

# Le manuel pour Z79FORTH

version 1.5 - samedi 26 octobre 2024



## Auteur

- François LAAGEL
- Marc PETREMANN

## Collaborateur

- ....

## Contents

Auteur.....	1
Collaborateur.....	1
<b>Introduction.....</b>	<b>5</b>
Objet.....	5
Déclinaisons.....	6
<b>La carte Z79Forth.....</b>	<b>7</b>
Les composants de la carte Z79Forth.....	8
Circuits intégrés, mémoires, processeur.....	8
Le processeur HD63C09E.....	9
<b>Alimentation de la carte Z79Forth.....</b>	<b>10</b>
Le câble USB.....	10
<b>Communiquer avec la carte Z79Forth.....</b>	<b>11</b>
Utilisation d'un terminal.....	12
Installer et utiliser Tera Term.....	12
Paramétrage de Tera Term.....	12
Première communication avec Tera Term.....	14
Compiler du code source FORTH vers la carte Z79Forth.....	14
<b>Edition et gestion des fichiers sources pour Z79Forth.....</b>	<b>16</b>
Les éditeurs de fichiers texte.....	16
Utiliser un IDE.....	17
Stockage sur GitHub.....	19
Quelques bonnes pratiques.....	20
<b>Utiliser les nombres avec Z79Forth.....</b>	<b>21</b>
Les nombres avec l'interpréteur FORTH.....	21
Saisie des nombres avec différentes base numérique.....	21
Changement de base numérique.....	22
Binaire et hexadécimal.....	23
Taille des nombres sur la pile de données FORTH.....	25
Accès mémoire et opérations logiques.....	26
<b>Un vrai FORTH 16 bits avec Z79Forth.....</b>	<b>28</b>
Les valeurs sur la pile de données.....	28
Les valeurs en mémoire.....	28
Traitement par mots selon taille ou type des données.....	29
Conclusion.....	30
<b>Affichage des nombres et chaînes de caractères.....</b>	<b>32</b>
Changement de base numérique.....	32
Préfixer les nombres entiers.....	33
Définition de nouveaux formats d'affichage.....	34
Affichage des caractères et chaînes de caractères.....	36
<b>Commentaires et mise au point.....</b>	<b>38</b>

Ecrire un code FORTH lisible.....	38
Utiliser des caractères UNICODE dans les noms.....	39
Indentation du code source.....	40
Les commentaires.....	41
Les commentaires de pile.....	41
Signification des paramètres de pile en commentaires.....	41
Commentaires des mots de définition de mots.....	42
Les commentaires textuels.....	43
Commentaire en début de code source.....	43
Moniteur de pile.....	44
Le décompilateur.....	45
<b>Dictionnaire / Pile / Variables / Constantes.....</b>	<b>47</b>
Étendre le dictionnaire.....	47
Piles et notation polonaise inversée.....	47
Manipulation de la pile de paramètres.....	48
La pile de retour et ses utilisations.....	49
Utilisation de la mémoire.....	49
Variables.....	50
Constantes.....	50
Valeurs pseudo-constantes.....	50
Outils de base pour l'allocation de mémoire.....	51
<b>Les mots de création de mots.....</b>	<b>52</b>
Utilisation de does>.....	52
Exemple de gestion de couleur.....	53
Exemple, écrire en pinyin.....	54
<b>Marquage des couches fonctionnelles avec MARKER.....</b>	<b>56</b>
Fonctionnement du mot FORGET.....	56
Marquage avec le mot marker.....	57
<b>La structure du dictionnaire FORTH.....</b>	<b>59</b>
Accès au champ du code exécutable.....	59
Structure d'un mot FORTH.....	60
Le champ de nom.....	60
Le champ de lien.....	61
Recherche d'un mot dans le dictionnaire.....	63
<b>Structures de données pour Z79Forth.....</b>	<b>66</b>
Préambule.....	66
Les tableaux en FORTH.....	66
Tableau de données 16 bits à une dimension.....	66
Mots de définition de tableaux.....	67
Lire et écrire dans un tableau.....	67
Gestion de structures complexes.....	68
Définition de struct et field.....	68
Exemples de structures.....	70
<b>Les caractères Unicode avec emit et key.....</b>	<b>72</b>
Le codage Unicode.....	72

Récupérer un caractère Unicode depuis le terminal.....	73
Affichage des caractères Unicode.....	74
Caractères Unicode dans les mots Forth.....	76
<b>Utiliser du code binaire.....</b>	<b>78</b>
Rétrocomputing et retour aux fondamentaux.....	78
FORTH : compact et élégant.....	79
Une version FORTH à chaînage direct.....	80
Décompiler les définitions Z79Forth.....	80
Codage de définitions en langage machine.....	82
<b>Ressources.....</b>	<b>84</b>
GitHub.....	84
Youtube.....	84

# 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**.

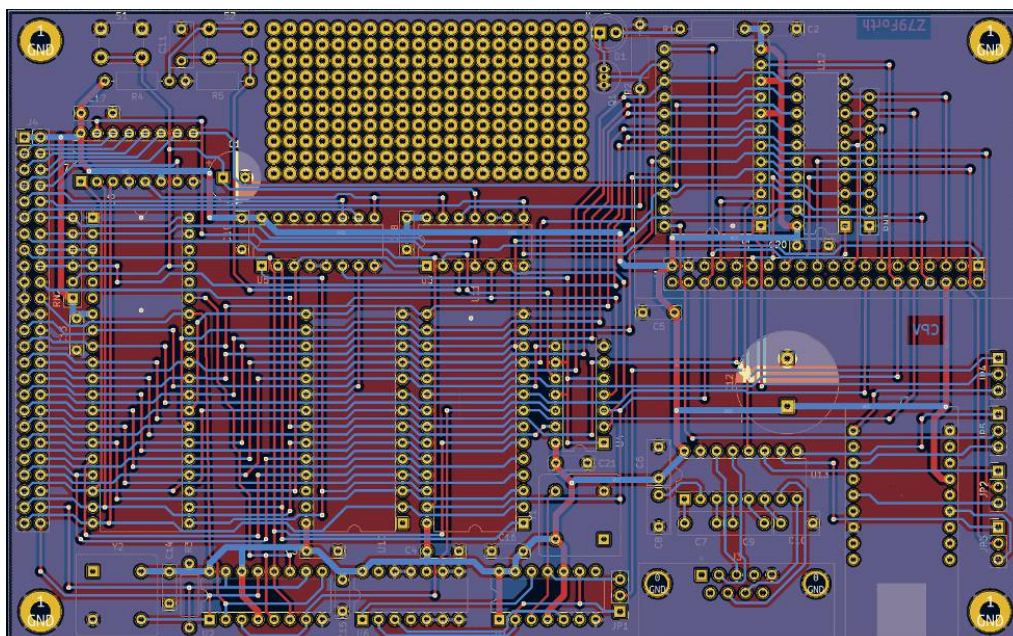
## Objet

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.

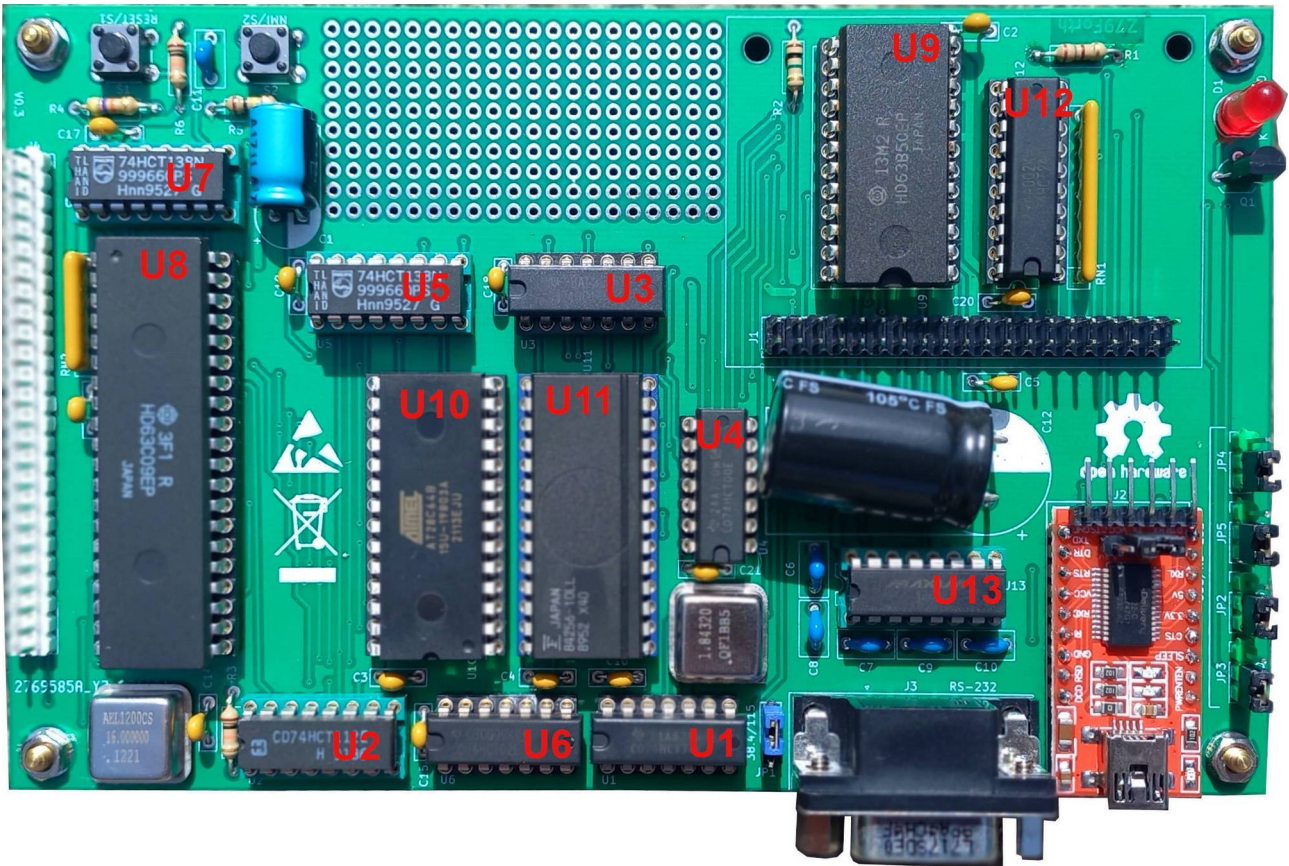


Figure 1: La carte Z79Forth

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.

## Les composants de la carte Z79Forth

Sur la carte Z79Forth, tous les composants sont identifiés par une lettre et des chiffres :

- **Un** pour les circuits intégrés, mémoires et processeur ;
- **Rn** pour les résistances, **Cn** pour les condensateurs ;
- **S1** et **S2** pour les interrupteurs ;
- **Jn** pour les broches d'accès bus, sélection de fonctions ;
- **Qn** pour les transistors
- **Yn** pour les quartz d'horloge.

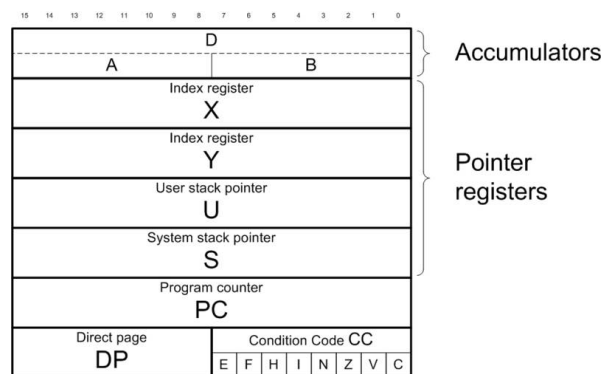
### Circuits intégrés, mémoires, processeur

- **U1 74HCT74** high-speed Si-gate CMOS devices and are pin compatible with low power Schottky TTL (LSTTL).
- **U2 74HCT112E** dual negative-edge triggered JK flip-flop. It features individual J and K inputs, clock (nCP) set (nSD) and reset (nRD) inputs.
- **U3 74HCT14** high-speed Si-gate CMOS devices and are pin compatible with low power Schottky TTL (LSTTL). The 74HC14 and 74HCT14 provide six inverting buffers with Schmitt-trigger action. They are capable of transforming slowly changing input signals into sharply defined, jitter-free output signals.
- **U4 74HCT00** high-speed Si-gate CMOS devices and are pin compatible with low power Schottky TTL (LSTTL). The 74HC00/74HCT00 provide the 2-input NAND function.
- **U5 74HCT138** high-speed Si-gate CMOS devices and are pin compatible with low power Schottky TTL (LSTTL).
- **U6 74HCT02** high-speed Si-gate CMOS devices and are pin compatible with low power Schottky TTL (LSTTL). The 74HC/HCT02 provide the 2-input NOR function.

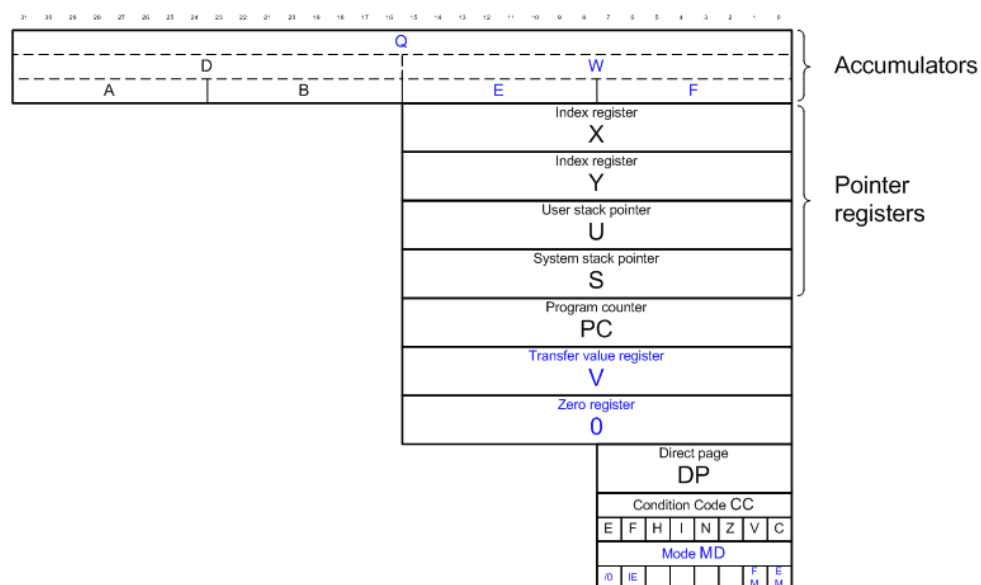


- **U7 74HCT138** high-speed Si-gate CMOS devices and are pin compatible with low power Schottky TTL (LSTTL).
- **U8** processeur **HD63C09E**, est le coeur de la carte Z79Forth. The Z79Forth reference board uses the HD63C09E clocked at 4 MHz. The choice of that particular frequency was meant to allow some bus cycles to be stretched down easily to 1 MHz, so as to be able to possibly accomodate extensions using components of the MC6800 product line. At this point, it is worth mentioning the fact that the HD6309 offers reduced power consumption and an extended instruction set which is heavily used by the Z79Forth software.
- **U9** HD63B50
- **U10 AT28C64B** high-performance electrically-erasable and programmable read only memory (EEPROM) ;
- **U11 F84256** RAM statique 32 KB 8 bits
- **U12 74HCT245** interface de bus bidirectionnel Octal ;
- **U13 max232EPE** line drivers/receivers intended for all EIA/TIA-232E and V.28/V.24 communications interfaces, particularly applications where  $\pm 12V$  is not available.

## Le processeur HD63C09E



6809 Internal Registers



# 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 USB 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 :

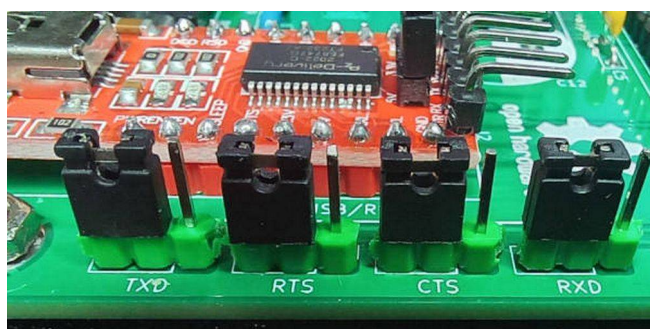


Figure 3: position des cavaliers JP2 à JP5

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 :

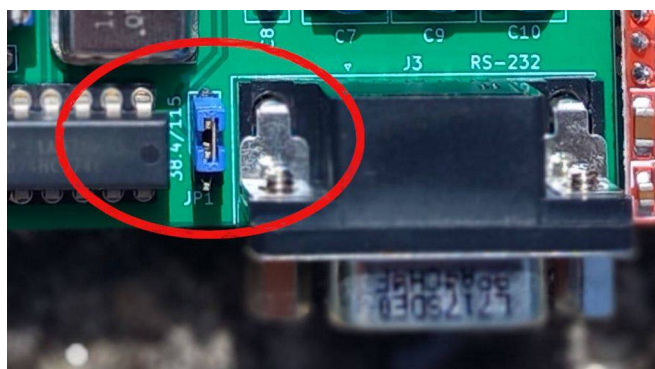


Figure 4: position du cavalier JP1

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 ;
- **Minicom** sous Linux

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

## Installer et utiliser Tera Term

Pour les programmeurs utilisant Windows, je conseille le terminal Tera Term. Vous pouvez le télécharger depuis ce site :

<https://teratermproject.github.io/index-en.html>

La version 5.2 n'est pas volumineuse. Elle occupe 8,8MB avant installation. Téléchargez le fichier *teraterm-5.2.exe*. Une fois téléchargé, ouvrez ce fichier et suivez les instructions jusqu'à installation complète.

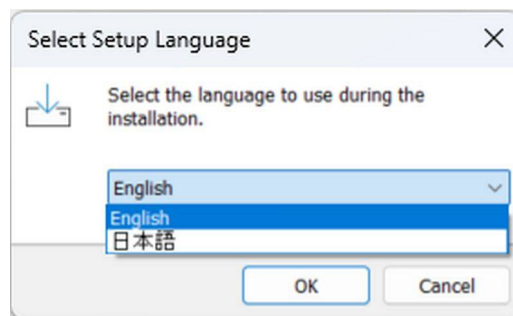


Figure 5: Sélection langue de Tera Term

Sélectionnez l'anglais comme langue d'utilisation de Tera Term. L'installation est simple et rapide.

## Paramétrage de Tera Term

Pour communiquer avec Z79Forth, il faut régler certains paramètres.

Cliquer sur *Setup* et sélectionnez *Serial Port*



Figure 6: paramètres liaison série

Ne sélectionnez pas le port série. Entrez les autres paramètres comme décrits dans la figure ci-dessus.

Cliquez sur *Setup* et sélectionnez *Windows*. Cette fenêtre permet de régler la couleur du texte dans le terminal et la couleur de fond. Faites selon vos préférences.

Cliquez sur *Setup* et sélectionnez *Terminal*

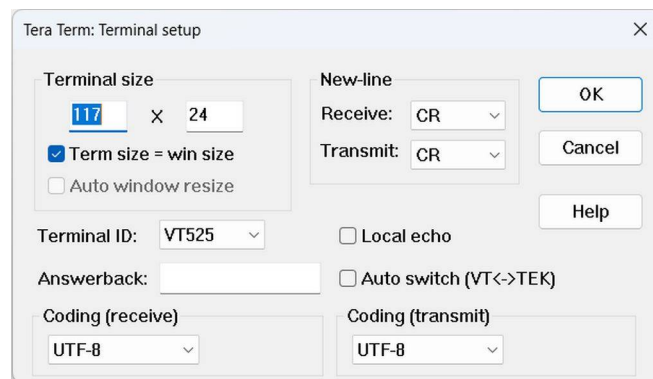


Figure 7: Sélection du terminal

Ici, on a sélectionné le terminal **VT525**. Ce terminal permet d'exploiter tous les caractères existants : ASCII standard, ASCII étendu, caractères accentués dans de nombreuses langues, idéogrammes chinois, japonais, coréens, etc. Cet aspect fait l'objet du chapitre *Les caractères Unicode avec emit et key*.

A l'aube de l'informatique, les premiers ordinateurs étaient très gros. Et pour les utiliser, on communiquait avec ces machines via un terminal. Un terminal VT (pour Video Terminal), est un dispositif informatique qui permet à un utilisateur d'interagir avec un ordinateur distant. Il s'agit essentiellement d'un écran et d'un clavier qui envoie des commandes à un ordinateur principal et transmet les résultats vers un écran local. Les terminaux VT ont joué un rôle crucial dans le développement de l'informatique, en particulier dans les environnements Unix et mainframe.

Tera Term est un émulateur de terminal. Pour communiquer avec notre carte Z79Forth, nous n'exploiterons que le port série **COMx**, où **x** est le numéro de port série.

## Première communication avec Tera Term

Une fois Tera Term paramétré, cliquez sur *Setup* et sélectionnez *Save setup*. Sauvegardez les paramètres sous le nom de fichier *TERATERM.INI* proposé par défaut.

Fermez Tera Term. Relancez Tera Term. Votre micro-ordinateur doit être relié à la carte Z79Forth par port USB. Si tout se passe bien, la mise en relation est immédiate. Si ce n'est pas le cas :

- cliquez sur *File* et sélectionnez *New Connection*
- cliquez sur *Serial*. Le port proposé est en général celui relié à la carte Z79Forth.

Si la liaison est effective, appuyez plusieurs fois sur la touche RET du clavier. Voici ce que ça donne :

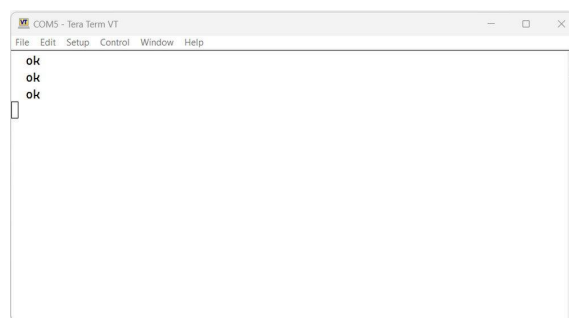


Figure 8: La communication entre Z79Forth et Tera Term est établie

Les paramètres sont à peu près similaires avec d'autres programmes d'émulation de terminaux.

## Compiler du code source FORTH vers la carte Z79Forth

Tout d'abord, rappelons que le langage FORTH est sur la carte Z79Forth ! FORTH n'est pas sur votre PC. Donc, on ne peut pas compiler le code source d'un programme en langage FORTH sur le PC.

Pour compiler un programme en langage FORTH, il faut au préalable ouvrir un fichier source sur le PC avec l'éditeur de votre choix.

Ensuite, on copie le code source à compiler. Ici, un code source ouvert avec Wordpad :



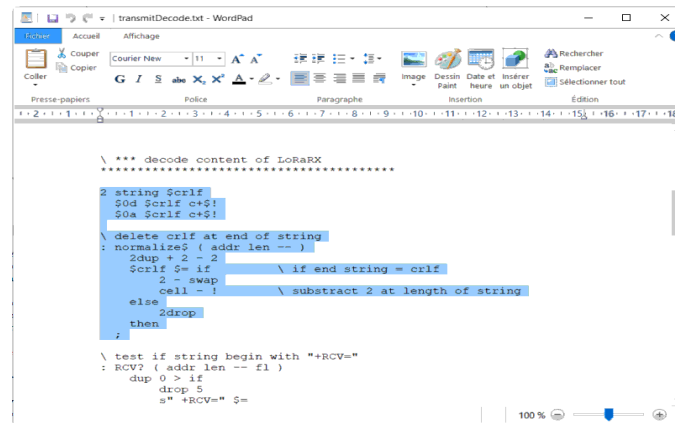


Figure 9: Edition code source FORTH avec Wordpad

Sélectionnez le code source ou la portion de code qui vous intéresse. Puis cliquez sur *copier*. Le code sélectionné est dans le tampon d'édition du PC.

Cliquez sur la fenêtre du terminal Tera Term. Faites *Coller*.

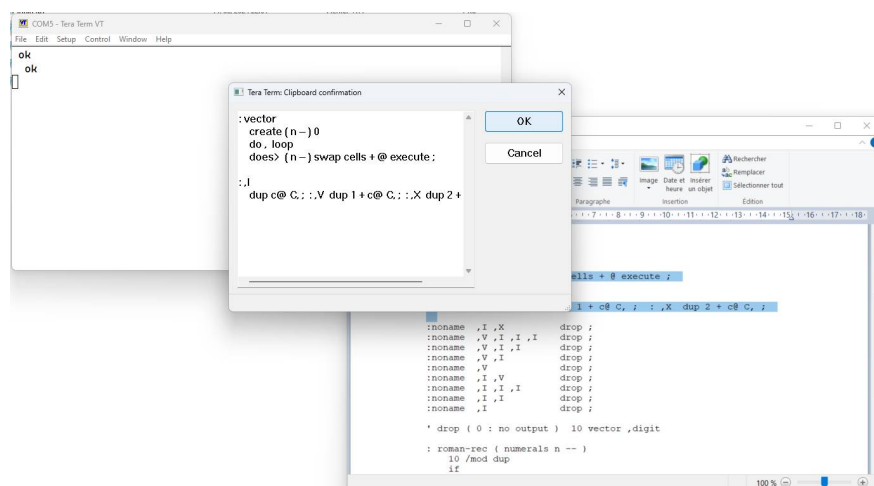


Figure 10: Copié/collé d'une portion de code depuis Wordpad vers Z79Forth via Tera Term

Le code FORTH ainsi copié/collé sera immédiatement interprété ou compilé par Z79Forth.

Pour exécuter un code compilé, il suffit de taper le mot FORTH à lancer, ce depuis le terminal Tera Term.

# Edition et gestion des fichiers sources pour Z79Forth

Comme pour la très grande majorité des langages de programmation, les fichiers sources écrits en langage FORTH sont au format texte simple. L'extension des fichiers en langage FORTH est libre :

- **txt** extension générique pour tous les fichiers texte ;
- **forth** utilisé par certains programmeurs FORTH ;
- **fth** forme compressée pour « FORTH » ;
- **4th** autre forme compressée pour « FORTH » ;

## Les éditeurs de fichiers texte

Sous Windows, l'éditeur de fichiers **edit** est le plus simple :

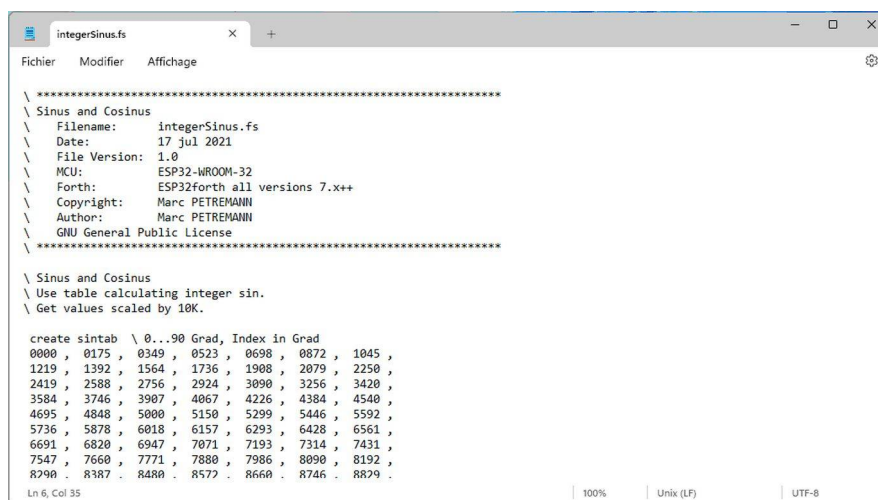


Figure 11: édition avec edit sous windows 11

Les autres éditeurs, comme **WordPad** sont déconseillés, car vous risquez de sauvegarder le code source en langage FORTH dans un format de fichier non compatible avec Z79Forth.

Sous Linux, l'équivalent s'appelle **gEdit**. MacOS dispose aussi d'un éditeur de texte simple.

Si vous utilisez une extension de fichier personnalisé, comme **4th**, pour vos fichiers source en langage FORTH, il faut faire reconnaître cette extension de fichiers par votre système pour permettre leur ouverture par l'éditeur de texte.

## Utiliser un IDE

Rien ne vous empêche d'utiliser un IDE<sup>1</sup>. Pour ma part, j'ai une préférence pour **Netbeans** que j'utilise aussi pour PHP, MySQL, Javascript, C, assembleur... C'est un IDE très puissant et aussi performant qu'**Eclipse** :

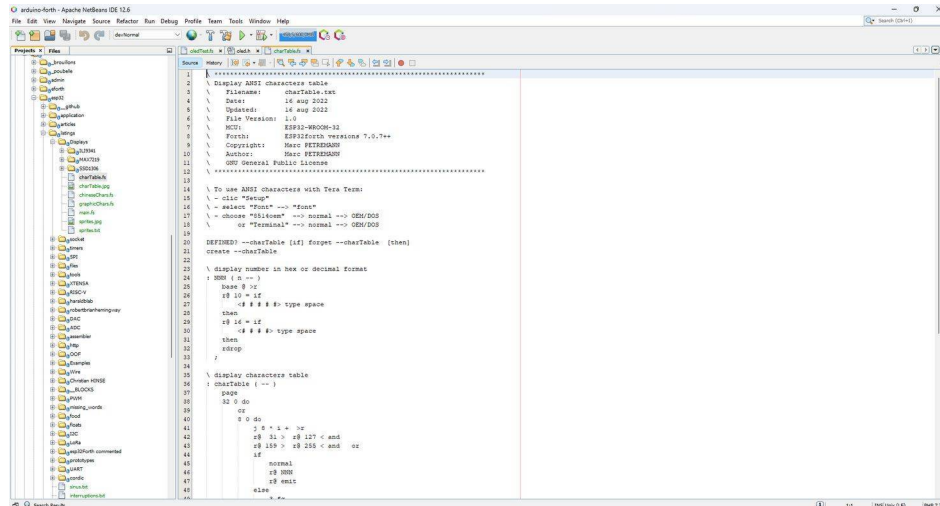


Figure 12: édition avec Netbeans

**Netbeans** offre plusieurs fonctionnalités intéressantes :

- gestion de versions avec **GIT** ;
- récupération de versions précédentes de fichiers modifiés ;
- comparaison de fichiers avec **Diff** ;
- transmission en un clic en **FTP** vers l'hébergement en ligne de votre choix ;

Avec l'option **GIT**, possibilité de partager des fichiers sur un dépôt et de gérer des collaborations sur des projets complexes. En local ou en collaboration, **GIT** permet la gestion des versions différentes d'un même projet, puis de fusionner ces versions. Vous pouvez créer votre dépôt GIT local. A chaque *Commit* d'un fichier ou d'un répertoire complet, les développements sont conservés en l'état. Ceci permet de retrouver d'anciennes versions d'un même fichier ou dossier de fichiers.

---

1 Integrated Development Environment

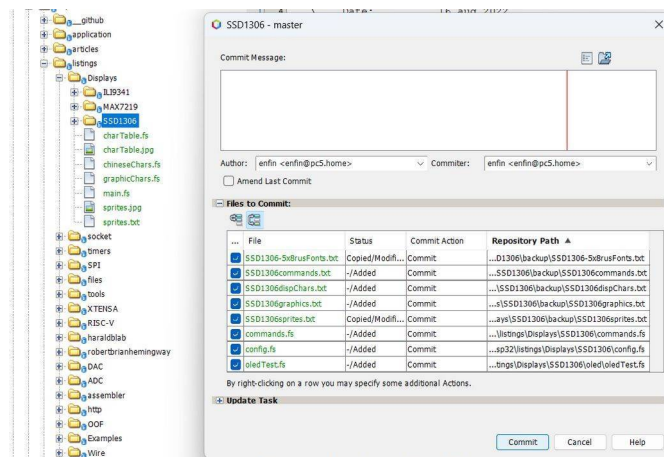


Figure 13: opération GIT dans Netbeans

Avec NetBeans, vous pouvez définir un embranchement de développement pour un projet complexe. Ici on crée une nouvelle branche :

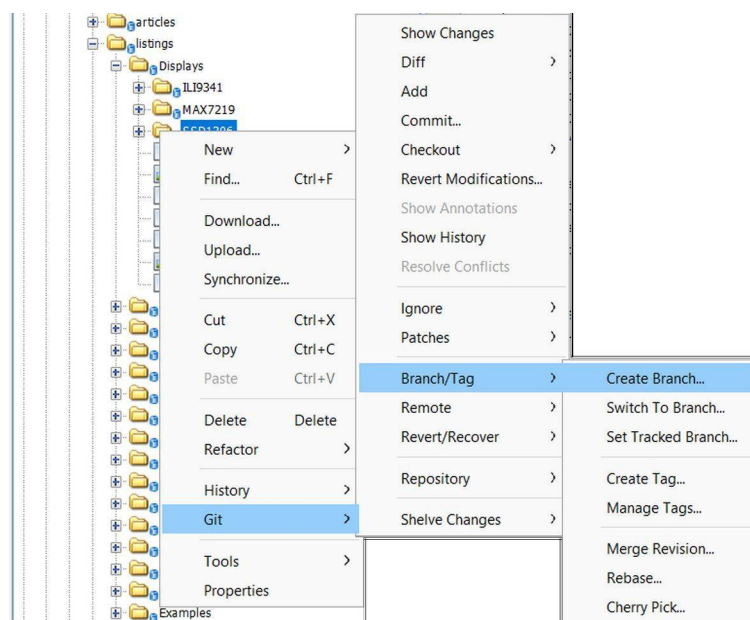


Figure 14: création d'une branche sur un projet

Exemple de situation qui justifie la création d'une branche :

- vous avez un projet fonctionnel ;
- vous envisagez de l'optimiser ;
- créez une branche et faites les optimisations dans cette branche...

Les modifications de fichiers sources dans une branche n'ont pas d'influence sur les fichiers dans le tronc *main*.

Accessoirement, il est plus que conseillé de disposer d'un support physique de sauvegarde. Un disque dur SSD c'est environ 50€ pour 300Gb d'espace de stockage. La vitesse d'accès en lecture ou écriture d'un support SSD est tout simplement bluffante !

## Stockage sur GitHub

Le site web **GitHub**<sup>2</sup> est, avec **SourceForge**<sup>3</sup>, un des meilleurs endroits pour stocker ses fichiers sources. Sur GitHub, vous pouvez mettre un dossier de travail en commun avec d'autres développeurs et gérer des projets complexes. L'éditeur Netbeans peut se connecter au projet et vous permet de transmettre ou récupérer des modifications de fichiers.

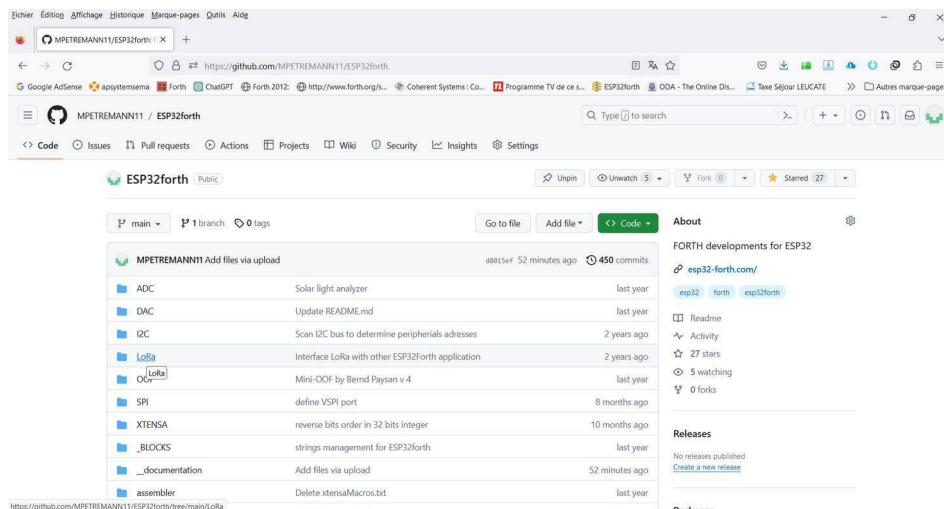


Figure 15: stockage des fichiers sur Github

Sur **GitHub**, vous pouvez gérer des embranchements de projets (*fork*). Vous pouvez aussi rendre confidentiel certaines parties de vos projets. Ici les branches dans les projets de flagxor/ueforth :

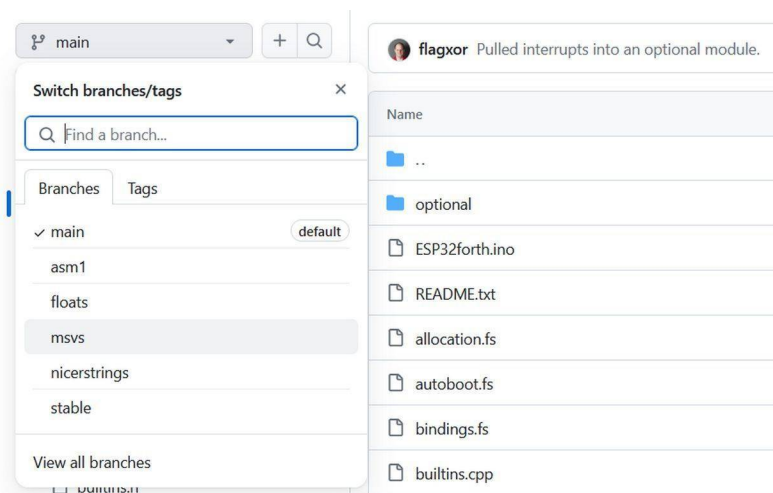


Figure 16: accès à une branche de projet

## Quelques bonnes pratiques

La première bonne pratique consiste à bien nommer ses fichiers et dossiers de travail. Vous développez pour Z79Forth, donc créez un dossier nommé **Z79Forth**.

<sup>2</sup> <https://github.com/>

<sup>3</sup> <https://sourceforge.net/>

Pour les essais divers, créez dans ce dossier un sous-dossier **sandbox** (bac à sable).

Pour les projets bien construits, créez un dossier par projet. Par exemple, vous programmez un jeu, créez un sous-dossier **game**.

Si vous avez des scripts d'usage général, créez un dossier **tools**. Si vous utilisez un fichier de ce dossier **tools** dans un projet, copiez et collez ce fichier dans le dossier de ce projet. Ceci évitera qu'une modification d'un fichier dans **tools** ne perturbe ensuite votre projet.

La seconde bonne pratique consiste à répartir le code source d'un projet dans plusieurs fichiers :

- **config.4th** pour stocker les paramètres du projet ;
- répertoire **documentation** pour stocker des fichiers dans le format de votre choix, en rapport avec la documentation du projet ;
- **myApp.4th** pour les définitions de votre projet. Choisissez un nom de fichier assez explicite . Par exemple, pour gérer un jeu de labyrinthe, prenez le nom **game-maze.4th**.

Une fois le projet dans son état final, l'idéal est de créer les blocs dans le système de gestion des blocs de Z79Forth, permettant une compilation quasi-instantanée du code de l'application.



## Utiliser les nombres avec Z79Forth

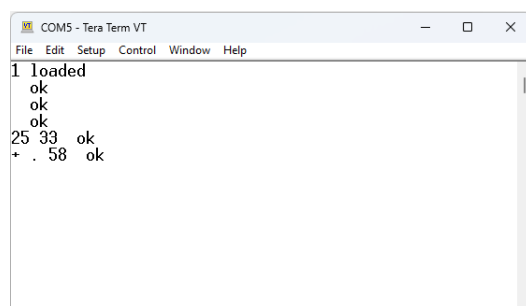
Nous avons démarré Z79Forth sans souci. Nous allons maintenant approfondir quelques manipulations sur les nombres pour comprendre comment maîtriser 79Forth.

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 17: 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
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 \ affiche:
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 \ affiche:
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 \ est plus lisible que 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 \ affiche:
OoR error (0000/0000)
F58F >NUMBER+0081

```

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

```

decimal
$ffff . \ affiche: -1

```

Je vous vois 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. \ affiche: 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 !
0000000000000001 \ met 1 sur la pile
1111111111111111 \ met -1 sur la pile
```

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 :

```
0000000000000001
1111111111111111
      10
```

Etape suivante :

```
0000000000000001
1111111111111111
      10
     100
```

Arrivé à la fin, on aura comme résultat :

```
0000000000000001
1111111111111111
1000000000000000
```

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 @ .      \ affiche: 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 @ .      \ affiche: 76C
score @
$00FF and .    \ affiche: 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 décimal
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'IDE<sup>4</sup> 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 ;

---

4 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 :

- si le terminal ne gère pas les caractères UNICODE, Z79Forth n'accepte pas les noms écrits dans des alphabets différents de l'alphabet latin.
- Si le terminal gère les caractères UNICODE, vous pourrez utiliser tous les alphabets possibles.

Vous pouvez utiliser ces alphabets pour les commentaires :

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

OU

```
: .date      \ 海报今天的日期
...code...  ;
```

Les caractères provenant d'un codage linguistiques et utilisés dans les commentaires ne sont utilisables que dans votre code source.

## Utiliser des caractères UNICODE dans les noms

Le langage FORTH a ceci de fascinant, c'est un langage informatique qui peut rapidement diverger de la norme établie dans d'autres langages de programmation.

Dans tous les langages de programmation, les noms de fonction ne peuvent être écrits qu'en caractères latin et limités au jeu ASCII-US. Seul le langage HTML et son extension CSS peuvent déroger en acceptant des noms de classes dans d'autres alphabet. Exemple de classe CSS :

```
.signature {
  margin-top: 16px;
  margin-bottom: 50px;
  text-align: right;
}
```

Il est tout à fait possible d'écrire **.signature** en russe :

```
.сигнатура {
  margin-top: 16px;
  margin-bottom: 50px;
  text-align: right;
}
```

```
}
```

De même, avec Z79Forth, on peut définir un mot avec cet alphabet :

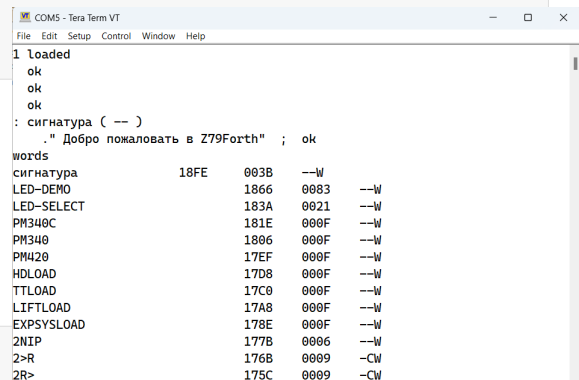
```
: сигнатура ( -- )  
  ." Добро пожаловать в Z79Forth" ;
```

Exécutons **words** :

Le mot **сигнатура** apparaît bien dans le dictionnaire FORTH.

ET on peut exécuter ce mot

```
сигнатура \ affiche :  
Добро пожаловать в Z79Forth ok
```



Ca fonctionne aussi bien avec des caractères chinois, arabes, hébreux, hindi, etc. Mais n'abusez pas de cette facilité !

## Indentation du code source

Que le code soit écrit 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 le 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 deux octets. Ce sont donc des entiers au format 16 bits. Cependant, certains mots traitent des données au format huit 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 myDelimiter
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 myDelimiter
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. Équivaut 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
;

```

## Le décompilateur

Dans un compilateur conventionnel, le code source est transformé en code objet contenant les adresses de référence à une bibliothèque équipant le compilateur. Pour disposer d'un code exécutable, il faut linker le code objet. A aucun moment le programmeur ne peut avoir accès au code exécutable contenu dans ses bibliothèque avec les seules ressources du compilateur.

Avec Z79Forth, le développeur peut décompiler ses routine, mais également visualiser les primitives prédéfinies donc comprendre le fonctionnement de son programme ou de FORTH dans son intégralité.

Pour voir la définition d'un mot, il suffit de taper **SEE** suivi du mot à décompiler. Exemple de compilation :

```

: C>F ( 0C --- 0F) \ Conversion Celsius in Fahrenheit
9 5 */ 32 +
;
see c>f
\ display:
1AF6 8E0009 lxi ???
1AF9 BDE7DB jsr NPUSH
1AFC 8E0005 lxi ???
1AFF BDE7DB jsr NPUSH
1B02 BDFA70 jsr */
1B05 8E0020 lxi ???
1B08 BDE7DB jsr NPUSH
1B0B 7EF899 jmp + ok

```

Testons **SEE** sur un mot prédéfini du vocabulaire:

```
see dup \ display :  
FC85 BDE9D7      jsr      CKDPTRA  
FC88 AE          fcb      $AE  
FC89 C4          fcb      $C4  
FC8A 7EE7DB      jmp      NPUSH ok
```

Ici, nous venons de décompiler **dup**. On constate que **dup** est écrit en langage assembleur...

# 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
```

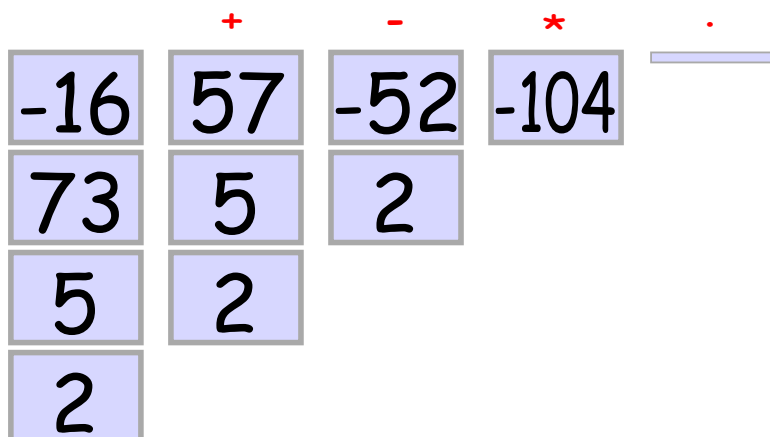
laisse la pile de paramètres dans l'état

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

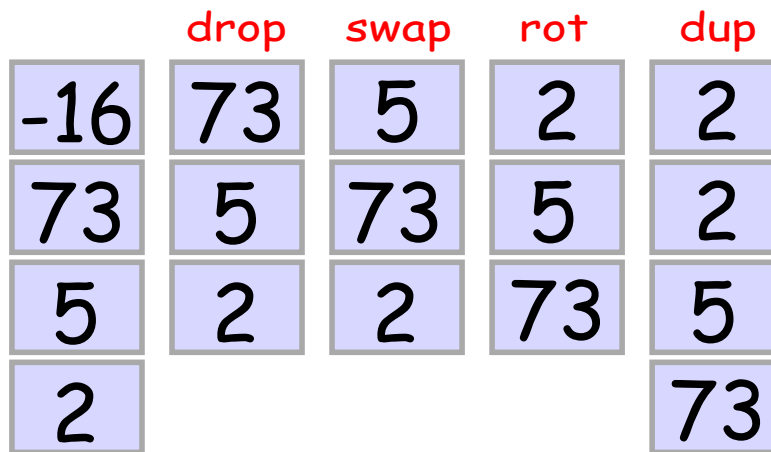


Figure 18: manipulation de la pile de données

actions sont présentées ci-dessous.

## La pile de retour et ses utilisations

L'utilisateur peut stocker et récupérer des données à partir de la pile de retour. 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 :

```
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 @ . \ affiche 21
5 temperatures @ . \ affiche 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 ?

# Marquage des couches fonctionnelles avec **MARKER**

Z79Forth ne dispose pas du mot **FORGET**.

Le mot **FORGET** figure pourtant dès l'origine du langage FORTH dans quasiment tous les standards: 79-Standard, 83-Standard...

Alors pourquoi ne figure-t-il pas dans Z79Forth ?

En fait, le standard le plus récent, forth200x, a déclaré le mot **FORGET** comme obsolète. Il peut encore rester dans des versions récentes de FORTH comme une concession pour les implémentations existantes aux anciens standards.

## Fonctionnement du mot **FORGET**

Voici comme s'utilise le mot **FORGET** sur les versions du langage FORTH incluant ce mot.

Reprenons une suite de définitions :

```
: >gray ( n -- n' )
  dup 2/ xor ;    \ n' = n xor ( décalage logique vers la droite 1x )

: gray> ( n -- n )
  0 1 16 lshift ( -- g b mask )
  begin
    >r          \ sauve mask sur pile retour
    2dup 2/ xor
    r@ and or
    r> 1 rshift
    dup 0=
  until
  drop nip ;    \ nettoie pile de données en laissant le résultat

: test
  2 base !      \ sélectionne base binaire
  32 0 do
    cr I dup 5 .r ." ==> " \ affiche valeurs (binaire)
    >gray dup 5 .r ." ==> " \ justifiés à droite sur 5 caractères
    gray>      5 .r
  loop
  decimal ;    \ remet en décimal
```

Comment supprimer le mot **test** ?

Avec **FORGET**, il suffit de taper:

```
FORGET test
```



Si ensuite on tape **WORDS**, on ne retrouvera que les définitions **>gray** et **gray>**.

Le mot **FORGET** permet donc de supprimer une partie des définitions :

```
: w1 ; ok
: w2 ; ok
: w3 ; ok
: w4 ; ok
: w5 ; ok
words
w5 w4 w3 w2 w1 .....
forget w3
words
w2 w1 .....
```

La séquence **forget w3** retire du dictionnaire le mot **w3** et tous les mots définis après lui, c'est à dire **w4** et **w5**.

Si le mot **FORGET** est commode en phase de développement, il présente cependant un risque de confusion en provoquant une suppression accidentelle sur des zones de dictionnaire sensibles. Un mot d'exécution vectorisée peut se voir détruire son vecteur. L'exécution de ce mot pourra ensuite perturber profondément FORTH.

Quand on développe en langage FORTH, on conçoit avant tout des ensembles de mots par couches fonctionnelles. Ce sera par exemple:

- une couche de gestion d'affichage pour un jeu ;
- une couche applicative décrivant le jeu et utilisant la couche précédente ;
- une couche finale verrouillant l'ensemble des couches applicatives.

Il existe un moyen élégant pour marquer ces différentes couches à l'aide du mot **marker**.

## Marquage avec le mot marker

Forth permet d'oublier les mots et tout ce qui a été attribué dans le dictionnaire après ceux-ci. Il faut utiliser un marqueur créé avec **marker**:

```
-gray
marker -gray
: >gray ( n -- n' )
  dup 2/ xor ;
: gray> ( n -- n )
  0 1 31 lshift ( -- g b mask )
  begin
    >r 2dup 2/ xor r@ and or r> 1 rshift dup 0=
  until
  drop nip ;
```

On retrouve dans le dictionnaire ces mots :

```
words
gray> >gray -gray
```

Il peut paraître surprenant d'invoquer ce mot **-gray** avant qu'il ne soit défini. D'ailleurs, à la première compilation, ce mot provoque un message d'erreur mais n'empêche pas la compilation :

```
-gray
'-gray' ? (0000/0000)
E44E NUMCVRA
E0D0 NMCVIRA
E0BD MINTLRA
marker -gray    ok
\ ...etc.....
```

On peut taper manuellement **-gray** et tous les mots définis après **-gray** seront supprimés du dictionnaire, **-gray** inclus.

Mais c'est encore plus intéressant quand on recompile notre code. Car le premier mot trouvé par l'interpréteur FORTH sera **-gray**:

```
-gray    ok
marker -gray    ok
```

A la première compilation on avait **'-gray' ? (0000/0000)**.

A la compilation suivante, on a **-gray ok**.

On peut recompiler autant de fois que nécessaire notre code FORTH. A chaque fois que l'interpréteur rencontrera le marqueur **-gray**, l'espace mémoire pris par tous les mots précédemment définis après **-gray** seront supprimés.

Cet espace libéré sera réutilisé par les nouvelles définitions.

Nous avons ici un avantage considérable par rapport à d'autres langages de programmation : la capacité à gérer les définitions compilées par couches fonctionnelles !

Chaque couche pourra être compilée après une autre couche logicielle, elle même suivie d'autres couches logicielles.

Quand le développement de l'application est finalisée, on peut supprimer toutes les lignes de code exploitant **marker**.

# La structure du dictionnaire FORTH

Quand vous achetez n'importe quel appareil, vous l'utilisez. Le plus souvent, vous ne cherchez pas à comprendre son fonctionnement. D'ailleurs, c'est souvent impossible, car montés par assemblage collés, soudés, vissés. Toute tentative de démontage risquerai même de l'endommager.

Avec FORTH, c'est exactement le contraire. Il n'y a pas d'un coté le code de l'utilisateur, de l'autre coté un obscur compilateur à utiliser sans autre explication.

FORTH offre des outils de compilation, mais aussi les moyens pour agir sur l'interprétation ou la compilation des définitions. Ici, nous allons approfondir en décrivant la structure du dictionnaire FORTH.

## Accès au champ du code exécutable

Le champ du code exécutable est l'élément de départ pour comprendre la structure du dictionnaire FORTH. Pour obtenir l'adresse mémoire de ce champ, on utilise le mot `'` (tick) :

```
' page u.      \ affiche: 63303
```

Le mot **PAGE** efface le contenu affiché. Pour exécuter ce mot, il y a deux possibilités :

- taper **PAGE** et appui sur la touche *Entrée* du clavier ;
- taper **63303 EXECUTE**

La seconde solution est réservée à des utilisations très spéciales. Reste une question essentielle : « Comment l'interpréteur FORTH fait-il pour retrouver **PAGE** dans le dictionnaire ? ».

La recherche d'un mot dans le dictionnaire est un mécanisme exploitant des champs, dont voici le résumé :

- **pf** (pour parameter field) qui est le champ des paramètres ;
- **cf** (pour code field) qui est le champ du code exécutable, celui qui est empilé lors de l'exécution du mot `'` (tick) ;
- **lf** (pour link field) qui est le champ de lien entre les mots ;
- **nf** (pour name field) qui est le champ du nom des mots.

Le champ **cf**, champ du code exécutable) est un champ essentiel pour récupérer certaines adresses, en particulier pour accéder au champ de données d'un mot défini par **CREATE** :

```
create score
```

```
hex
' score u.          \ affiche: 18F1
score u.            \ affiche: 18FA
' score >body u.    \ affiche: 18FA
```

## Structure d'un mot FORTH

Voici la structure d'un mot FORTH dans Z79Forth :

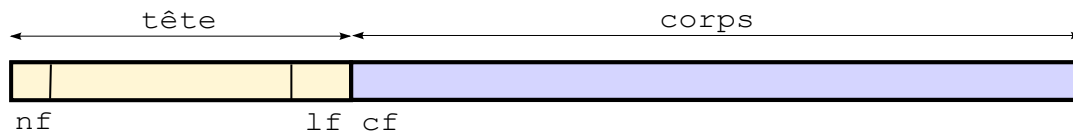


Figure 19: stucture d'un mot FORTH

- la **tête** est la partie du mot permettant de gérer ce mot dans le dictionnaire ;
- le **corps** contient le code exécutable, les données dans le cas de certaines variables, les données des mots définis par **CREATE**.

L'utilisation du mot **'** (tick) empile l'adresse **xt**, c'est à dire la même adresse que le champ **cf** dans le schéma ci-dessus.

ATTENTION : avec Z79Forth, le champ **cf** contient toujours du code machine. Voir le chapitre *Utiliser du code binaire*. Il est fortement déconseillé de modifier le contenu de ce champ dans vos définitions compilées, au risque de perturber gravement le fonctionnement du programme.

Revenons sur notre mot **SCORE**. Nous savons comment récupérer l'adresse **xt**. Rappel :

```
create score
hex
' score u.          \ affiche: 18F1
```

Pour passer du champ **nf** au champ **lf**, il suffit de décrémenter de deux unités la valeur **xt** :

```
' score 1- 1- u.    \ affiche: 18EF
```

Mais à quoi correspond le contenu de ce champ **lf** ? L'adresse contenue dans ce champ **lf** pointe vers le champ **nf** du dernier mot du dictionnaire défini avant la définition de notre mot **SCORE**.

## Le champ de nom

Le premier octet du champ du nom contient le nombre de caractères constituant le nom du mot. Exemple :

```
' dup 10 - 20 dump    \ affiche :
    00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0123456789ABCDEF
```

```

FC70      03 4E 49 50 FC 6A 8D 21 20 15
03      .NIP.j.! ..
FC80 44 55 50 FC 75 BD E9 D7 AE C4 7E E7 DB 04 44 52
DUP.u.....~...DR
FC90 4F 50 FC 7F 7E

```

On retrouve ici la succession des octets **03 44 55 50**. Le premier octet, ici **03**, correspond à la longueur du nom, ici **DUP**.

Pour afficher un nom, il suffit de partir de l'adresse de ce champ **nf** :

```

FC7F count type      \ affiche: DUP

```

Le premier octet du champ **nf** est également utilisé par divers indicateurs, seul les 5 bits de poids faible nous intéressent :

```

\ get string for the name of FORTH word from nf address
: name>string ( nf -- addr len )
    count $1F and
;

```

Maintenant, on peut définir le mot qui va nous permettre de passer du champ **nf** au champ **lf** :

```

\ get lf address from nf address
: n>link ( nf -- lf )
    name>string +
;

```

On peut également définir le mot qui va afficher le nom d'un mot FORTH à partir de l'adresse de son champ **nf** :

```

\ display name of a word
: .name ( nf -- )
    name>string      \ get addr len from nf field
    type space      \ display name + space
;

```

Avec ces premières définitions, nous allons maintenant voir comment est organisé le dictionnaire FORTH.

## Le champ de lien

Pour comprendre le rôle de ce champ de lien, commençons par un mot essentiel, le mot **LAST**. Ce mot pointe vers le champ **nf** du dernier mot défini.

```

: display.card ;
last .name      \ affiche: DISPLAY.CARD
: init.card ;
last .name      \ affiche: INIT.CARD

```

Voyons ce que contient le champ **lf** de **init.card** :

```
last n>link @ .name \ affiche: DISPLAY.CARD
```

Le champ **lf** de **init.card** pointe vers le champ **nf** de **display.card**. Dans le dictionnaire FORTH, tous les mots sont liés de cette manière :

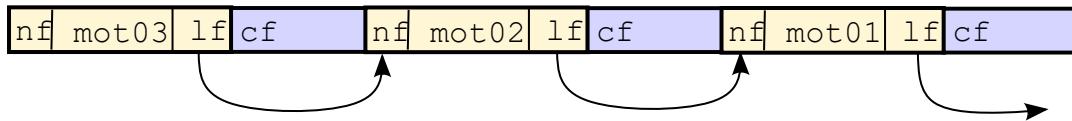


Figure 20: liens entre les mots FORTH

Les mots FORTH sont enregistrés séquentiellement dans le dictionnaire. Le dernier mot défini apparaît en premier. Chaque mot est lié au mot précédemment défini par son champ **lf** pointant vers le champ **nf** du mot qui le précède dans le dictionnaire.

Avec cette information, on peut envisager la définition du mot **vlist** :

```
\ display content of current vocabulary
: vlist ( -- )
  cr
  last >r
  begin
    r@ .name
    r@ n>link @
  while
    r> n>link @ >r
  repeat
    r> drop
  ;
vlist \ display:
INIT.CARD DISPLAY.CARD VLIST .NAME #OUT+! MAX-OUT #OUT N>LINK NAME>STRING
LED-DEMO LED-SELECT PM340C PM340 PM420 HDLOAD TTLOAD LIFTLOAD EXPSYSLOAD
2NIP 2>R 2R> 2R@ RSHIFT LSHIFT ACTION-OF IS DEFER@ DEFER! DEFER TO VALUE
WITHIN >BODY .( /STRING ABORT" UNUSED ENVIRONMENT? ERASE TOUPPER COMPARE
(-TEXT) BOUNDS (SGN) -TRAILING ANSILOAD INDEX LINE : H. BSA BS BFBSADR...
```

Le dictionnaire est exploré, mot par mot, en sautant successivement des adresses **nf** vers **lf**, en commençant par le dernier mot défini dans le dictionnaire. Arrivé au premier mot du dictionnaire, le champ **lf** contient la valeur **\$0000**. C'est l'indication qu'il n'y a plus d'autre mot à explorer.

Cette structure de dictionnaire est désignée sous l'acronyme **TIL** (Threaded Interpretive Languages). Programmer en FORTH consiste simplement à agrandir le dictionnaire. Plus il y a de mots, plus la recherche d'un mot sera longue. Cette lenteur, toute relative, n'est qu'apparente, car le programme final exploite seulement les parties de code stockées dans les champs **cf**. Et comme Z79Forth stocke les codes exécutables en langage machine, les performances sont maximales.

Les primitives de base de la version Z79Forth tiennent dans une espace mémoire de 8Ko. Hormis FORTH, il n'existe aucun compilateur aussi petit et performant. Et cette compacité

persiste dans les codes compilés. Place occupée en mémoire RAM par notre exemple **vlist** :

```
here u.          \ affiche: 6377
( ...on compile vlist )
here u.          \ affiche: 6605
6605 6377 - .    \ affiche: 228
```

L'espace mémoire RAM par occupé par l'ensemble des mots pour **vlist** est de 228 octets ! Les lenteurs de compilations proviennent essentiellement du code source transmis à Z79Forth via le terminal. Sur un PC et une version comme eForth Windows, des programmes sources en langage FORTH, cent fois plus volumineux, sont compilés quasi instantanément !

## Recherche d'un mot dans le dictionnaire

Que ce soit depuis l'interpréteur ou en compilation, la recherche des mots dans le dictionnaire FORTH est automatique. Seule contrainte, pour utiliser une définition dans un nouveau mot, il faut que cette définition soit dans le dictionnaire.

Mais qu'en est-il si accidentellement (ou volontairement) on compile plusieurs fois un même mot ?

```
: myWord ( -- )
  ." First word test" ;
: myWord ( -- )
  ." Second word test" ;
: .word1 myWord ;
```

Ici, on compile deux fois un mot nommé **myWord** :

```
.word1          \ affiche: Second word test
```

Le mot **.word1** exploite la définition la plus récente dans l'ordre de recherche du mot **myWord** dans le dictionnaire. Exécution de **vlist** :

```
.WORD1 MYWORD MYWORD VLIST .NAME #OUT+! MAX-OUT #OUT N>LINK NAME>STRING
LED-DEMO LED-SELECT PM340C PM340 PM420 HDLOAD TTLOAD LIFTLOAD EXPSYSLOAD
```

Ici, en rouge, la version la plus récente de myWord compilée par .word1.

La version plus ancienne, ici en bleu, du mot **myWord** ne sera plus exploitable. Cette version est dans le dictionnaire, mais on ne peut plus l'exécuter ou la compiler ! Seul un mécanisme d'exploration comme celui programmé dans **vlist** arrivera encore à le retrouver.

Rajoutons ces définitions :

```
: myWord ( -- )
  ." Third word test" ;
: .word1 myWord ;
```

Exécutions **vlist** qui va nous restituer ceci :

```
.WORD1 MYWORD .WORD1 MYWORD MYWORD VLIST .NAME #OUT+! MAX-OUT #OUT N>LINK
NAME>STRING LED-DEMO LED-SELECT PM340C PM340 PM420 HDLOAD TTLOAD LIFTLOAD
```

Exécution de **.word1** :

```
.word1      \ affiche: Third word test
```

C'est la dernière version du mot **.word1** qui est exécuté, c'est à dire celle utilisant la version la plus récente de **myWord**, ici en rouge dans les premières lignes de **vlist**.

Modifions légèrement l'ensemble du code :

```
: myWord  ( -- )
  ." First word test"  ;
: myWord  ( -- )
  ." Second word test"  ;
: .word1 myWord ;
: myWord  ( -- )
  ." Third word test"  ;
: .word2 myWord ;
.word1      \ affiche: Second word test
.word2      \ affiche: Third word test
```

On constate que la redéfinition d'un mot ne concerne que les mots qui seront compilés après cette redéfinition.

En programmation FORTH, il faut éviter ce cas de figure. Les risques :

- avoir un comportement inattendu du programme final ;
- générer des programmes ingérables ;

La manière la plus radicale, pour éviter ces redéfinitions, consiste à utiliser des noms de mots préfixés, à la manière de **javaScript** :

```
: test.myWord  ( -- )
  ." First word test"  ;
: prod.myWord  ( -- )
  ." Second word test"  ;
: .word1 test.myWord ;
: proto.myWord  ( -- )
  ." Third word test"  ;
: .word2 prod.myWord ;
.word1      \ affiche: First word test
.word2      \ affiche: Second word test
```

Lors de la compilation des définitions, surveillez la compilation. Toute redéfinition génère un message d'alerte :

```
WARNING: .word1 is being redefined
```



Ce message est un simple avertissement. Il ne bloque pas la compilation. Si ce message apparaît lors de la compilation finale d'un programme, ce peut être source de potentiels problèmes d'exécution de ce programme final.

# Structures de données pour Z79Forth

## Préambule

Z79Forth est une version 16 bits du langage FORTH. Cette taille de données est déterminée par la taille des éléments déposés sur la pile de données. Pour connaître la taille en octets des éléments, il faut exécuter le mot **cells**. Exécution de ce mot pour Z79Forth :

```
1 cells . \ affiche 2
```

La valeur 2 signifie que la taille des éléments déposés sur la pile de données est de 2 octets, soit  $2 \times 8 \text{ bits} = 16 \text{ bits}$ .

Avec une version FORTH 32 bits, **cells** empilera la valeur 4. De même, si vous utilisez une version 64 bits, **cells** empilera la valeur 8.

## Les tableaux en FORTH

Commençons par des structures assez simples : les tableaux. Nous n'aborderons que les tableaux à une ou deux dimensions.

### Tableau de données 16 bits à une dimension

C'est le type de tableau le plus simple. Pour créer un tableau de ce type, on utilise le mot **create** suivi du nom du tableau à créer:

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

Dans ce tableau, on stocke 6 valeurs: 34, 37....12. Pour récupérer une valeur, il suffit d'utiliser le mot **@** en incrémentant l'adresse empilée par **temperatures** avec le décalage souhaité:

```
temperatures    \ empile addr
  0 cells        \ calcule décalage 0
  +              \ ajout décalage à addr
  @ .            \ affiche 34

temperatures    \ empile addr
  1 cell *       \ calcule décalage 1
  +              \ ajout décalage à addr
  @ .            \ affiche 37
```

On peut factoriser le code d'accès à la valeur souhaitée en définissant un mot qui va calculer cette adresse:

```

: temp@ ( index -- value )
    cells temperatures + @
;
0 temp@ . \ affiche 34
2 temp@ . \ affiche 42

```

Vous noterez que pour n valeurs stockées dans ce tableau, ici 6 valeurs, l'index d'accès doit toujours être dans l'intervalle [0..n-1].

## Mots de définition de tableaux

Voici comment créer un mot de définition de tableaux d'entiers à une dimension :

```

: array ( comp: -- | exec: index -- addr )
    create
    does>
        swap cells +
;
array myTemps
    21 ,    32 ,    45 ,    44 ,    28 ,    12 ,
0 myTemps @ . \ affiche 21
5 myTemps @ . \ affiche 12

```

Dans notre exemple, nous stockons 6 valeurs comprises entre 0 et 255. Il est aisé de créer une variante de **array** pour gérer nos données de manière plus compacte:

```

: arrayC ( comp: -- | exec: index -- addr )
    create
    does>
        +
;
arrayC myCTemps
    21 c,    32 c,    45 c,    44 c,    28 c,    12 c,
0 myCTemps c@ . \ display 21
5 myCTemps c@ . \ display 12

```

Avec cette variante, on stocke les mêmes valeurs dans deux fois moins d'espace mémoire qu'en utilisant les cellules sur 16 bits (2 octets).

## Lire et écrire dans un tableau

Il est tout à fait possible de créer un tableau vide de n éléments et d'écrire et lire des valeurs dans ce tableau:

```

arrayC myCTemps
    6 allot \ réserve 6 octets
    0 myCTemps 6 0 fill \ remplis ces 6 octets avec valeur 0
32 0 myCTemps c! \ stocke 32 dans myCTemps[0]
25 5 myCTemps c! \ stocke 25 dans myCTemps[5]
0 myCTemps c@ . \ affiche 32

```

Dans notre exemple, le tableau contient 6 éléments.

Il est facile de créer des tableaux à plusieurs dimensions. Exemple de tableau à deux dimensions:

```
63 constant SCR_WIDTH
16 constant SCR_HEIGHT
create mySCREEN
  SCR_WIDTH SCR_HEIGHT * allot          \ réserve 63 * 16 octets
  mySCREEN SCR_WIDTH SCR_HEIGHT * b1 fill \ remplis cet espace avec
  'space'
```

Ici, on définit un tableau à deux dimensions nommé **mySCREEN** qui sera un écran virtuel de 16 lignes et 63 colonnes.

Il suffit de réserver un espace mémoire qui soit le produit des dimensions X et Y du tableau à utiliser. Voyons maintenant comment gérer ce tableau à deux dimensions :

```
: xySCRaddr ( x y -- addr )
  SCR_WIDTH * +
  mySCREEN +
;
: SCR@ ( x y -- c )
  xySCRaddr c@
;
: SCR! ( c x y -- )
  xySCRaddr c!
;
char X 15 5 SCR!    \ stocke char X à col 15 ligne 5
15 5 SCR@ emit      \ affiche X
```

## Gestion de structures complexes

Le langage C dispose d'instructions permettant de définir des structures complexes. Ces instructions n'existent pas dans Z79Forth.

### Définition de struct et field

Dans les précédents exemples, nous avons vu comment définir des tableaux de données à une ou deux dimensions. Reprenons le cas d'un tableau à deux dimensions :

```
5 constant spriteWidth
7 constant spriteHeight
create mySprite
  spriteWidth spriteHeight * allot
```

Si je veux définir un sprite avec d'autres dimensions, je dois définir deux autres constantes, spécifiques à ce nouveau sprite.

L'idée serait d'embarquer les dimensions de chaque sprite dans les données de ce sprite :

```
create mySprite
```

```
5 , 7 ,
5 7 * allot
```

Les deux premières cellules contiennent respectivement la largeur et la hauteur du sprite. Pour accéder aux données de notre sprite, définissons trois accesseurs :

```
: ->sprite.width  ( addr - addr' )
    0 + ;
: ->sprite.height  ( addr - addr' )
    2 + ;
: ->sprite.content  ( addr - addr' )
    4 + ;
```

Pour accéder à la première adresse des données de notre sprite :

```
mySprite ->sprite.content
```

La première idée venant à l'esprit sera de factoriser la création de ces accesseurs au travers d'une définition générique **field** :

```
int field ->sprite.width
int field ->sprite.height
int field ->sprite.content
```

Le mot **int** est une sorte de constante définie comme suit :

```
: typer ( len -- )
    constant
;

1 typer char
1 typer byte
2 typer int
4 typer double
1 typer i8
2 typer i16
4 typer i32
```

L'idée, en utilisant ces constantes, sera d'incrémenter un compteur à chaque déclaration d'un nouveau champ avec **field**. C'est ce qu'on va réaliser dans la définition de **field** :

```
\ Define a field in data structure
: field ( comp: c -- | exec: addr -- addr' )
    create
        dup
        last-struct @ ,      \ get and compile latest position in structure
        last-struct +!      \ increment latest structure
    does>
        @ +                  \ get real position of data in structure
;
```

La valeur **last-struct** pointe vers la dernière structure définie. Cette structure mémorisera chaque nouvelle position d'un champ dans cette structure. Une structure sera initiée par une sorte de variable :

```
\ store parameter address of latest defined structure
0 value last-struct

\ Define a new structure
: struct ( comp: -- <name> | -- n )
  create
    0 , \ initial value
    last count + 2 +
    >body to last-struct \ store param. addr in last-struct
  does>
    @
;
;
```

**Information** : la syntaxe et l'utilisation des mots **struct**, **field**, **typer** sont inspirées de celles définies dans les versions eForth et ESP32Forth créées par Brad NELSON.

Le code source complet de ces mots adaptés à Z79Forth est disponible ici :

<https://github.com/MPETREMANN11/Z79Forth/blob/master/SW/MP%20extensions/structures.4th>

## Exemples de structures

Pour définir une structure sprite :

```
struct sprite
  int field ->sprite.width
  int field ->sprite.height
  int field ->sprite.content

create bigSmiley
  5 , 7 , 5 7 * allot

bigSmiley ->sprite.width @ . \ affiche largeur du sprite
bigSmiley ->sprite.height @ \ affiche hauteur du sprite
```

Voici un autre exemple trivial de structure:

```
struct YMDHMS
  int field ->year
  byte field ->month
  byte field ->day
  byte field ->hour
  byte field ->min
  byte field ->sec
```

Ici, on définit la structure **YMDHMS**. Cette structure gère les accesseurs **->year ->month ->day ->hour ->min** et **->sec**.

Le mot **YMDHMS** a comme seule utilité d'initialiser et associer les accesseurs à la structure complexe. Voici comment sont utilisés ces accesseurs :

```
create DateTime
  YMDHMS allot

2022 DateTime ->year  !
  03 DateTime ->month c!
  21 DateTime ->day   c!
  22 DateTime ->hour  c!
  36 DateTime ->min   c!
  15 DateTime ->sec   c!

: .date ( date -- )
  >r
  ." YEAR: " r@ ->year    @ . cr
  ." MONTH: " r@ ->month c@ . cr
  ." DAY: " r@ ->day     c@ . cr
  ." HH: " r@ ->hour    c@ . cr
  ." MM: " r@ ->min     c@ . cr
  ." SS: " r@ ->sec     c@ . cr
  r> drop
;

DateTime .date
```

Le mot **DateTime** est défini comme un tableau. L'accès à chaque champ de ce tableau est réalisé par l'intermédiaire de son accesseur.

Dans la structure **YMDHMS**, l'année est au format 16 bits, toutes les autres champs sont réduits à des entiers 8 bits. Dans le code de **.date**, l'utilisation des accesseurs permet un accès facile à chaque élément de notre structure complexe.

# Les caractères Unicode avec emit et key

En préambule, vous avez installé un terminal compatible VT.

Sous Windows, je préconise le terminal **TeraTerm**, compatible VT100 à VT525.

**TeraTerm** n'émule pas complètement toutes les fonctionnalités des terminaux VT, mais pour ce qui nous intéresse, nous allons analyser comment gérer les caractères Unicode.

Commençons par les caractères ASCII affichables, ceux dont le code est compris entre 32 et 127. Voici comment afficher la table des caractères disponibles sur 7 bits :

```
: tableChars ( -- )
  cr
  base @ >r hex
  128 32 do
    16 0 do
      j i + dup . space emit space space
    loop
  cr
  16 +loop
;
```

tableChars

```
tableChars
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  ¨
```

Voici le résultat de l'exécution de **tableChars**:

Si nous tentons d'afficher un caractère avec un code supérieur à 127, il y aura une erreur d'affichage.

```
132 emit \ affiche ✖
215 emit \ affiche ✖
```

## Le codage Unicode

Unicode est un système de codage standardisé qui attribue un numéro unique à chaque caractère, permettant ainsi aux ordinateurs de les reconnaître et de les afficher correctement, quelle que soit la langue ou la plateforme utilisée.

Chaque caractère Unicode est associé à un point de code, un nombre qui l'identifie de manière unique. Ce point de code est ensuite converti en une séquence d'octets pour être stocké dans un ordinateur.



Il existe différentes encodages Unicode, les plus courants étant UTF-8 et UTF-16. Ces encodages déterminent la manière dont les points de code sont convertis en octets.

**UTF-8** est l'encodage le plus utilisé sur Internet. Il est compatible avec les anciens systèmes qui ne supportent pas l'Unicode et est très efficace pour les textes en anglais et dans les langues européennes occidentales.

Chaque caractère Unicode est représenté par une séquence d'un à quatre octets. Le nombre d'octets nécessaire dépend de la valeur du point de code Unicode du caractère. Les premiers bits de chaque octet d'une séquence indiquent le nombre total d'octets de la séquence. Cela permet au décodeur de savoir immédiatement combien d'octets il doit lire pour reconstituer le caractère.

- **octet initial** : commence toujours par une séquence de bits '110' pour une séquence de 2 octets, '1110' pour une séquence de 3 octets, ou '11110' pour une séquence de 4 octets. Les bits suivants contiennent une partie du point de code Unicode.
- **octets suivants** : commencent toujours par la séquence de bits '10'. Les bits suivants contiennent également une partie du point de code Unicode.

Prenons le caractère "é" (accent aigu). Son point de code Unicode est U+00E9. Il sera codé sur deux octets en UTF-8 de la manière suivante :

- **premier octet**: **11000011** (le '110' indique une séquence de 2 octets), \$C3 en hexadécimal
- **deuxième octet**: **10111001**, \$A9 en hexadécimal

```
$c3 emit $a7 emit \ affiche é
```

## Récupérer un caractère Unicode depuis le terminal

Un caractère Unicode UTF8 ne peut pas être récupéré simplement par le mot **key**. L'exécution de **key** ne fonctionne que pour les caractères dont le code ASCII est compris entre 32 et 127. Pour récupérer les valeurs constituant un caractère Unicode dont le code est supérieur à 127, il faut exécuter **key** plusieurs fois, typiquement de 2 à 4 fois :

```
hex key key . . \ press 'é' on keyboard, display A9 C3
```

Voici la définition du mot **ukey** qui va tester le caractère saisi au terminal et empiler de un à quatre octets correspondant à la séquence Unicode du caractère

```
\ get an UNICODE character
: ukey ( -- )
  key dup 1 lshift >r
  begin
    r@ $80 and
  while
    r> 1 lshift >r
```

```

    key
  repeat
  r> drop
;


```

Pour tous les autres caractères, il faudra utiliser le codage UTF8 (Unicode). Voici le codage du caractère  :

```
$88 $96 $E2 emit emit emit \ affiche 
```

On peut simplifier le codage en utilisant un tableau de trois octets :


```

create blackChar
$E2 c, $96 c, $88 c,
blackChar 3 type \ affiche 


```

Créons un mot de définition de caractères :

```

































: Unicode: ( comp: c1 c2 c3 -- <mot> | exec: --- )
  create
    c, c, c,
  does>
    3 type ;
$88 $96 $E2 Unicode: blackChar
blackChar \ affiche 

```

Le caractère  est extrait de la table Unicode documentée ici :


[https://en.wikipedia.org/wiki/Block\\_Elements](https://en.wikipedia.org/wiki/Block_Elements)

Extrait de cette table :

Blocks	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
U+258x																
U+259x																

En testant le premier caractère  de cette table avec **ukey**, on obtient ces codes :

```
hex ukey . . . \ affiche 80 96 E2
```

Puis on teste le dernier caractère  avec **ukey** :

```
hex ukey . . . \ affiche 0F 96 E2
```

Note : pour tester un caractère Unicode avec **ukey**, il faut copier/coller le caractère Unicode après avoir lancé **ukey**. Les caractères Unicode ne sont pas disponibles directement au clavier, sauf si on connaît la combinaison des touches ALT-NNNNNN. Ces combinaisons peuvent varier selon le système utilisé (Windows, Linux, MacOS...).

## Affichage des caractères Unicode

Pour afficher des caractères Unicode, on a plusieurs solutions. La plus simple consiste à utiliser une chaîne de caractères :

```
: upperHalfBlock ( -- )
    ." ■" ;
```

Ou via la définition **Unicode**:

```
$80 $96 $E2 Unicode:  upperHalfBlock
```

L'exécution de **upperHalfBlock** affiche notre caractère ■. Le mot **Unicode** exploite un tout petit tableau de trois octets. Ce tableau stocke les trois codes successifs du caractère ■. On aurait pu aussi bien utiliser **emit** comme ceci :

```
: upperHalfBlock ( -- )
    $E2 emit  $96 emit  $80 emit ;
```

La solution utilisant **type** dans la partie exécution de **Unicode** est la plus compacte.

Que se passerait-il si on utilisait un tableau de quatre octets et en y mettant juste le code ASCII d'un caractère simple ?

```
create charA
    char A c,    0 c,    0 c,    0 c,
charA 4 type    \  affiche A
```

Que ce tableau contienne 1 ou 2 ou 3 ou 4 octets valides, si la séquence correspond à un caractère Unicode valide, ce caractère sera affiché par **type**. La question sera : « comment afficher un caractère Unicode figurant dans une table Unicode ? »

On va d'abord définir un tampon d'émission de caractère vide :

```
\ buffer for Unicode char
create ubuffer
    4 allot
```

Ensuite on définit l'affichage du contenu de ce tampon :

```
\ display buffer content
: .ubuffer ( -- )
    ubuffer 4 type
;
```

On initialise le contenu de ce tampon en y stockant la séquence Unicode, sur 4 octets, du premier caractère de la table de caractères qui nous intéresse. Ici, ce sera la valeur **\$E2968000** qui correspond au caractère ■ :

```
\ select 1st code of Unicode blocks set
: uBlocksSelect ( -- )
    ubuffer 4 erase
    $E2968000. ubuffer 2!
;
```

Le mot **.uBlock** récupère un index et calcule la valeur du 3ème octets à modifier dans le tampon **ubuffer** :

```
\ display one char from Unicode blocks table
```

```

: .uBlock ( i -- ) \ i must be in interval [0..31]
  $80 + ubuffer 2 + c!
  .ubuffer
;

```

On vérifie que toutes ces définitions fonctionnent bien :

```

\ display table of Unicode blocks set
: uBlockTable ( -- )
  cr
  2 0 do
    16 0 do
      j $10 * i + .uBlock
      space
    loop
  cr cr
loop
;
uBlockSelect
uBlockTable

```



Figure 21: exécution de uBlockTable

## Caractères Unicode dans les mots Forth

La structure des noms dans Z79Forth autorise l'emploi des caractères Unicode. Exemple :

```

: été ." Summer" ;
été \ display Summer

```

On peut aussi utiliser nos caractères Unicode de blocs :

```

: █ 6 .uBlock ;

```

Le caractère █ devient un mot FORTH ! Utilisons-le dans une définition :

```

: bigLine ( n -- )
  0 do █ loop
;

```

Le mot **bigLine** va générer une barre horizontale de n caractères de longueur. Ceci nous permet de tracer un graphe :

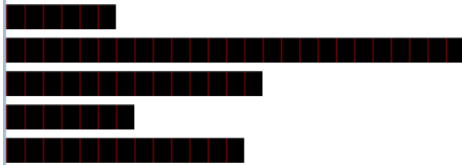
```

: graph ( -- )
  cr
  06 bigLine cr
  25 bigLine cr
  14 bigLine cr

```

```
07 bigLine cr
13 bigLine cr
;
graph
```

graph



*Figure 22: exécution de graph*

Cette manipulation peut fonctionner avec beaucoup de versions du langage Forth, en particulier toutes les versions accédant aux caractères Unicode, soit directement, soit au travers d'un terminal compatible Unicode.

Mais l'exploitation de caractères Unicode dans les noms est déconseillé, car en dehors de tous les standards de programmation.

## Utiliser du code binaire

A l'exception de rares versions du langage FORTH écrites dans certains langages de programmation, en JavaScript par exemple, quasiment toutes les versions ont un noyau écrit en code machine (code binaire).

Ce code peut être généré par un assembleur ou par un compilateur, en langage C par exemple.

Pour cette raison, une version FORTH a toujours ces contraintes :

- dépendance à un certain type de processeur (Intel, Motorola, Z80, 6809, Xtensa...) ;
- dépendance à un environnement logiciel (DOS, OSx, Windows, Linux, Android) ;

Les versions eForth et dérivées sont construites autour d'une machine FORTH virtuelle décrite en langage C. Elles ne sont liées au processeur que par l'intermédiaire des bibliothèques binaires permettant la génération de la version FORTH exploitable.

A l'origine des premiers ordinateurs personnels, l'espace mémoire était réduit au strict minimum. De par l'architecture même du processeur, disposant d'un mode d'adressage 16 bits, l'environnement de programmation utilisable se résumait à ceci :

- un langage de programmation et de contrôle matériel en ROM ou EPROM ;
- un espace de mémoire vive pour les programmes de l'utilisateur.

Sur ces ordinateurs personnels des débuts de l'informatique, le langage de programmation et la mémoire vive se partageaient 64 Kilobits d'espace mémoire adressable. Sauf à utiliser des astuces matérielles, on était limité à cet espace.

## Rétrocomputing et retour aux fondamentaux

On peut utiliser un ordinateur moderne sans chercher à comprendre comment fonctionnaient les premiers ordinateurs personnels grand public. On va résumer sommairement l'architecture d'un ordinateur actuel :

- la couche matérielle : une carte mère et son processeur, des organes de contrôle et de communication (liaison série, parallèle, écran, clavier, souris, joystick)...
- le BIOS : un programme assez petit qui donne les premières instructions au processeur, souvent inexistant sur les micro-ordinateurs domestiques ;
- le système d'exploitation chargé par le BIOS qui initialise l'environnement matériel et logiciel de haut niveau ;

- les applications mises en route par le système d'exploitation ou l'utilisateur....

Sur un ordinateur personnel des débuts de l'informatique, avec le langage assembleur, un peu de temps et une bonne documentation, il était possible, seul dans son coin, de monter une application. C'est dans les années 1977-1990 que le langage FORTH a connu une certaine gloire.

## **FORTH : compact et élégant**

Le langage FORTH fait partie de ces langages qui ont attiré beaucoup de développeurs. C'est un langage atypique, car il n'y a pas vraiment d'applications : **l'application étend le langage !**

Caractéristiques des versions FORTH pour micro-ordinateurs personnels :

- un noyau en code machine de 1Ko à 2Ko
- un ensemble de primitives étendant ce noyau à 8Ko
- quelques applications d'aide au développement : éditeur, gestion graphique, etc....

Cette extraordinaire compacité a comme corollaire une rapidité d'exécution sans égale, comparé aux autres langages de programmation de l'époque : Basic, LISP, Logo...

Certaines versions du langage FORTH intégraient un moteur multi-tâche.

Alors pourquoi FORTH a quasiment disparu en 2024 ? Une des réponses serait : à cause de LINUX. LINUX est essentiellement écrit en langage C. Pour fédérer des milliers de programmeurs, il s'est construit un éco-système de développement, cantonnant FORTH à une niche plus proche du matériel.

Un compilateur C a un fonctionnement très complexe. Pour résumer, un code source, en langage C est d'abord pré-analysé, puis il est mis en lien avec les bibliothèques requises, suivi d'une pré-compilation et s'achevant par la génération du code exécutable.

Le langage FORTH utilise deux états :

- un interpréteur : il lit les instructions provenant du clavier ou d'un fichier source ;
- un compilateur : il transforme le code source en étendant le dictionnaire. Cette transformation s'effectue en une seule passe.

En utilisant Z79Forth, vous faites donc un véritable voyage dans le temps en retrouvant un environnement minimaliste, mais très performant. La carte Z79Forth accède à un support d'enregistrement de très haute capacité :

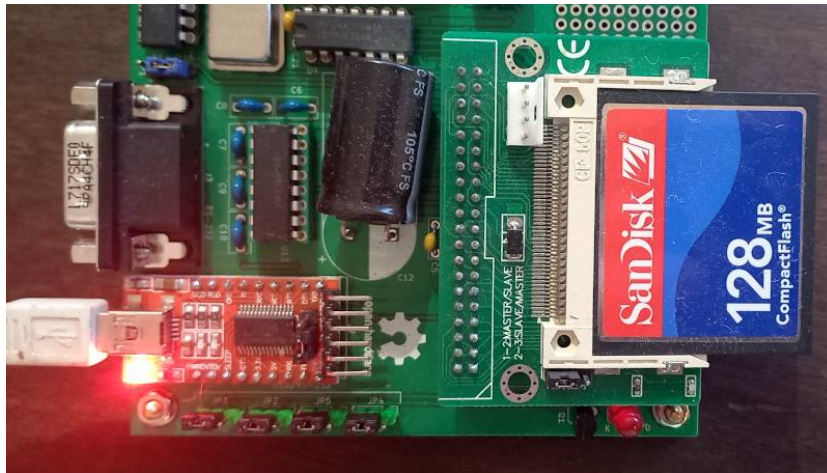


Figure 23: l'espace de stockage flash de 128 MB

Comparé aux anciens supports numériques disponibles à l'époque des micro-ordinateurs personnels, pouvoir accéder à un tel espace de stockage pour Z79Forth est tout simplement luxueux !

## Une version FORTH à chaînage direct

Il n'existe que deux types d'architecture du langage FORTH :

- **chaînage indirect** : chaque mot a une adresse d'exécution et le compilateur FORTH enregistre l'adresse d'exécution des mots à compiler. Un moteur FORTH va interpréter chaque adresse d'exécution ;
- **chaînage direct** : l'adresse d'exécution d'un mot FORTH est écrit en code machine. Le compilateur FORTH enregistre une succession d'appel de sous-programmes vers chaque mot compilé dans une définition.

Z79Forth utilise le chaînage direct.

## Décompiler les définitions Z79Forth

Z79Forth dispose d'un décompilateur/désassembleur. Pour décompiler une définition, il faut utiliser le mot **see** :

```
see dup
FC85 BDE9D7    jsr    CKDPTRA
FC88 AE        fcb    $AE
FC89 C4        fcb    $C4
FC8A 7EE7DB    jmp    NPUSH
```

Ici on a décompilé le mot **dup**.

Dans cette décompilation, à quoi correspond l'adresse **FC85** ?

```
hex
' dup u.      \ affiche: FC85
```



L'adresse **FC85** est donc l'adresse du code d'exécution de **dup**.

Dans la décompilation de **dup**, on distinguera trois parties :

```
FC85 BDE9D7    jsr    CKDPTRA
FC88 AE        fcb    $AE
FC89 C4        fcb    $C4
FC8A 7EE7DB    jmp     NPUSH
```

- en vert l'adresse à laquelle est implantée physiquement le code exécutable ;
- en rouge le code exécutable ;
- en bleu la traduction en assembleur du code exécutable...

Dans la traduction en assembleur, apparaissent deux labels, **CKDPTRA** et **NPUSH**. Ces deux labels ne figurent pas dans le vocabulaire FORTH. Dans les sources de Z79Forth :

- **CKDPTRA**      Check data stack minimum depth
- **NPUSH**        Push n on data stack

Voyons ce que donne la décompilation d'une définition compilée au-dessus du noyau Z79Forth :

```
: square dup * ;
3 square .    \ affiche: 9
4 square .    \ affiche: 16
```

Le mot **square** élève simplement un nombre au carré. Voyons sa décompilation :

```
see square
18F2 BDFC85    jsr    DUP
18F5 7EF926    jmp     *
```

Grâce au chaînage direct, Z79Forth a transformé le code source de **square** en code machine !

Avec Z79Forth, il n'est plus nécessaire de coder en langage machine, car le compilateur produit un code exécutable parfaitement optimisé.

Où se trouve l'adresse du champ d'exécution de notre mot **square** ?

```
` square hex u.    \ affiche: 18F2
```

L'adresse **18F2** est la même que celle où débute le code compilé de la définition de notre mot **square**.

Que se passe-t-il si on compile une définition vide ?

```
: emptyDef ;
see emptyDef      \ affiche:
1903 39           rts
```

Le code exécutable de **emptyDef** contient une instruction codée sur un seul octet ! Cette information va être très utile par la suite pour envisager le codage en langage machine.

## Codage de définitions en langage machine

Comme expliqué précédemment, coder en langage machine, avec Z79Forth, est inutile.

Cependant, dans certaines situations, coder en langage machine permet de définir des mots dont l'exécution sera plus rapide qu'en langage FORTH de haut niveau. Dans l'exemple qui suit, on va échanger les octets d'une valeur 16 bits déposée sur la pile de données :

```
: bswap ; -1 allot
$ec c,  $c4 c,    \ ldd ,u
$1e c,  $89 c,    \ exg a,b
$ed c,  $c4 c,    \ std ,u
$39 c,           \ rts
```

Le code binaire a été généré à l'aide d'un assembleur en ligne :

<http://6809.uk/>

Disassembly output	Labels	Assembly language input
4000: EC C4 LDD ,U		ldd ,u
4002: 1E 89 EXG A,B		exg a,b
4004: ED C4 STD ,U		std ,u
4006: 39 RTS		rts
4007: 00 00 NEG <\$00		
4009: 00 00 NEG <\$00		
400B: 00 00 NEG <\$00		
400D: 00 00 NEG <\$00		
400F: 00 00 NEG <\$00		
4011: 00 00 NEG <\$00		
4013: 00 00 NEG <\$00		

Figure 24: utilisation d'un assembleur en ligne

Le code binaire est récupéré dans la colonne *disassembly output* pour être utilisé par Z79Forth.

Test du mot **bswap** :

```
hex
1234 bswap .    \ affiche: 3412
5472 bswap .    \ affiche: 7254
```

Voici ce que donne la décompilation de notre mot **bswap** :

```
see bswap      \ affiche:
190C EC        fcb      $EC
190D C4        fcb      $C4
190E 1E        fcb      $1E
190F 89        fcb      $89
1910 ED        fcb      $ED
1911 C4        fcb      $C4
1912 39        rts
```

La manière dont notre mot **bswap** a été codé doit rester très ponctuelle et répondre exclusivement à des critères de performance et de compacité :

- l'usage du code machine réduit la portabilité du code source ;
- il faut apprendre l'assembleur 6809/6309 pour utiliser le code machine ;
- il faut disposer de certains outils en dehors de Z79Forth, ici un assembleur en ligne.

Aurait-on pu écrire **bswap** entièrement en FORTH ? Oui :

```
: bswap ( n1 - n2 )
    $100 /mod
    swap $100 * + ;
```

Ce code est simplement plus long en terme d'espace mémoire occupé dans le dictionnaire, mais aussi moins rapide en exécution que la version en code machine.

Si vous êtes amené à définir un mot en code machine, il est fortement conseillé de mettre dans vos fichiers sources les deux versions, de cette manière :

```
\ invert bytes of n
\ : bswap ( n1 - n2 )
\     $100 /mod
\     swap $100 * + ;
: bswap ; -1 allot
$ec c,  $c4 c,  \ ldd ,u
$1e c,  $89 c,  \ exg a,b
$ed c,  $c4 c,  \ std ,u
$39 c,      \ rts
```

Ici, la version définie en FORTH est mise en commentaire.

Cette manière de procéder facilitera la réutilisation du code source si vous êtes amené à transférer l'application vers une version du langage FORTH autre que Z79Forth.

## Ressources

- **Z79FORTH / Facebook**  
<https://www.facebook.com/groups/505661250539263>
- **Z79Forth Blog**  
<https://z79forth.blogspot.com/>
- **FORTH code analyzer**  
Enter your FORTH code. Select Z79Forth. Find your code with syntax highlighting and hyperlinks to known words  
<https://analyzer.arduino-forth.com/>

## GitHub

- **Z79forth** is a journey into retro computing that takes advantage of modern technologies where appropriate (CMOS, USB and CompactFlash)  
<https://github.com/frenchie68/Z79Forth>

## Youtube

- CPE1704TKS: Engineering Field Notes from a Debugging Session  
<https://www.youtube.com/watch?v=OFIxfCywh0Q>

## Index

and.....	27	dup.....	49	u.....	25
base.....	32	emit.....	36	value.....	50
c!.....	50	format HH:MM:SS.....	30	variable.....	50
c@.....	50	GIT.....	17	vitesse de transmission série.	11
caractéristiques.....	7	HD63C09E.....	9	vlist.....	62
cavalier JP1.....	11	hex.....	22, 32	;	47
cavaliers JP2 à JP5.....	11	hold.....	34	:	47
chaînage direct.....	80	last.....	61	."	37
code binaire.....	78	mémoire.....	49	.s.....	44
composants.....	8	Netbeans.....	17	@.....	49
constant.....	50	pile de retour.....	49	#.....	34
create.....	31, 52	préfixe.....	33	#>.....	34
decimal.....	22, 32	s".....	37	#s.....	34
DOES>.....	52	space.....	37	<#.....	34
drop.....	49	struct.....	70		