "toi" en français), il peut taper "ni".

Dans ce code très simplifié, on peut programmer des mots pinyin pour écrire en mandarin. Le code ci-après ne fonctionne qu'avec le terminal PuTTY :

```

\ Work only with PuTTY terminal
: chinese:
    create ( c1 c2 c3 -- )
        c, c, c,
    does>
        3 type
;

```

Pour trouver le code UTF8 d'un caractère chinois, copiez le caractère chinois, depuis Google Translate par exemple. Exemple :

```
Good Morning --> 早安 (Zao an)
```

Copiez 早 et allez dans le terminal PuTTY et tapez :

```
key key key \ followed by key <enter>
```

collez le caractère 早. Z79Forth doit afficher les codes suivants :

```
230 151 169
```

Pour chaque caractère chinois, on va exploiter ces trois codes ainsi :

```
169 151 230 chinese: Zao  
137 174 229 chinese: An
```

Utilisation :

```
Zao An \ display 早安
```

Avouez quand même que programmer ainsi c'est autre chose que ce qu'on peut faire en langage C. Non ?

Ressources

- **Z79FORTH / Facebook**
<https://www.facebook.com/groups/505661250539263>
- **Z79Forth Blog**
<https://z79forth.blogspot.com/>

GitHub

- **Z79forth** est un voyage dans l'informatique rétro qui tire parti des technologies modernes lorsque cela est approprié (CMOS, USB et CompactFlash)
<https://github.com/frenchie68/Z79Forth>

Youtube

- CPE1704TKS: Notes de terrain d'ingénierie issues d'une session de débogage
<https://www.youtube.com/watch?v=OFIxfCywh0Q>

Index

and.....	17	dup.....	37	value.....	39
BASE.....	22	EMIT.....	26	variable.....	38
c!.....	38	format HH:MM:SS.....	20	vitesse de transmission série...9	
c@.....	38	hex.....	12	35
caractéristiques.....	6	HEX.....	22	35
cavalier JP1.....	9	HOLD.....	24	27
cavaliers JP2 à JP5.....	9	mémoire.....	38	.s.....	33
constant.....	38	pile de retour.....	37	@.....	38
create.....	21, 40	préfixe.....	23	#.....	24
decimal.....	12	ressources.....	44	#>.....	24
DECIMAL.....	22	S".....	27	#S.....	24
DOES>.....	40	SPACE.....	27	<#.....	24
drop.....	37	u.....	15		