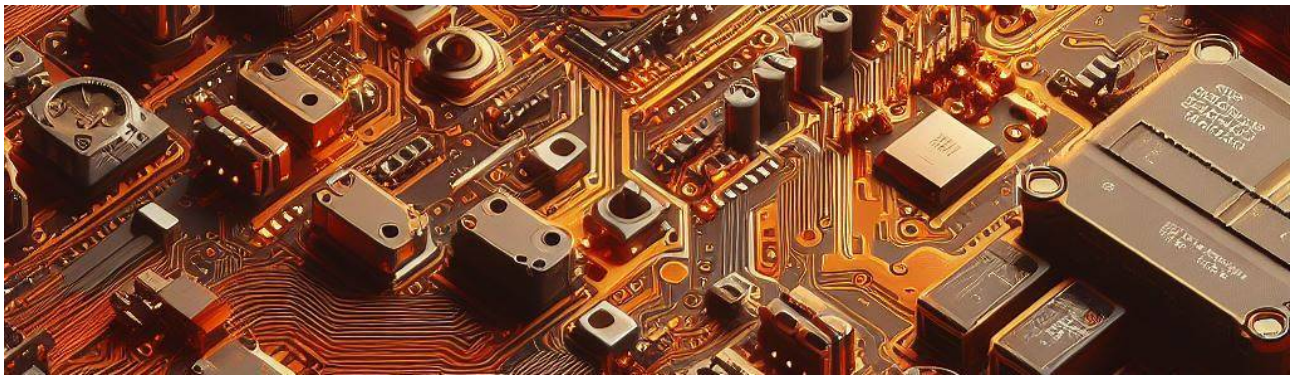
Le grand livre de eFORTH Linux

version 1.3 - 7 décembre 2023



Auteur

- Marc PETREMANN

Collaborateur(s)

- xxx

Table des matières

Introduction

Je gère depuis 2019 plusieurs sites web consacrés aux développements en langage FORTH pour les cartes ARDUINO et ESP32, ainsi que les versions eForth web – Linux - Windows :

- ARDUINO : https://arduino-forth.com/
- ESP32 : https://esp32.arduino-forth.com/
- eForth web : https://eforth.arduino-forth.com/

Ces sites sont disponibles en deux langues, français et anglais. Chaque année je paie l'hébergement du site principal **arduino-forth.com**.

Il arrivera tôt ou tard – et le plus tard possible – que je ne sois plus en mesure d'assurer la pérennité de ces sites. La conséquence sera que les informations diffusées par ces sites disparaissent.

Ce livre est la compilation du contenu de mes sites web. Il est diffusé librement depuis un dépôt Github. Cette méthode de diffusion permettra une plus grande pérennité que des sites web.

Accessoirement, si certains lecteurs de ces pages souhaitent apporter leur contribution, ils sont bienvenus :

- pour proposer des chapitres ;
- pour signaler des erreurs ou suggérer des modifications ;
- pour aider à la traduction…

Aide à la traduction

Google Translate permet de traduire des textes facilement, mais avec des erreurs. Je demande donc de l'aide pour corriger les traductions.

En pratique, je fournis, les chapitres déjà traduits, dans le format LibreOffice. Si vous voulez apporter votre aide à ces traductions, votre rôle consistera simplement à corriger et renvoyer ces traductions.

La correction d'un chapitre demande peu de temps, de une à quelques heures.

Pour me contacter : petremann@arduino-forth.com

Installer eForth sous Linux

eForth Linux est une version très puissante destinée au système Linux. eForth Linux fonctionne sur toutes les versions récentes de Linux, y compris dans un environnement virtuel Linux.

Prérequis

Vous devez disposer d'un système Linux opérationnel :

- installé sur un ordinateur utilisant Linux comme seul système d'exploitation ;
- installé dans un environnement virtuel.

Si vous disposez seulement d'un ordinateur sous Windows 10 ou 11, vous pouvez installer Linux dans le sous-système **WSL**[1].

Le Sous-système Windows pour Linux permet aux développeurs d'exécuter un environnement GNU/Linux (et notamment la plupart des utilitaires, applications et outils en ligne de commande) directement sur Windows, sans modification et tout en évitant la surcharge d'une machine virtuelle traditionnelle ou d'une configuration à double démarrage.

L'intérêt d'une installation d'une distribution Linux dans **WSL** permet d'avoir à disposition une version Linux en mode commande en quelques secondes. Ici, **Ubuntu** est accessible depuis le système de fichiers Windows et se lance en un seul clic :

1 WSL = Windows Subsystem Linux

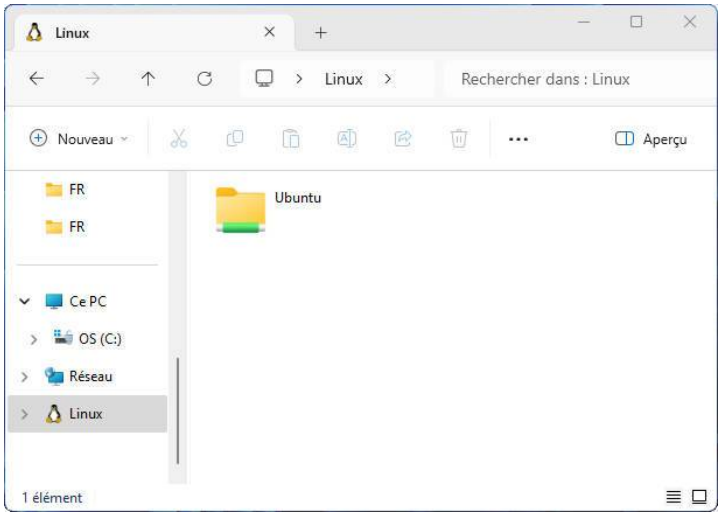


Figure 1: Ubuntu accessible en un clic depuis WSL sous Windows

Toutes les instructions pour installer **WSL2** puis la distribution Linux de votre choix sont disponibles ici :
https://learn.microsoft.com/fr-fr/windows/wsl/install

Par défaut, WSL2 propose d'installer la distribution **Linux Ubuntu**.

Installer eForth Linux sous Linux

Si vous lancez Ubuntu (ou toute autre version de Linux), vous vous retrouverez par défaut dans votre répertoire utilisateur. On commence par accéder au dossier **usr/bin** :

```
cd /usr/bin
```

On va maintenant télécharger la version du fichier binaire de ueForth Linux :

- soit depuis la page d'accueil du site ESP32forth de Brad NELSON :
  https://esp32forth.appspot.com/ESP32forth.html
- soit depuis le dépôt de stockage eforth Google :
  https://eforth.storage.googleapis.com/releases/archive.html

Dans la liste des fichiers proposés, copiez le lien web mentionnant linux :

```
https://eforth.storage.googleapis.com/releases/ueforth-7.0.7.15.linux
```

Sous Linux, tapez la commande `wget` :

```
sudo wget https://eforth.storage.googleapis.com/releases/ueforth-7.0.7.15.linux
```

Le téléchargement va déposer automatiquement le fichier dans le dossier **usr/bin** précédemment sélectionné. Si vous avez repris le lien ci-avant, vous vous retrouvez avec un fichier nommé **ueforth-7.0.7.15.linux** dans ce dossier.

On renomme ce fichier avec la commande `mv` :

```
sudo mv ueforth-7.0.7.15.linux ueforth
```

On vérifie que tout s'est bien déroulé avec une simple commande `dir ue*`. On doit voir la présence de notre fichier **ueforth**.

Il nous reste une dernière manipulation à effectuer, rendre ce fichier exécutable par le système Linux :

```
sudo chmod 755 ueforth
```

Et c'est fini ! eForth Linux est maintenant utilisable depuis n'importe quel répertoire Linux.

Lancer eForth Linux

Pour lancer **eForth** au démarrage de **Linux** :

```
ueforth
```

eForth Linux démarre aussitôt :

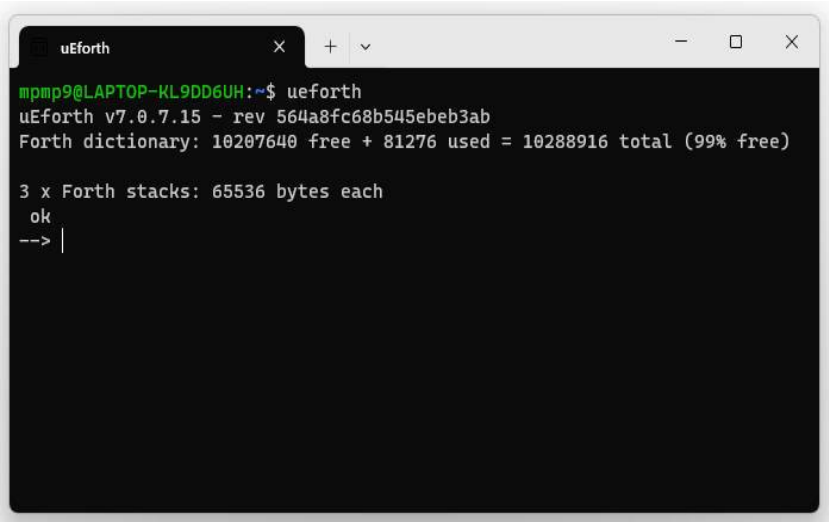


Figure 2: eForth Linux est actif

Vous pouvez maintenant tester eForth et programmer vos premières applications en langage FORTH.

ATTENTION : cette version eForth gère les entiers au format 64 bits. C'est facile à vérifier :

```
cell .      \ display : 8
```

Soit une dimension de 8 octets pour les entiers. Cet avertissement est indispensable si vous reprenez du code FORTH écrit pour des versions 16 ou 32 bits.

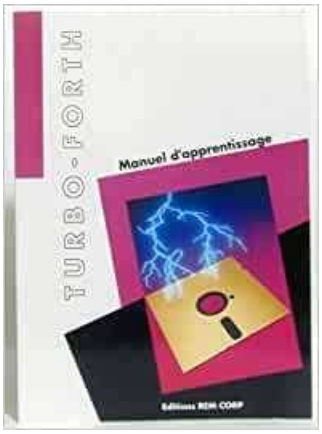
Bonne programmation.

Pourquoi programmer en langage FORTH sur eForth Linux?

Préambule

Je programme en langage FORTH depuis 1983. J'ai cessé de programmer en FORTH en 1996. Mais je n'ai jamais cessé de surveiller l'évolution de ce langage. J'ai repris la programmation en 2019 sur ARDUINO avec FlashForth puis ESP32forth.

Je suis co-auteur de plusieurs livres concernant le langage FORTH :



- Introduction au ZX-FORTH (ed Eyrolles - 1984 - ASIN:B0014IGOXO)
- Tours de FORTH (ed Eyrolles - 1985 - ISBN-13: 978-2212082258)
- FORTH pour CP/M et MSDOS (ed Loisitech - 1986)
- TURBO-Forth, manuel d'apprentissage (ed Rem CORP - 1990)
- TURBO-Forth, guide de référence (ed Rem CORP - 1991)

La programmation en langage FORTH a toujours été un loisir jusqu'en 1992 où le responsable d'une société travaillant en sous-traitance pour l'industrie automobile me contacte. Ils avaient un souci de développement logiciel en langage C. Il leur fallait commander un automate industriel.

Les deux concepteurs logiciels de cette société programmaient en langage C: TURBO-C de Borland pour être précis. Et leur code n'arrivait pas à être suffisamment compact et rapide pour tenir dans les 64 Kilo-octets de mémoire RAM. On était en 1992 et les extensions de type mémoire flash n'existaient pas. Dans ces 64 Ko de mémoire vive, ll fallait faire tenir MS-DOS 3.0 et l'application !

Celà faisait un mois que les développeurs en langage C tournaient le problème dans tous les sens, jusqu'à réaliser du reverse engineering avec SOURCER (un désassembleur) pour éliminer les parties de code exécutable non indispensables.

J'ai analysé le problème qui m'a été exposé. En partant de zéro, j'ai réalisé, seul, en une semaine, un prototype parfaitement opérationnel qui tenait le cahier des charges. Pendant trois années, de 1992 à 1995, j'ai réalisé de nombreuses versions de cette application qui a été utilisée sur les chaînes de montage de plusieurs constructeurs automobiles.

Limites entre langage et application

Tous les langages de programmation sont partagés ainsi :

- un interpréteur et le code source exécutable: BASIC, PHP, MySQL, JavaScript, etc... L'application est contenue dans un ou plusieurs fichiers qui sera interprété chaque fois que c'est nécessaire. Le système doit héberger de manière permanente l'interpréteur exécutant le code source ;
- un compilateur et/ou assembleur : C, Java, etc... Certains compilateurs génèrent un code natif, c'est à dire exécutable spécifiquement sur un système. D'autres, comme Java, compilent un code exécutable sur une machine Java virtuelle.

Le langage FORTH fait exception. Il intègre :

- un interpréteur capable d'exécuter n'importe quel mot du langage FORTH
- un compilateur capable d'étendre le dictionnaire des mots FORTH.

C'est quoi un mot FORTH?

Un mot FORTH désigne toute expression du dictionnaire composée de caractères ASCII et utilisable en interprétation et/ou en compilation : `words` permet de lister tous les mots du dictionnaire FORTH.

Certains mots FORTH ne sont utilisables qu'en compilation: `if else then` par exemple.

Avec le langage FORTH, le principe essentiel est qu'on ne crée pas une application. En FORTH, on étend le dictionnaire ! Chaque mot nouveau que vous définissez fera autant partie du dictionnaire FORTH que tous les mots pré-définis au démarrage de FORTH. Exemple :

```
: typeToLoRa ( -- )
    0 echo !    \ desactive l'echo d'affichage du terminal
    ['] serial2-type is type
  ;
: typeToTerm ( -- )
    ['] default-type is type
    -1 echo !   \ active l'echo d'affichage du terminal
  ;
```

On crée deux nouveaux mots: `typeToLoRa` et `typeToTerm` qui vont compléter le dictionnaire des mots pré-définis.

Un mot c'est une fonction?

Oui et non. En fait, un mot peut être une constante, une variable, une fonction... Ici, dans notre exemple, la séquence suivante :

```
: typeToLoRa ...code... ;
```

aurait son équivalent en langage C:

```
void typeToLoRa() { ...code... }
```

En langage FORTH, il n'y a pas de limite entre le langage et l'application.

En FORTH, comme en langage C, on peut utiliser n'importe quel mot déjà défini dans la définition d'un nouveau mot.

Oui, mais alors pourquoi FORTH plutôt que C ?

Je m'attendais à cette question.

En langage C, on ne peut accéder à une fonction qu'au travers de la principale fonction `main()`. Si cette fonction intègre plusieurs fonctions annexes, il devient difficile de retrouver une erreur de paramètre en cas de mauvais fonctionnement du programme.

Au contraire, avec FORTH, il est possible d'exécuter - via l'interpréteur - n'importe quel mot pré-défini ou défini par vous, sans avoir à passer par le mot principal du programme.

L'interpréteur FORTH est immédiatement accessible sur la carte ESP32 via un programme de type terminal et une liaison USB entre la carte ESP32 et le PC.

La compilation des programmes écrits en langage FORTH s'effectue dans la carte ESP32 et non pas sur le PC. Il n'y a pas de téléversement. Exemple :

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

Cette définition est transmise par copié/collé dans le terminal. L'interpréteur/compilateur FORTH va analyser le flux et compiler le nouveau mot `>gray`.

Dans la définition de `>gray`, on voit la séquence `dup 2/ xor`. Pour tester cette séquence, il suffit de la taper dans le terminal. Pour exécuter `>gray`, il suffit de taper ce mot dans le terminal, précédé du nombre à transformer.

Le langage FORTH comparé au langage C

C'est la partie que j'aime le moins. Je n'aime pas comparer le langage FORTH par rapport au langage C. Mais comme quasiment tous les développeurs utilisent le langage C, je vais tenter l'exercice.

Voici un test avec `if()` en langage C:

```
if(j > 13){                 // Si tous les bits sont recus
    rc5_ok = 1;             // Le processus de decodage est OK
    detachInterrupt(0);     // Desactiver l'interruption externe (INT0)
    return;
}
```

Test avec `if` en langage FORTH (extrait de code):

```
var-j @ 13 >        \ Si tous les bits sont recus
    if
        1 rc5_ok !  \ Le processus de decodage est OK
        di          \ Desactiver l'interruption externe (INT0)
        exit
    then
```

Voici l'initialisation de registres en langage C:

```
void setup() {
  //  Configuration du module Timer1
  TCCR1A = 0;
  TCCR1B = 0;          // Desactive le module Timer1
  TCNT1  = 0;          // Definit valeur préchargement Timer1 sur 0 (reset)
  TIMSK1 = 1;          // activer interruption de debordement Timer1
}
```

La même définition en langage FORTH:

```
: setup ( -- )
  \ Configuration du module Timer1
  0 TCCR1A !
  0 TCCR1B !     \ Desactive le module Timer1
  0 TCNT1  !     \ Définit valeur préchargement Timer1 sur 0 (reset)
  1 TIMSK1 !     \ activer interruption de debordement Timer1
;
```

Ce que FORTH permet de faire par rapport au langage C

On l'a compris, FORTH donne immédiatement accès à l'ensemble des mots du dictionnaire, mais pas seulement. Via l'interpréteur, on accède aussi à toute la mémoire allouée à eForth Linux :

```
hex here 100 dump
```

Vous devez retrouver quelque chose qui ressemble à ceci sur l'écran du terminal :

```
3FFEE964                     DF DF 29 27 6F 59 2B 42 FA CF 9B 84
3FFEE970           39 4E 35 F7 78 FB D2 2C A0 AD 5A AF 7C 14 E3 52
3FFEE980           77 0C 67 CE 53 DE E9 9F 9A 49 AB F7 BC 64 AE E6
3FFEE990           3A DF 1C BB FE B7 C2 73 18 A6 A5 3F A4 68 B5 69
3FFEE9A0           F9 54 68 D9 4D 7C 96 4D 66 9A 02 BF 33 46 46 45
3FFEE9B0           45 39 33 33 2F 0D 08 18 BF 95 AF 87 AC D0 C7 5D
3FFEE9C0           F2 99 B6 43 DF 19 C9 74 10 BD 8C AE 5A 7F 13 F1
3FFEE9D0           9E 00 3D 6F 7F 74 2A 2B 52 2D F4 01 2D 7D B5 1C
3FFEE9E0           4A 88 88 B5 2D BE B1 38 57 79 B2 66 11 2D A1 76
3FFEE9F0           F6 68 1F 71 37 9E C1 82 43 A6 A4 9A 57 5D AC 9A
3FFEEA00           4C AD 03 F1 F8 AF 2E 1A B4 67 9C 71 25 98 E1 A0
3FFEEA10           E6 29 EE 2D EF 6F C7 06 10 E0 33 4A E1 57 58 60
3FFEEA20           08 74 C6 70 BD 70 FE 01 5D 9D 00 9E F7 B7 E0 CA
3FFEEA30           72 6E 49 16 0E 7C 3F 23 11 8D 66 55 EC F6 18 01
3FFEEA40           20 E7 48 63 D1 FB 56 77 3E 9A 53 7D B6 A7 A5 AB
```

```
3FFEEA50            EA 65 F8 21 3D BA 54 10 06 16 E6 9E 23 CA 87 25
3FFEEA60            E7 D7 C4 45
```

Ceci correspond au contenu de la mémoire flash.

Et ça, le langage C ne saurait pas le faire?

Si. mais pas de façon aussi simple et interactive qu'en langage FORTH.

Voyons un autre cas mettant en avant l'extraordinaire compacité du langage FORTH...

Mais pourquoi une pile plutôt que des variables?

La pile est un mécanisme implanté sur quasiment tous les microcontrôleurs et microprocesseurs. Même le langage C exploite une pile, mais vous n'y avez pas accès.

Seul le langage FORTH donne un accès complet à la pile de données. Par exemple, pour faire une addition, on empile deux valeurs, on exécute l'addition, on affiche le résultat: `2 5 +` . affiche `7`.

C'est un peu déstabilisant, mais quand on a compris le mécanisme de la pile de données, on apprécie grandement sa redoutable efficacité.

La pile de données permet un passage de données entre mots FORTH bien plus rapidement que par le traitement de variables comme en langage C ou dans n'importe quel autre langage exploitant des variables.

Êtes-vous convaincus?

Personnellement, je doute que ce seul chapitre vous convertisse irrémédiablement à la programmation en langage FORTH. En cherchant à maîtriser LINUX, vous avez deux possibilités :

- programmer en langage C et exploiter les nombreuses librairies disponibles. Mais vous resterez enfermés dans les capacités de ces librairies. L'adaptation des codes en langage C requiert une réelle connaissance en programmation en langage C et maîtriser l'architecture des cartes ESP32. La mise au point de programmes complexes sera toujours un souci.
- tenter l'aventure FORTH et explorer un monde nouveau et passionnant. Certes, ce ne sera pas facile. Il faudra comprendre l'architecture LINUX, les librairies... En contrepartie, vous aurez accès à une programmation parfaitement adaptée à vos projets.

Existe-t-il des applications professionnelles écrites en FORTH?

Oh oui! A commencer par le télescope spatial HUBBLE dont certains composants ont été écrits en langage FORTH.

Le TGV allemand ICE (Intercity Express) utilise des processeurs RTX2000 pour la commande des moteurs via des semi-conducteurs de puissance. Le langage machine du processeur RTX2000 est le langage FORTH.

Ce même processeur RTX2000 a été utilisé pour la sonde Philae qui a tenté d'atterrir sur une comète.

Le choix du langage FORTH pour des applications professionnelles s'avère intéressant si on considère chaque mot comme une boîte noire. Chaque mot doit être simple, donc avoir une définition assez courte et dépendre de peu de paramètres.

Lors de la phase de mise au point, il devient facile de tester toutes les valeurs possibles traitées par ce mot. Une fois parfaitement fiabilisé, ce mot devient une boîte noire, c'est à dire une fonction dont on fait une confiance sans limite à son bon fonctionnement. De mot en mot, on fiabilise plus facilement un programme complexe en FORTH que dans n'importe quel autre langage de programmation.

Mais si on manque de rigueur, si on construit des usines à gaz, il est aussi très facile d'obtenir une application qui fonctionne mal, voire de planter carrément FORTH!

Pour finir, il est possible, en langage FORTH, d'écrire les mots que vous définissez dans n'importe quelle langue humaine. Cependant, les caractères utilisables sont limités au jeu de caractères ASCII compris entre 33 et 127. Voici comment on pourrait réécrire de manière symbolique les mots **high** et **low**:

```
\ Active broche de port, ne changez pas les autres.
: __/ ( pinmask portadr -- )
    mset
  ;
\ Desactivez une broche de port, ne change pas les autres.
: \__ ( pinmask portadr -- )
    mclr
  ;
```

A partir de ce moment, pour allumer la LED, on peut taper :

```
_o_  __/     \ allume LED
```

Oui! La séquence **_o_ __/** est en langage FORTH !

Bonne programmation.

Un vrai FORTH 64 bits avec eForth Linux

eForth Linux est un vrai FORTH 64 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 eForth Linux, l'interpréteur FORTH est disponible. Si vous entrez n'importe quel nombre, il sera déposé sur la pile sous sa forme d'entier 64 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 64 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 + .  \ display 80
```

Contrairement au langage C, on ne définit pas de type **int8** ou **int16** ou **int32** ou **int64**.

Avec eForth Linux, un caractère ASCII sera désigné par un entier 64 bits, mais dont la valeur sera bornée [32..256[. Exemple :

```
67 emit  \ display C
```

Les valeurs en mémoire

eForth Linux permet de définir des constantes, des variables. Leur contenu sera toujours au format 64 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 64 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 64 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 -- )
    <# :## :## # # #>  type
  ;
4225 .hms  \ display: 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 est TOUJOURS un entier 64 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
        c,
    does>
        dup 1+ swap c@ 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    \ display   .- -... -.-.
```

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....

Edition et gestion fichiers sources pour eForth Linux

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 ;
- **fs** notre extension préférée…

Les éditeurs de fichiers texte

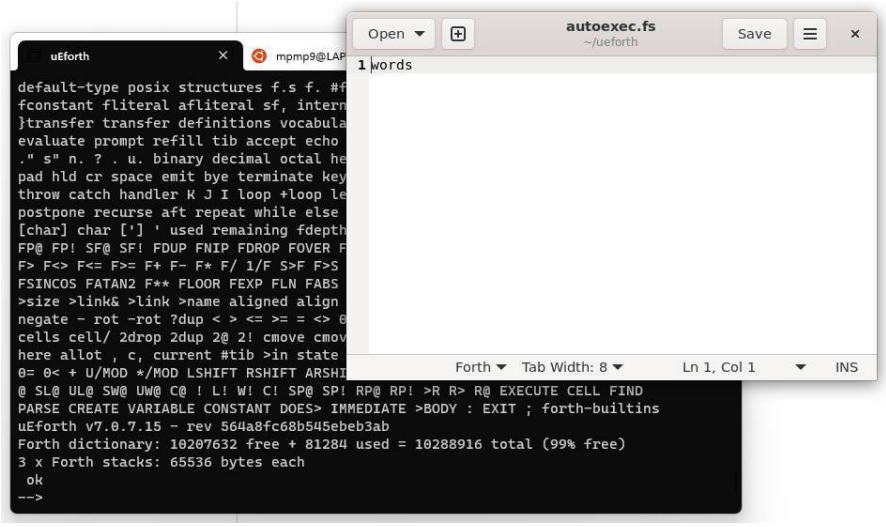


Figure 3: édition du fichier autoexec.fs avec gedit sous Linux

Sous Linux, l'éditeur de fichiers **gedit** est le plus simple :

Si vous utilisez une extension de fichier personnalisé, comme **fs**, pour vos fichiers source en langage FORTH, Linux reconnaîtra ces fichiers comme textes simples.

Stockage sur GitHub

Le site web **GitHub**[2] est, avec **SourceForge**[3], un des meilleurs endroits pour stocker ses fichiers sources.

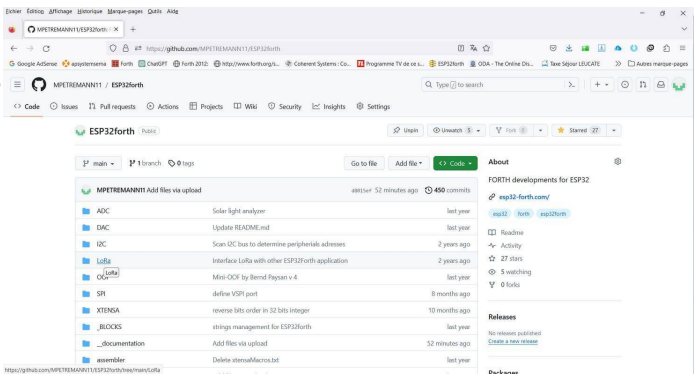


Figure 4: stockage des fichiers sur Github

2 https://github.com/
3 https://sourceforge.net/

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.

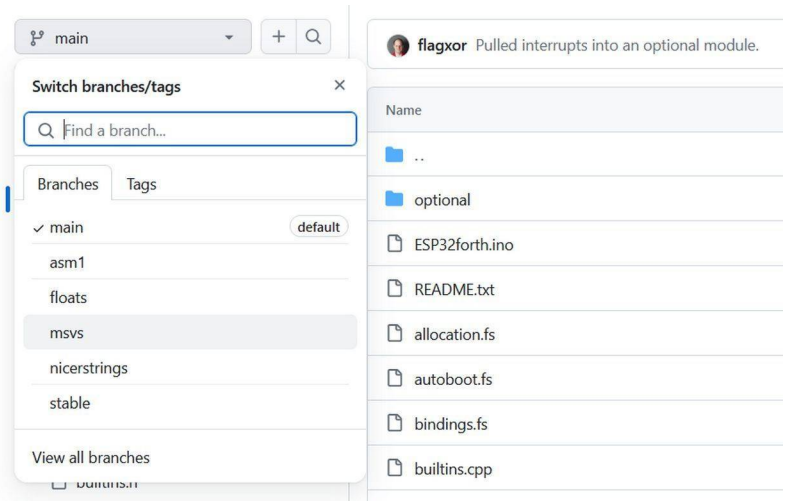


Figure 5: accès à une branche dans un projet

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

Editer des fichiers pour eForth Linux depuis Windows

Si vous avez installé une version Linux qui s'exécute dans l'environnement WSL2, il est parfaitement possible d'éditer les fichiers sources Linux depuis Windows :

- lancez Ubuntu depuis Windows
- une fois Ubuntu actif, sortez le pointeur de souris de la fenêtre WSL. Vous revenez ainsi dans l'environnement Windows. Ouvrez le gestionnaire de fichiers Windows.
- dans le volet de gauche, cliquez sur *Linux* ;

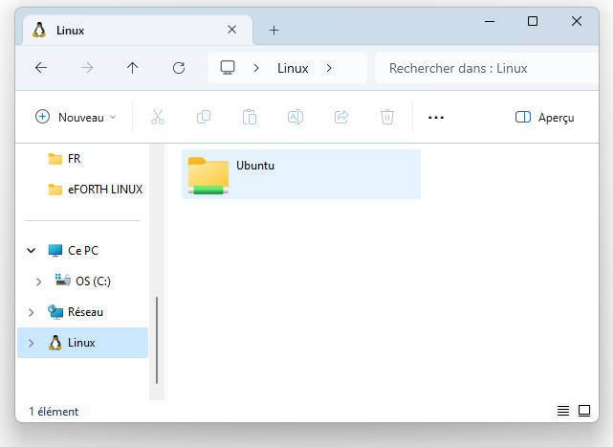


Figure 6: accès aux fichiers Linux depuis Windows

- dans le volet principal, cliquez sur la version Linux, ici *Ubuntu* ;

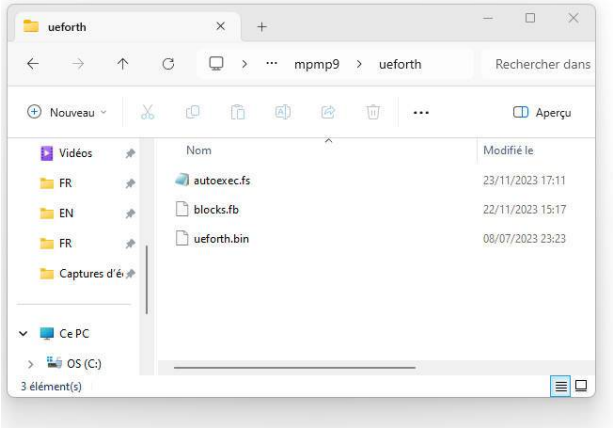


Figure 7: les fichiers Linux visibles depuis Windows

- accédez au dossier eForth : dossier home → Utilisateur → ueforth →
- sélectionnez le fichier à éditer. Pour l'exemple, on va ouvrir **autoexec.fs** ;

si vous utilisez un IDE, comme Netbeans, voici comment paramétrer cet IDE pour y intégrer vos projets de développement eForth Linux.

Création et gestion de projets FORTH avec Netbeans

En pré-requis, vous devez installer Netbeans. Lien pour téléchargement et installation : https://netbeans.apache.org/front/main/

Netbeans peut s'installer sous Windows ou Linux. Pour ma part, ayant déjà Netbeans installé sous Windows, je ne vais pas surcharger ma machine en installant une version Linux. En conséquence, les explications qui suivent concernent la gestion d'un projet eForth Linux via WSL2 depuis Windows.

Créer un projet eForth avec Netbeans

Là, également un pré-requis :

- ueforth Linux est installé dans Linux via WSL2 Windows.
- Les fichiers sources sont dans un dossier Windows :
  `Linux → Ubuntu → home → userName → ueforth`
  où `userName` est le nom d'utilisateur défini à l'installation de Linux
- tous les fichiers source eForth Linux sont enregistrés dans le répertoire `ueforth`

Lancez Netbeans. Pour créer un nouveau projet Netbeans :

- cliquez sur *File* → sélectionnez *New Project…*

- dans la fenêtre **New Project**, sélectionnez Catégories : *PHP* et dans Projects : *PHP Application with Existing Sources*

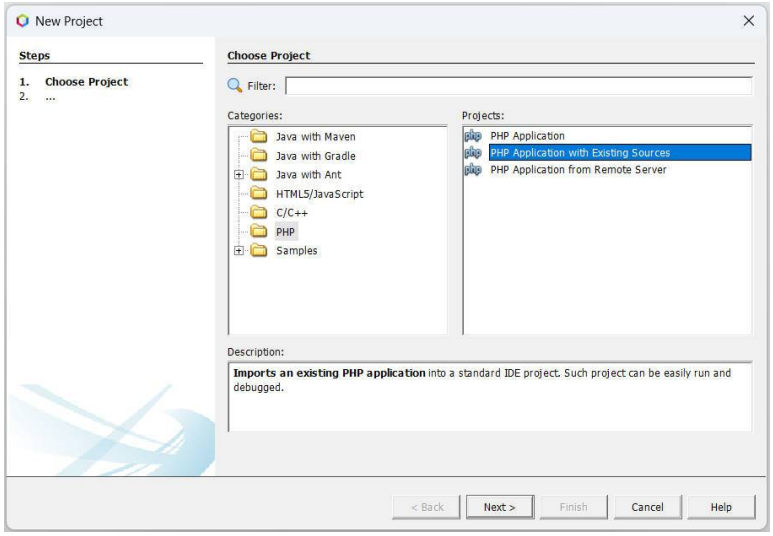


Figure 8: création projet PHP

- cliquez sur *Next >*
- Dans le champ **Name and Location** → *Sources Folder*, saisissez le chemin des fichiers source eForth Linux

Pour retrouver le bon chemin du dossier **ueforth** depuis Windows, lancez l'explorateur de fichiers. Dans la partie inférieure droite, cliquez sur **Ubuntu**. Ensuite cliquez sur les dossiers :

`home → userName → ueforth`

Dans le bandeau de navigation, en haut, vous devez retrouver le chemin du dossier ueforth. Posez le pointeur de souris dans ce bandeau. Copiez le chemin :

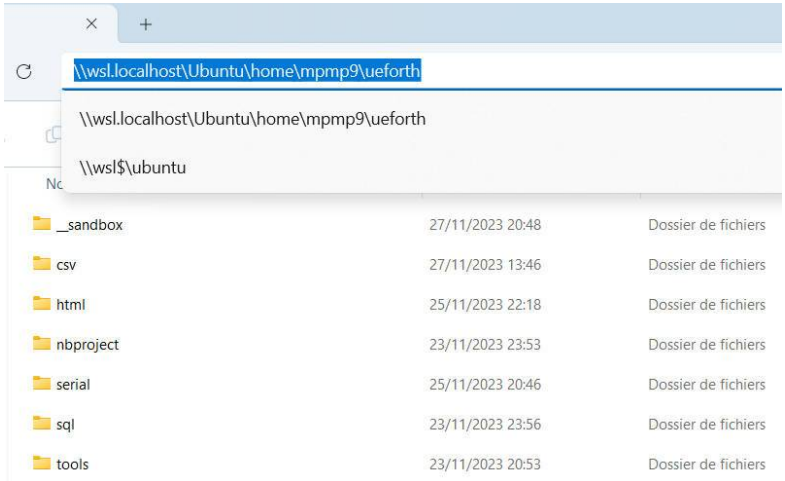


Figure 9: copie du chemin vers ueforth sous Linux

Collez ce chemin dans le champ de Netbeans décrit plus haut. Terminez la création du nouveau projet dans Netbeans. Vous pouvez maintenant retrouver tous les fichiers de votre projet dans Netbeans :

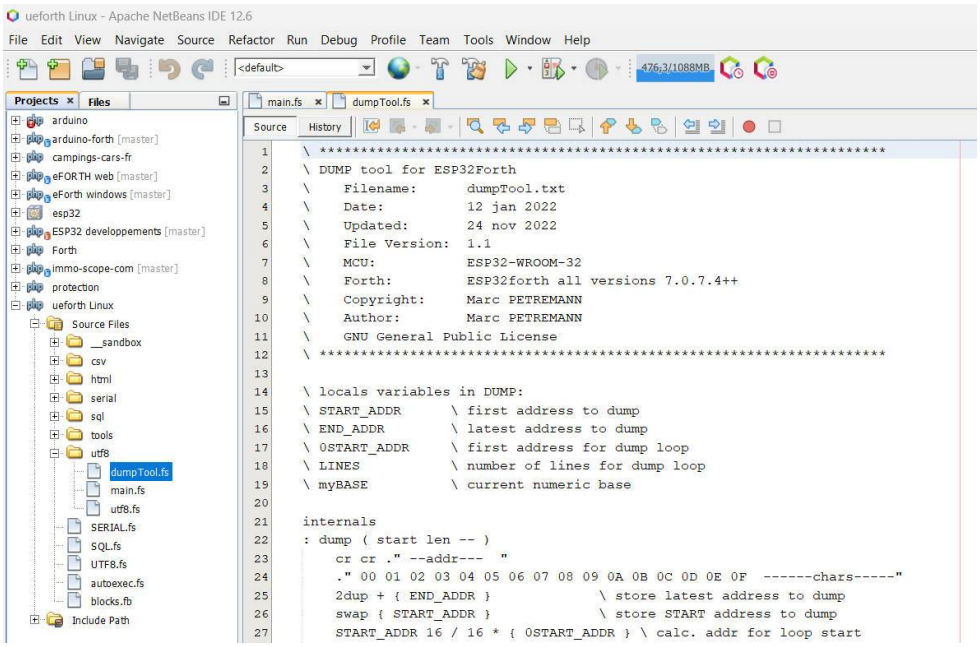


Figure 10: le nouveau projet est opérationnel

Maintenant, toute édition, création, modification ou suppression de fichier depuis Netbeans est immédiatement répercuté dans le dossier de votre projet **ueforth** sous Linux.

Quelques bonnes pratiques

La première bonne pratique consiste à bien nommer ses fichiers et dossiers de travail. Vous développez pour eforth, donc restez dans le dossier nommé **ueforth**.

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 voulez développer un jeu, créez un sous-dossier **myGame**.

Si vous avez des scripts d'usage général, créez un sous-dossiet **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.

Pour les essais FORTH sans but précis, mettez-les dans un dossier **__sandbox**.

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

- **config.fs** 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 ;

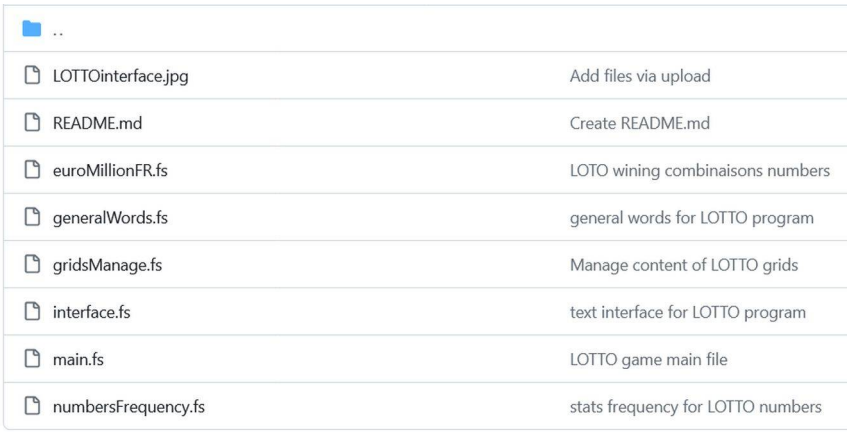


Figure 11: exemple de nommage de fichiers source Forth

- **myApp.fs** pour les définitions de votre projet. Choisissez un nom de fichier assez explicite . Par exemple, pour gérer votre jeu, prenez le nom **game-commands.fs**.

Exécution du contenu d'un fichier par eForth Linux

Depuis eForth Linux, l'exécution du contenu d'un fichier source s'effectue très simplement en utilisant le mot `include` suivi du nom de fichier :

```
include autoexec.fs
```

exécute le contenu du fichier **autoexec.fs**.

Si le fichier à lire est dans un sous-dossier, on fera précéder le nom du fichier par le nom du dossier. Exemple pour lancer **main.fs** dans le sous-dossier **myGame** :

```
cd mygame
include main.fs
```

Si vous avez correctement installé **ueforth**, son lancement peut être suivi du nom du ficher source à exécuter. Sous Linux :

```
cd ueforth
ueforth UTF8.fs
```

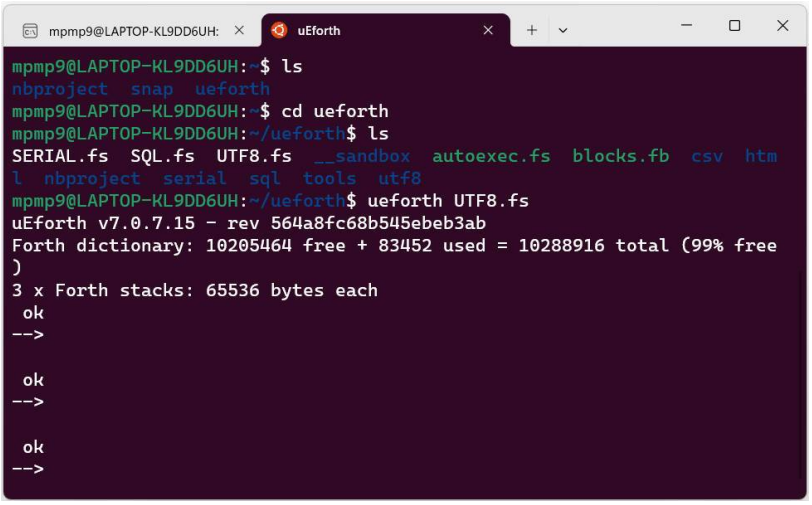


Figure 12: exécution d'un fichier au lancement de ueforth

Linux enregistre toutes les commandes système, même après arrêt du PC et redémmarage. Il est donc très facile de recommencer un traitement de projet en quelques appuis de touche *flèche vers le haut*.

En résumé, sous réserve d'avoir une version Linux accessible depuis **Windows WSL2**, vous éditez les fichiers source avec Netbeans depuis Windows. Et vous traitez les fichiers d'un projet depuis Linux.

Si vous êtes dans un environnement *full Linux*, les manœuvres ne sont pas très différentes. Pour lancer ueforth, vous devrez ouvrir une fenêtre de commandes sous Linux.

Le système de fichiers Linux

eForth Linux intègre les composants essentiels pour accéder aux fichiers du système Linux.

Pour compiler le contenu d'un fichier source, ici le fichier **dumpTool.fs** dans le dossier **tools**, édité par **gedit**, taper :

```
include /tools/dumpTool.fs
```

Le mot `include` est un mot du dictionnaire eForth.

Pour voir la liste des fichiers Linux, utilisez le mot `ls`:

```
ls   \ display :
.
..
autoexec.fs
blocks.fb
ueforth.bin
tools
 ok
```

Ici, on voit le dossier **tools**. Eforth Linux n'utilise pas de coloration syntaxique comme le fait Linux. Pour voir le contenu de ce sous-dossier **tools**, taper :

```
ls tools   \ display :
ls tools
.
..
dumpTool.fs
```

Il n'y a pas d'option de filtrage des noms de fichiers ou de pseudo-répertoires.

Manipulation des fichiers

Pour effacer intégralement un fichier, utiliser le mot `rm` suivi du nom de fichier à supprimer. Ici on souhaite effacer le fichier myTest.fs qui a été créé et ne sert plus :

```
rm myTest.fs  \ display :
  ok
```

Pour renommer un fichier, utilisez le mot `mv`. Par exemple, on veut renommer un fichier **myTest.txt**:

```
mv myTest.txt myTest.fs
ls   \ display :
.
..
autoexec.fs
blocks.fb
```

```
myTest.fs
tools
```

Pour copier un fichier, utilisez le mot **cp**:

```
cp myTest.fs testColors.fs
ls  \ display :
.
..
autoexec.fs
blocks.fb
myTest.fs
testColors.fs
tools
```

Pour voir le contenu d'un fichier, utilisez le mot **cat**:

```
cat autoexec.fs
\ affiche contenu de autoexec.fs
```

Pour enregistrer le contenu d'une chaîne dans un fichier, on enregistre le contenu de la chaîne avec **dump-file** :

```
r| ." Insère mon texte dans myTest" |   s" myTest.fs"   dump-file
```

On ne s'étendra pas sur ces manipulations qui peuvent aussi bien s'effectuer depuis Linux ou un éditeur de texte source.

Organiser et compiler ses fichiers avec eForth Linux

Nous allons voir comment gérer des fichiers pour une application en cours de mise au point avec eForth Linux.

Il est convenu que tous les fichiers utilisés sont au format texte ASCII.

Les explications qui suivent ne sont données qu'à titre de conseils. Ils sont issus d'une certaine expérience et ont pour but de faciliter le développement de grosses applications avec eForth Linux.

Tous les fichiers sources de votre projet sont sur votre ordinateur dans l'environnement Linux. Il est conseillé d'avoir un sous-dossier dédié à ce projet. Par exemple, vous travaillez sur un jeu nommé rubik, vous créez donc un répertoire nommé **rubik**.

Concernant les extensions des noms de fichiers, nous conseillons d'utiliser l'extension **fs**.

L'édition des fichiers sur ordinateur est réalisée avec n'importe quel éditeur de fichiers texte, **gedit** sous Linux.

Dans ces fichiers sources, ne pas utiliser de caractère non inclus dans les caractères du code ASCII. Certains codes étendus peuvent perturber la compilation des programmes.

Organiser ses fichiers

Dans la suite, tous nos fichiers auront l'extension **fs**.

Partons de notre répertoire **rubik** sur notre ordinateur.

Le premier fichier que nous allons créer dans ce répertoire sera le fichier **main.fs**. Ce fichier contiendra les appels à chargement de tous les autres fichiers de notre application en cours de développement.

Exemple de contenu de notre fichier **main.fs**:

```
\ RUBIK game main file
s" config.fs" included
```

En phase de développement, le contenu de ce fichier **main.fs** sera chargé depuis un fichier **RUBIK.fs** placé dans le même dossier que eForth et contenant ceci :

```
cd rubik
s" main.fs" included
```

Ceci provoque l'exécution du contenu de notre fichier **main.fs**. Le chargement des autres fichiers sera exécuté depuis ce fichier **main.fs**. Ici on exécute le chargement du fichier **config.fs** dont voici un extrait:

```
0 value MAX_DEPTH
3 constant CUBE_SIZE
```

Dans ce fichier **config.fs** on mettra toutes les valeurs constantes et divers paramètres globaux utilisés par les autres fichiers.

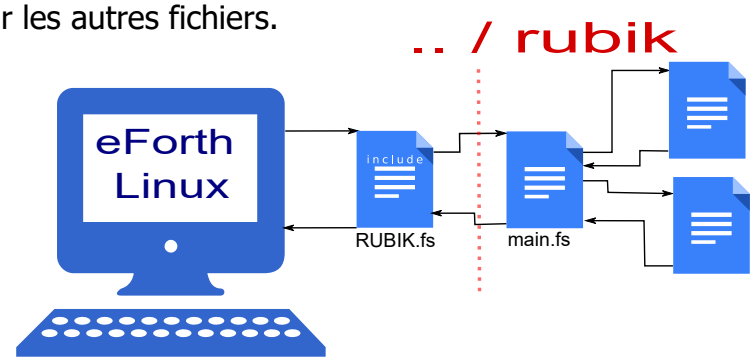


Figure 13: enchaînement des fichiers du projet RUBIK

Il est conseillé de mettre tous les fichiers d'un même projet dans le dossier de ce projet, ici **rubik** pour notre exemple.

Enchaînement des fichiers

Chaque fichier peut faire appel à un fichier avec le mot `included`. Voici un exemple de hiérarchie de fichiers ainsi inclus :

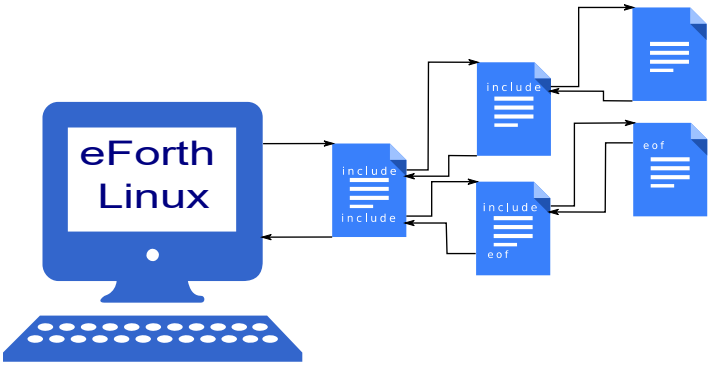


Figure 14: enchaînement de fichiers

Ici, eForth appelle un premier fichier. Même si c'est faisable, il est déconseillé de réaliser des enchaînements en cascade. Préférez une succession de chargement de fichiers depuis **main.fs.** Exemple :

```
DEFINED? --tempusFugit [if] forget --tempusFugit  [then]
create --tempusFugit
s" strings.fs"          included
s" RTClock.fs"          included
s" clepsydra.fs"        included
s" config.fs"           included
s" dispTools.fs"        included
```

Dans cette succession de fichiers, on utilise le fichier **strings.fs**. C'est un fichier dit *outil*. C'est la copie d'un fichier d'usage assez général et dont le contenu étend le dictionnaire FORTH.

En travaillant avec une copie du fichier d'origine, on peut y apporter des corrections ou des améliorations sans risquer d'altérer le fonctionnement du code dans le fichier d'origine. Si ces modifications sont consolidées, on peut les transférer dans le fichier d'origine.

Pour chaque fichier de code source FORTH, datez les versions. Ça vous permettra de retrouver la chronologie des modifications de code.

Conclusion

Les fichiers enregistrés dans le système Linux sont disponibles de manière permanente. Si vous accédez à une version Linux dans un système de gestion WSL2 depuis Windows, ces fichiers seront aussi accessibles au système de fichiers Windows.

Commentaires et mise au point

Il n'existe pas d'IDE[4] 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 :

```
: inGrid? { n gridPos -- fl } 0 { fl } gridPos getGridAddr for aft
getNumber n = if -1 to fl then then next drop  fl ;
```

Ce code sera parfaitement compilé par eForth Linux. 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 `inGrid?`. Eforth Linux permet d'écrire des noms de mots très longs. La taille des mots définis 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: `grid.test.number`
- à la manière CamelCoding `gridTestNumber`
- pour programmeur voulant un code très compréhensible `is-number-in-the-grid`
- programmeur qui aime le code concis `gtn?`

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

- constantes en caractères majuscules `LOTTO_NUMBERS_IN_GRID`
- mot de définition d'autres mots `lottoNumber:` mot suivi de deux points ;
- mot de transformation d'adresse `>date`, ici le paramètre d'adresse est incrémenté d'une certaine valeur pour pointer sur la donnée adéquate ;
- mot de stockage mémoire `date@` ou `date!`

4 Integrated Development Environment = Environnement de Développement Intégré

- Mot d'affichage de donnée `.date`

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

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

ou

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

Indentation du code source

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

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

Exemple :

```
: inGrid? { n gridPos -- fl }
    0 { fl }
    gridPos getGridAddr
    for
        aft
            getNumber n =
            if
                -1 to fl
            then
        then
    next
    drop
    fl
  ;
```

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

```
: inGrid? { n gridPos -- fl }
    0 { fl }   gridPos getGridAddr
    for aft
            getNumber n =
            if -1 to fl  then
```

```
        then
    next
    drop  fl
  ;
```

C'est d'ailleurs souvent le cas avec des structures `case of endof endcase` ;

```
: socketError ( -- )
    errno dup
    case
        2 of     ." No such file "                  endof
        5 of     ." I/O error "                     endof
        9 of     ." Bad file number "               endof
       22 of     ." Invalid argument "              endof
    endcase
    . quit
  ;
```

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 eForth Linux, les données de pile occupent 8 octets. Ce sont donc des entiers au format 64 bits. Alors on met quoi sur la pile de données ? Avec eForth Linux, ce seront **TOUJOURS DES DONNEES 64 BITS** ! Un exemple avec le mot `c!` :

```
create myDelemiter
    0 c,
64 myDelimiter c!   ( c addr -- )
```

Ici, le paramètre `c` indique qu'on empile une valeur entière au format 64 bits, mais dont la valeur sera toujours comprise dans l'intervale [0..255].

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

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

```
create myDelemiter
    0 c,
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.
  Non utilisé avec eForth Linux qui est déjà au format 64 bits ;
- **fl** indique une valeur booléenne, 0 ou non zéro ;
- **n** indique un entier. Entier signé 64 bits pour eForth Linux;
- **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 to SSD1306 connected via I2C bus
        sendDatasToSSD1306
  ;
```

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 OLED SSD1306 128x32 display
\    Filename:      SSD10306commands.fs
\    Date:          21 may 2023
\    Updated:       21 may 2023
\    File Version:  1.0
\    MCU:           ESP32-WROOM-32
\    Forth:         ESP32forth 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.

Outils de diagnostic et mise au point

Le premier outil concerne l'alerte de compilation ou d'interprétation :

```
3 5 25 --> : TEST ( ---)
 ok
3 5 25 -->     [ HEX ] ASCII A DDUP    \ DDUP don't exist
```

Ici, le mot **DDUP** n'existe pas. Toute compilation après cette erreur sera vouée à l'échec.

Le décompilateur

Dans un compilateur conventionnel, le code source est transformé en code exécutable 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 eForth Linux, le développeur peut décompiler ses définitions. Pour décompiler un mot, il suffit de taper **see** suivi du mot à décompiler :

```
: C>F ( øC --- øF) \ Conversion Celsius in Fahrenheit
    9 5 */ 32 +
  ;
see c>f
\ display:
: C>F
    9 5 */ 32 +
;
```

Beaucoup de mots du dictionnaire FORTH de eForth Linux peuvent être décompilés.

La décompilation de vos mots permet de détecter d'éventuelles erreurs de compilation.

Dump mémoire

Parfois, il est souhaitable de pouvoir voir les valeurs qui sont en mémoire. Le mot `dump` accepte deux paramètres: l'adresse de départ en mémoire et le nombre d'octets à visualiser :

```
create myDATAS 01 c, 02 c, 03 c, 04 c,
hex
myDATAS 4 dump     \ displays :
3FFEE4EC                                                01 02 03 04
```

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.

Réaliser des tests unitaires

eForth dispose du mot **assert** permettant de réaliser des tests. Le meilleur endroit pour exploiter ce mot, c'est dans un fichier **tests.fs**. Exemple :

```
$1234 100div  nip  $34  = assert
$1234 100div  drop $12  = assert
```

Ici, on teste le mot **100div** qui laisse sur la pile le quotient et le reste de la division par 256 (100 en hexadécimal). Le test doit laisser une valeur vraie ou fausse sur la pile. Si le test restitue une valeur nulle, **assert** génère un message **ERROR**.

Voici un autre exemple exploitant **assert** :

```
$0080 bytesToUTF8 $c280   = assert
$0544 bytesToUTF8 $d584   = assert
$a894 bytesToUTF8 $eaa294 = assert
```

Ici, on teste le mot **bytesToUTF8**. Ce mot provient d'un code en cours de développement. Les valeurs à tester proviennent de la documentation UTF8 en ligne. Ces trois lignes permettent de tester instantanément **bytesToUTF8** avec plusieurs cas types. Si le mot ne génère pas le résultat attendu, **assert** signalera qu'il y a une erreur de test.

Création et utilisation de assert(

Le mot **assert** a un inconvénient majeur. Si on effectue beaucoup de tests sur différents mots, ce dans un fichier, ici t**ests.fs**, on récupère seulement un signalement d'erreur, mais pas d'information sur la ligne de test qui a généré cette erreur.

Il se trouve que la version *gForth* dispose du mot **assert(**, dont la syntaxe d'utilisation est :

```
assert( 0 >gray  0 = )
assert( 1 >gray  1 = )
assert( 2 >gray  3 = )
```

Le code *gForth* a été adapté pour afficher le contenu erroné. Voici le code source de cette version :

```
-1 value ASSERT_LEVEL

variable assert-start

: assert(   ( -- )
    tib >in @ + assert-start !
    ASSERT_LEVEL 0= if
        POSTPONE (
    then
  ; immediate

: ) ( fl -- )
    0= if
        cr ." ASSERT : "
        assert-start @
        tib >in @ + over - 1- type
         -1 throw
    then
  ; immediate
```

Dans ce code, on a une valeur **ASSERT_LEVEL**. Si cette valeur est mise à zéro, **assert(** se comporte comme le mot (.

Ensuite, on a une variable **assert-start**. Cette variable sert à mémoriser l'emplacement de **assert(** dans la chaîne d'interprétation traitée par eForth.

Le mot **)** teste le flag booléen. S'il est égal à zéro, il génère un message d'erreur et affiche le code situé après **assert(** qui est à l'origine de l'erreur de test.

Si vous êtes sur un projet de développement, voici un exemple de chaînage type des fichiers dans **main.fs** :

```
s" gray.fs"     included
s" assert.fs"   included
s" tests.fs"    included
```

Le fichier **gray.fs** contient le code FORTH en cours de développement et de mise au point. Le fichier **assert.fs** contient le code de `assert(`. Enfin, notre fichier **tests.fs** contient la batterie de tests à réaliser sur les définitions en cours de mise au point.

Ainsi, avec une simple séquence **`include main.fs`**, le code en cours de développement est compilé, puis il est instantanément testé au travers des tests unitaires écrits dans **tests.fs**.

Le mot **`assert(`** a été écrit pour ne rien afficher si les tests ont été exécutés avec succès.

Cette stratégie de développement avec tests unitaires permet de détecter rapidement des erreurs de code si vous modifiez une définition qui est soumise aux tests unitaires.

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, eForth Linux construit le nombre dans le dictionnaire l'espace alloué pour le nouveau mot, suivant le code spécial qui met le numéro stocké sur la pile chaque fois que le mot est exécuté. 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 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 ok<#,ram>
```

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.

Gestion du dictionnaire

Le mot `forget` suivi du mot à supprimer enlèvera toutes les entrées de dictionnaire que vous avez faites depuis ce mot:

```
: test1 ;
: test2 ;
: test3 ;
forget test2  \ efface test2 et test3 du dictionnaire
```

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. eForth Linux 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
```

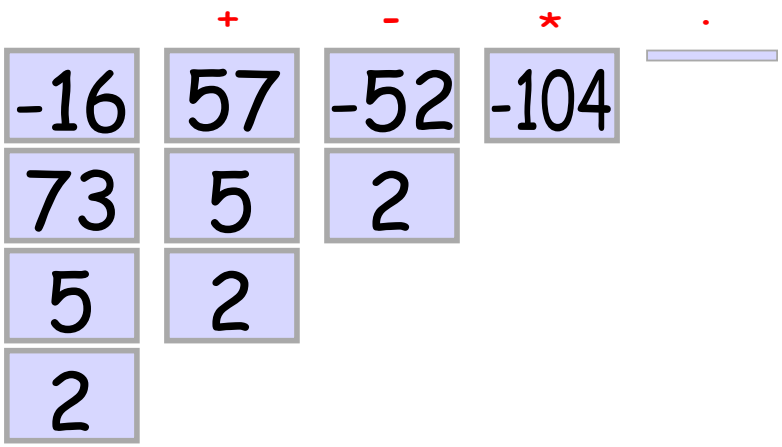
laisse la pile de paramètres dans l'état

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

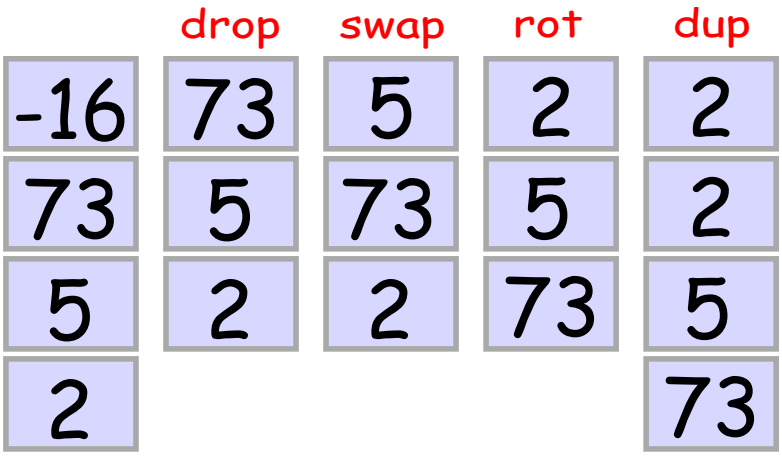
Notez que eForth Linux affiche commodément les éléments de la pile lors de l'interprétation de chaque ligne et que la valeur de -16 est affichée sous la forme d'entier non signé 64 bits. En outre, le mot `.` consomme la valeur de données -104, laissant la pile vide. Si nous exécutons . sur la pile maintenant vide, l'interpréteur externe abandonne avec une erreur de pointeur de pile STACK UNDERFLOW ERROR.

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, eForth Linux 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



actions sont présentées ci-dessous.

La pile de retour et ses utilisations

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

En plus de servir de réservoir d'adresses de retour, l'utilisateur peut également stocker et récupérer à partir de la pile de retour, mais cela doit être fait avec soin car la pile de retour est essentielle à l'exécution du programme. Si vous utilisez la pile de retour pour le

stockage temporaire, vous devez la remettre dans son état d'origine, sinon vous ferez probablement planter le système eForth Linux. 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. Pour supprimer simplement une valeur du haut de la pile, il y a le mot `rdrop`. Le mot `r@` copie le haut de la pile de retour dans la pile de paramètres.

Utilisation de la mémoire

Dans eForth Linux, les nombres 64 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. cellules de mémoire 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
```

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 un constante est la valeur des données restant sur la pile.

```
\ définit les pins VSPI
19 constant VSPI_MISO
23 constant VSPI_MOSI
18 constant VSPI_SCLK
05 constant VSPI_CS

\ définit la fréquence du port SPI
4000000 constant SPI_FREQ

\ sélectionne le vocabulaire SPI
only FORTH  SPI also

\ initialise le port SPI
: init.VSPI ( -- )
    VSPI_CS OUTPUT pinMode
    VSPI_SCLK VSPI_MISO VSPI_MOSI VSPI_CS SPI.begin
    SPI_FREQ SPI.setFrequency
  ;
```

Valeurs pseudo-constantes

Une valeur définie avec value est un type hybride de variable et constante. Nous définissons et initialisons une valeur et 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` et non d'autre chose.

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` poussera 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 variables locales avec eForth Linux

Introduction

Le langage FORTH traite les données essentiellement par la pile de données. Ce mécanisme très simple offre une performance inégalée. A contrario, suivre le cheminement des données peut rapidement devenir complexe. Les variables locales offrent une alternative intéressante.

Le faux commentaire de pile

Si vous suivez les différents exemples FORTH, vous avez noté les commentaires de pile encadrés par `(` et `)`. Exemple:

```
\ addition deux valeurs non signées, laisse sum et carry sur la pile
: um+ ( u1 u2 -- sum carry )
    \ ici la définition
  ;
```

Ici, le commentaire `( u1 u2 -- sum carry )` n'a absolument aucune action sur le reste du code FORTH. C'est un pur commentaire.

Quand on prépare une définition complexe, la solution est d'utiliser des variables locales encadrées par `{` et `}`. Exemple:

```
: 2OVER { a b c d }
    a b c d a b
  ;
```

On définit quatre variables locales `a b c` et `d`.

Les mots `{` et `}` ressemblent aux mots `(` et `)` mais n'ont pas du tout le même effet. Les codes placés entre `{` et `}` sont des variables locales. Seule contrainte: ne pas utiliser de noms de variables qui pourraient être des mots FORTH du dictionnaire FORTH. On aurait aussi bien pu écrire notre exemple comme ceci:

```
: 2OVER { varA varB varC varD }
    varA varB varC varD varA varB
  ;
```

Chaque variable va prendre la valeur de la donnée de pile dans l'ordre de leur dépôt sur la pile de données. ici, 1 va dans `varA`, 2 dans `varB`, etc..:

```
--> 1 2 3 4
 ok
1 2 3 4 --> 2over
 ok
1 2 3 4 1 2 -->
```

Notre faux commentaire de pile peut être complété comme ceci:

```
: 2OVER { varA varB varC varD -- varA varB varC varD varA varB }
  ......
```

Les caractères qui suivent `--` n'ont pas d'effet. Le seul intérêt est de rendre notre faux commentaire semblable à un vrai commentaire de pile.

Action sur les variables locales

Les variables locales agissent exactement comme des pseudo-variables définies par value. Exemple:

```
: 3x+1 { var -- sum }
    var 3 * 1 +
  ;
```

A le même effet que ceci:

```
0 value var
: 3x+1 ( var -- sum )
    to var
    var 3 * 1 +
  ;
```

Dans cet exemple, var est défini explicitement par value.

On affecte une valeur à une variable locale avec le mot `to` ou `+to` pour incrémenter le contenu d'une variable locale. Dans cet exemple, on rajoute une variable locale `result` initialisée à zéro dans le code de notre mot:

```
: a+bEXP2 { varA varB -- (a+b)EXP2 }
    0 { result }
    varA varA *      to result
    varB varB *     +to result
    varA varB * 2 * +to result
    result
  ;
```

Est-ce que ce n'est pas plus lisible que ceci?

```
: a+bEXP2 ( varA varB -- result )
    2dup
    * 2 * >r
    dup *
    swap dup * +
    r> +
  ;
```

Voici un dernier exemple, la définition du mot `um+` qui additionne deux entiers non signés et laisse sur la pile de données la somme et la valeur de débordement de cette somme:

```
\ addition deux entiers non signés, laisse sum et carry sur la pile
```

```
: um+ { u1 u2 -- sum carry }
    0 { sum }
    cell for
        aft
            u1 $100 /mod to u1
            u2 $100 /mod to u2
            +
            cell 1- i - 8 * lshift  +to sum
        then
    next
    sum
    u1 u2 + abs
  ;
```

Voici un exemple plus complexe, la réécriture de **DUMP** en exploitant des variables locales:

```
\ variables locales dans DUMP:
\ START_ADDR      \ première addresse pour dump
\ END_ADDR        \ dernière addresse pour dump
\ 0START_ADDR     \ première adresse pour la boucle dans dump
\ LINES           \ nombre de lignes pour la boucle dump
\ myBASE          \ base numérique courante
internals
: dump ( start len -- )
    cr cr ." --addr---  "
    ." 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  ------chars-----"
    2dup + { END_ADDR }             \ store latest address to dump
    swap { START_ADDR }             \ store START address to dump
    START_ADDR 16 / 16 * { 0START_ADDR } \ calc. addr for loop start
    16 / 1+ { LINES }
    base @ { myBASE }               \ save current base
    hex
    \ outer loop
    LINES 0 do
        0START_ADDR i 16 * +        \ calc start address for current line
        cr <# # # # #  [char] - hold # # # # #> type
        space space     \ and display address
        \ first inner loop, display bytes
        16 0 do
            \ calculate real address
            0START_ADDR j 16 * i + +
            ca@ <# # # #> type space \ display byte in format: NN
        loop
        space
        \ second inner loop, display chars
        16 0 do
            \ calculate real address
            0START_ADDR j 16 * i + +
```

```
            \ display char if code in interval 32-127
            ca@     dup 32 < over 127 > or
            if      drop [char] . emit
            else    emit
            then
        loop
    loop
    myBASE base !               \ restore current base
    cr cr
  ;
forth
```

L'emploi des variables locales simplifie considérablement la manipulation de données sur les piles. Le code est plus lisible. On remarquera qu'il n'est pas nécessaire de pré-déclarer ces variables locales, il suffit de les désigner au moment de les utiliser, par exemple: **`base @ { myBASE }`**.

ATTENTION: si vous utilisez des variables locales dans une définition, n'utilisez plus les mots **`>r`** et **`r>`**, sinon vous risquez de perturber la gestion des variables locales. Il suffit de regarder la décompilation de cette version de **`DUMP`** pour comprendre la raison de cet avertissement:

```
: dump  cr cr s" --addr---  " type
    s" 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  ------chars-----" type
    2dup + >R SWAP >R -4 local@ 16 / 16 * >R 16 / 1+ >R base @ >R
    hex -8 local@ 0 (do) -20 local@ R@ 16 * + cr
    <# # # # # 45 hold # # # # #> type space space
    16 0 (do) -28 local@ j 16 * R@ + + CA@ <# # # #> type space 1 (+loop)
    0BRANCH rdrop rdrop space 16 0 (do) -28 local@ j 16 * R@ + + CA@ DUP 32 < OVER 127 > OR
    0BRANCH DROP 46 emit BRANCH emit 1 (+loop) 0BRANCH rdrop rdrop 1 (+loop)
    0BRANCH rdrop rdrop -4 local@ base ! cr cr rdrop rdrop rdrop rdrop rdrop ;
```

Structures de données pour eForth Linux

Préambule

eForth Linux est une version 64 bits du langage FORTH. Ceux qui ont pratiqué FORTH depuis ses débuts ont programmé avec des versions 16 ou 32 bits. 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 `cell`. Exécution de ce mot pour eForth Linux:

```
cell .  \ affiche 8
```

La valeur 8 signifie que la taille des éléments déposés sur la pile de donnéees est de 8 octets, soit 8x8 bits = 64 bits.

Avec une version FORTH 16 bits, `cell` empilera la valeur 2. De même, si vous utilisez une version 32 bits, cell empilera la valeur 4.

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 32 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 cell *        \ 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 )
    cell * 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 cell * +
  ;
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 quatre fois moins d'espace mémoire.

Gestion de structures complexes

eForth Linux dispose du vocabulaire structures. Le contenu de ce vocabulaire permet de définir des structures de données complexes.

Voici un exemple trivial de structure :

```
structures
```

```
struct YMDHMS
    ptr field >year
    ptr field >month
    ptr field >day
    ptr field >hour
    ptr field >min
    ptr field >sec
```

Ici, on définit la structure YMDHMS. Cette structure gère les pointeurs `>year >month >day >hour >min` et `>sec`.

Le mot `YMDHMS` a comme seule utilité d'initialiser et regrouper les pointeurs dans la structure complexe. Voici comment sont utilisés ces pointeurs:

```
create DateTime
    YMDHMS allot

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

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

DateTime .date
```

On a défini le mot `DateTime` qui est un tableau simple de 6 cellules 64 bits consécutives. L'accès à chacune des cellules est réalisée par l'intermédiaire du pointeur correspondant. On peut redéfinir l'espace alloué de notre structure `YMDHMS` en utilisant le mot `i8` pour pointer des octets:

```
structures
struct cYMDHMS
    ptr field >year
    i8  field >month
    i8  field >day
    i8  field >hour
    i8  field >min
```

```
    i8  field >sec

create cDateTime
    cYMDHMS allot

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

: .cDate ( 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
  ;
cDateTime .cDate    \ affiche:
\  YEAR: 2022
\ MONTH: 3
\   DAY: 21
\    HH: 22
\    MM: 36
\    SS: 15
```

Dans cette structure cYMDHMS, on a gardé l'année au format 64 bits et réduit toutes les autres valeurs à des entiers 8 bits. On constate, dans le code de .cDate, que l'utilisation des pointeurs permet un accès aisé à chaque élément de notre structure complexe....

Les nombres réels avec eForth Linux

Si on teste l'opération `1 3 /` en langage FORTH, le résultat sera 0.

Ce n'est pas surprenant. De base, eForth Linux n'utilise que des nombres entiers 64 bits via la pile de données. Les nombres entiers offrent certains avantages :

- rapidité de traitement ;
- résultat de calculs sans risque de dérive en cas d'itérations ;
- conviennent à quasiment toutes les situations.

Même en calculs trigonométriques, on peut utiliser une table d'entiers. Il suffit de créer un tableau avec 90 valeurs, où chaque valeur correspond au sinus d'un angle, multiplié par 1000.

Mais les nombres entiers ont aussi des limites :

- résultats impossibles pour des calculs de division simple, comme notre exemple 1/3;
- nécessite des manipulations complexes pour appliquer des formules de physique.

Depuis la version 7.0.6.5, eForth Linux intègre des opérateurs traitant des nombres réels.

Les nombres réels sont aussi dénommés nombres à virgule flottante.

Les réels avec eForth Linux

Afin de distinguer les nombres réels, il faut les terminer avec la lettre "e":

```
3           \ empile 3 sur la pile de données
3e          \ empile 3 sur la pile des réels
5.21e f.    \ affiche 5.210000
```

C'est le mot `f.` qui permet d'afficher un nombre réel situé au sommet de la pile des réels.

Precision des nombres réels avec eForth Linux

Le mot `set-precision` permet d'indiquer le nombre de décimales à afficher après le point décimal. Voyons ceci avec la constante `pi`:

```
pi f.       \ affiche 3.141592
4 set-precision
pi f.       \ affiche 3.1415
```

La précision limite de traitement des nombres réels avec eForth Linux est de six décimales :

```
12 set-precision
```

```
1.987654321e f.       \ affiche 1.987654668777
```

Si on réduit la précision d'affichage des nombres réels en dessous de 6, les calculs seront quand même réalisés avec une précision à 6 décimales.

Constantes et variables réelles

Une constante réelle est définie avec le mot **fconstant**:

```
0.693147e fconstant ln2   \ logarithme naturel de 2
```

Une variable réelle est définie avec le mot **fvariable**:

```
fvariable intensity
170e 12e F/ intensity SF!   \ I=P/U   ---  P=170w   U=12V
intensity SF@ f.            \ affiche 14.166669
```

ATTENTION: tous les nombres réels transitent par la **pile des nombres réels**. Dans le cas d'une variable réelle, seule l'adresse pointant sur la valeur réelle transite par la pile de données.

Le mot **SF!** enregistre une valeur réelle à l'adresse ou la variable pointée par son adresse mémoire. L'exécution d'une variable réelle dépose l'adresse mémoire sur la pile données classique.

Le mot **SF@** empile la valeur réelle pointée par son adresse mémoire.

Opérateurs arithmétiques sur les réels

eForth Linux dispose de quatre opérateurs arithmétiques **F+ F- F* F/**:

```
1.23e 4.56e F+ f.   \ affiche 5.790000      1.23-4.56
1.23e 4.56e F- f.   \ affiche -3.330000     1.23-4.56
1.23e 4.56e F* f.   \ affiche 5.608800      1.23*4.56
1.23e 4.56e F/ f.   \ affiche 0.269736      1.23/4.56
```

eForth Linux dispose aussi de ces mots :

- **1/F** calcule l'inverse d'un nombre réel;
- **fsqrt** calcule la racine carrée d'un nombre réel.

```
5e 1/F f.       \ affiche 0.200000      1/5
5e fsqrt f.     \ affiche 2.236068      sqrt(5)
```

Opérateurs mathématiques sur les réels

eForth Linux dispose de plusieurs opérateurs mathématiques :

- **F**** élève un réel r_val à la puissance r_exp
- **FATAN2** calcule l'angle en radian à partir de la tangente.

- **FCOS** (r1 -- r2) Calcule le cosinus d'un angle exprimé en radians.
- **FEXP** (ln-r -- r) calcule le réel correspondant à e EXP r
- **FLN** (r -- ln-r) calcule le logarithme naturel d'un nombre réel.
- **FSIN** (r1 -- r2) calcule le sinus d'un angle exprimé en radians.
- **FSINCOS** (r1 -- rcos rsin) calcule le cosinus et le sinus d'un angle exprimé en radians.

Quelques exemples :

```
  2e 3e f** f.     \ affiche 8.000000
  2e 4e f** f.     \ affiche 16.000000
 10e 1.5e f** f.   \ affiche 31.622776

4.605170e FEXP F.     \ affiche 100.000018

pi 4e f/
FSINCOS f. f.    \ affiche 0.707106 0.707106
pi 2e f/
FSINCOS f. f.    \ affiche 0.000000 1.000000
```

Opérateurs logiques sur les réels

eForth Linux permet aussi d'effectuer des tests logiques sur les réels :

- **F0<** (r -- fl) teste si un nombre réel est inférieur à zéro.
- **F0=** (r -- fl) indique vrai si le réel est nul.
- **f<** (r1 r2 -- fl) fl est vrai si r1 < r2.
- **f<=** (r1 r2 -- fl) fl est vrai si r1 <= r2.
- **f<>** (r1 r2 -- fl) fl est vrai si r1 <> r2.
- **f=** (r1 r2 -- fl) fl est vrai si r1 = r2.
- **f>** (r1 r2 -- fl) fl est vrai si r1 > r2.
- **f>=** (r1 r2 -- fl) fl est vrai si r1 >= r2.

Transformations entiers ↔ réels

eForth Linux dispose de deux mots pour transformer des entiers en réels et inversement :

- **F>S** (r -- n) convertit un réel en entier. Laisse sur la pile de données la partie entière si le réel a des parties décimales.
- **S>F** (n -- r: r) convertit un nombre entier en nombre réel et transfère ce réel sur la pile des réels.

Exemple :

```
35 S>F
F.   \ affiche 35.000000

3.5e F>S .   \ affiche 3
```

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. 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.

eForth Linux 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 eForth Linux;
- **HEX** pour sélectionner la base numérique hexadécimale ;
- **BINARY** pour sélectionner la base numérique binaire.

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 :

```
: SEXTAL ( ---)         \ sélectionne la base numérique binaire
    6 BASE ! ;
DECIMAL 255 SEXTAL .    \ affiche     1103
```

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.

Définition de nouveaux formats d'affichage

Forth dispose de primitives permettant d'adapter l'affichage d'un nombre à un format quelconque. Avec eForth Linux, 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 ---)
  <# # # [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 **<#** 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
    <# :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
    <#
    4 0 DO                  \ Format Loop
        FOUR-DIGITS
    LOOP
    #> TYPE SPACE           \ Binary display
```

```
    R> BASE ! ;             \ Initial digital base restoration
```

Exemple d'exécution :

```
DECIMAL 12 AFB    \ affiche     0000 0000 0000 0110
HEX 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   \ display : 06.18.05.12.54
```

Cet agenda, 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 ROT 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 données" ;
```

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 données
```

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
```

Variables chaînes de caractères

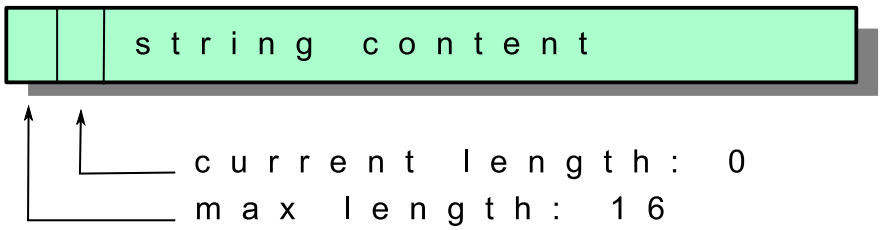
Les variables alpha-numérique texte n'existent pas nativement dans eForth Linux. Voici le premier essai de définition du mot **string** :

```
\ define a strvar
: string  ( comp: n --- names_strvar | exec: --- addr len )
    create
        dup
        c,       \ n is maxlength
        0 c,     \ 0 is real length
        allot
    does>
        2 +
        dup 1 - c@
  ;
```

Une variable chaîne de caractères se définit comme ceci :

```
16 string strState
```

Voici comment est organisé l'espace mémoire réservé pour cette variable texte :



Code des mots de gestion de variables texte

Voici le code source complet permettant la gestion des variables texte :

```
DEFINED? --str [if] forget --str  [then]
create --str


\ compare two strings
: $= ( addr1 len1 addr2 len2 --- fl)
    str=
  ;

\ define a strvar
: string ( n --- names_strvar )
    create
        dup
        ,                   \ n is maxlength
```

```
        0 ,                 \ 0 is real length
        allot
    does>
        cell+ cell+
        dup cell - @
    ;

\ get maxlength of a string
: maxlen$  ( strvar --- strvar maxlen )
    over cell - cell - @
    ;

\ store str into strvar
: $!  ( str strvar --- )
    maxlen$                 \ get maxlength of strvar
    nip rot min             \ keep min length
    2dup swap cell - !      \ store real length
    cmove                   \ copy string
    ;

\ Example:
\ : s1
\     s" this is constant string" ;
\ 200 string test
\ s1 test $!

\ set length of a string to zero
: 0$! ( addr len -- )
    drop 0 swap cell - !
  ;

\ extract n chars right from string
: right$  ( str1 n --- str2 )
    0 max over min >r + r@ - r>
    ;

\ extract n chars left frop string
: left$  ( str1 n --- str2 )
    0 max min
    ;

\ extract n chars from pos in string
: mid$  ( str1 pos len --- str2 )
    >r over swap - right$ r> left$
    ;

\ append char c to string
: c+$!  ( c str1 -- )
```

```
    over >r
    + c!
    r> cell - dup @ 1+ swap !
    ;

\ work only with strings. Don't use with other arrays
: input$ ( addr len -- )
    over swap maxlen$ nip accept
    swap cell - !
  ;
```

La création d'une chaîne de caractères alphanumérique est très simple :

```
64 string myNewString
```

Ici, nous créons une variable alphanumérique `myNewString` pouvant contenir jusqu'à 64 caractères.

Pour afficher le contenu d'une variable alphanumérique, il suffit ensuite d'utiliser `type`. Exemple :

```
s" This is my first example.." myNewString $!
myNewString type   \ display: This is my first example..
```

Si on tente d'enregistrer une chaîne de caractères plus longue que la taille maximale de notre variable alphanumérique, la chaîne sera tronquée :

```
s" This is a very long string, with more than 64 characters. It can't store
complete"
myNewString $!
myNewString type
  \ affiche: This is a very long string, with more than 64 characters. It
can
```

Ajout de caractère à une variable alphanumérique

Certains périphériques, le transmetteur LoRa par exemple, demandent à traiter des lignes de commandes contenant les caractères non alphanumériques Le mot `c+$!` permet cette insertion de code :

```
32 string AT_BAND
s" AT+BAND=868500000" AT_BAND $!  \ set frequency at 865.5 Mhz
$0a AT_BAND c+$!
$0d AT_BAND c+$!    \ add CR LF code at end of command
```

Le dump mémoire du contenu de notre variable alphanumérique `AT_BAND` confirme la présence des deux caractères de contrôle en fin de chaine :

```
--> AT_BAND dump
--addr---  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  ------chars-----
3FFF-8620  8C 84 FF 3F 20 00 00 00 13 00 00 00 41 54 2B 42  ...? .......AT+B
3FFF-8630  41 4E 44 3D 38 36 38 35 30 30 30 30 30 0A 0D BD  AND=868500000...
 ok
```

Voici une manière astucieuse de créer une variable alphanumérique permettant de transmettre un retour chariot, un **CR+LF** compatible avec les fins de commandes pour le transmetteur LoRa:

```
2 string $crlf
$0d $crlf c+$!
$0a $crlf c+$!

: crlf ( -- )       \ same action as cr, but adapted for LoRa
    $crlf type
  ;
```

Les mots à action différée

Les mots à action différée sont définis par le mot de définition **defer**. Pour en comprendre les mécanismes et l'intérêt à exploiter ce type de mot, voyons plus en détail le fonctionnement de l'interpréteur interne du langage FORTH.

Toute définition compilée par : (deux-points) contient une suite d'adresses codées correspondant aux champs de code des mots précédemment compilés. Au cœur du système FORTH, le mot **EXECUTE** admet comme paramètre ces adresses de champ de code, adresses que nous abrégerons par **cfa** pour Code Field Address. Tout mot FORTH a un **cfa** et cette adresse est exploitée par l'interpréteur interne de FORTH:

```
' <mot>
\ dépose le cfa de <mot> sur la pile de données
```

Exemple:

```
' WORDS
\ empile le cfa de WORDS.
```

A partir de ce **cfa**, connu comme seule valeur littérale, l'exécution du mot peut s'effectuer avec **EXECUTE**:

```
' WORDS EXECUTE
\ exécute WORDS
```

Bien entendu, il aurait été plus simple de taper directement **WORDS**. A partir du moment où un **cfa** est disponible comme seule valeur littérale, il peut être manipulé et notamment stocké dans une variable:

```
variable vector
' WORDS vector !
vector @ .
\ affiche cfa de WORDS stocké dans la variable vector
```

On peut exécuter **WORDS** indirectement depuis le contenu de **vector** :

```
vector @ EXECUTE
```

Ceci lance l'exécution du mot dont le **cfa** a été stocké dans la variable **vector** puis remis sur la pile avant utilisation par **EXECUTE**.

C'est un mécanisme similaire qui est exploité par la partie exécution du mot de définition **defer**. Pour simplifier, **defer** crée un en-tête dans le dictionnaire, à la manière de **variable** ou **constant**, mais au lieu de déposer simplement une adresse ou une valeur sur la pile, il lance l'exécution du mot dont le **cfa** a été stocké dans la zone paramétrique du mot défini par **defer**.

Définition et utilisation de mots avec defer

L'intialisation d'un mot défini par `defer` est réalisée par `is` :

```
defer vector
' words is vector
```

L'exécution de `vector` provoque l'exécution du mot dont le **cfa** a été précédemment affecté:

```
vector       \ exécute   words
```

Un mot créé par `defer` sert à exécuter un autre mot sans faire appel explicitement à ce mot. Le principal intérêt de ce type de mot réside surtout dans la possibilité de modifier le mot à exécuter:

```
' page is vector
```

`vector` exécute maintenant `page` et non plus `words`.

On utilise essentiellement les mots définis par `defer` dans deux situations:

- définition d'une référence avant ;
- définition d'un mot dépendant du contexte d'exploitation.

Dans le premier cas, la définition d'une référence avant permet de surmonter les contraintes de la sacro-sainte précédence des définitions.

Dans le second cas, la définition d'un mot dépendant du contexte d'exploitation permet de résoudre la plupart des problèmes d'interfaçage avec un environnement logiciel évolutif, de conserver la portabilité des applications, d'adapter le comportement d'un programme à des situations contrôlées par divers paramètres sans nuire aux performances logicielles.

Définition d'une référence avant

Contrairement à d'autres compilateurs, FORTH n'autorise pas la compilation d'un mot dans une définition avant qu'il ne soit défini. C'est le principe de la précédence des définitions:

```
: word1 ( ---)    word2    ;
: word2 ( ---)   ;
```

Ceci génère une erreur à la compilation de `word1`, car `word2` n'est pas encore défini. Voici comment contourner cette contrainte avec `defer` :

```
defer word2
: word1 ( ---)    word2    ;
: (word2) ( ---)   ;
' (word2) is  word2
```

Cette fois-ci, `word2` a été compilé sans erreur. Il n'est pas nécessaire d'affecter un cfa au mot d'exécution vectorisée `word2`. Ce n'est qu'après la définition de `(word2)` que la zone paramétrique du `word2` est mise à jour. Après affectation du mot d'exécution vectorisée

`word2`, `word1` pourra exécuter sans erreur le contenu de sa définition. L'exploitation des mots créés par `defer` dans cette situation doit rester exceptionnel.

Un cas pratique

Vous avez une application à créer, avec des affichages en deux langues. Voici une manière astucieuse en exploitant un mot défini par `defer` pour générer du texte en français ou en anglais. Pour commencer, on va simplement créer un tableau des jours en anglais :

```
:noname s" Saterday" ;
:noname s" Friday" ;
:noname s" Thursday" ;
:noname s" Wednesday" ;
:noname s" Tuesday" ;
:noname s" Monday" ;
:noname s" Sunday" ;

create ENdayNames ( --- addr)
      , , , , , , ,
```

Puis on crée un tableau similaire pour les jours en français :

```
:noname s" Samedi" ;
:noname s" Vendredi" ;
:noname s" Jeudi" ;
:noname s" Mercredi" ;
:noname s" Mardi" ;
:noname s" Lundi" ;
:noname s" Dimanche" ;

create FRdayNames ( --- addr)
      , , , , , , ,
```

Enfin on crée notre mot à action différée `dayNames` et la manière de l'initialiser :

```
defer dayNames

: in-ENGLISH
    ['] ENdayNames is dayNames  ;

: in-FRENCH
    ['] FRdayNames is dayNames  ;
```

Voici maintenant les mots permettant de gérer ces deux tableaux :

```
: _getString { array length -- addr len }
    array
    swap cell *
    + @ execute
    length ?dup if
        min
```

```
    then
  ;

10 value dayLength
: getDay ( n -- addr len )        \ n interval [0..6]
    dayNames dayLength _getString
  ;
```

Voici ce que donne l'exécution de `getDay` :

```
in-ENGLISH 3 getDay type cr   \ display :  Wednesday
in-FRENCH  3 getDay type cr   \ display :  Mercredi
```

On définit ici le mot `.dayList` qui affiche le début des noms des jours de la semaine :

```
: .dayList { size -- }
    size to dayLength
    7 0 do
        i getDay type space
    loop
  ;

in-ENGLISH 3 .dayList cr   \ display :  Sun Mon Tue Wed Thu Fri Sat
in-FRENCH  1 .dayList cr   \ display :  D L M M J V S
```

Dans la seconde ligne, nous n'affichons que la première lettre de chaque jour de la semaine.

Dans cet exemple, nous exploitons `defer` pour simplifier la programmation. En développement web, on utiliserait des *templates* pour gérer des sites multilingues. En FORTH, on déplace simplement un vecteur dans un mot à action différée. Ici nous gérons seulement deux langues. Ce mécanisme peut s'étendre facilement à d'autres langues, car nous avons séparé la gestion des messages textuels de la partie purement applicative.

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 ESP32forth, 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.

Dans le chapitre Structures de données pour ESP32forth, 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 eForth Linux, 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>`", qui est souvent utilisée ensemble pour créer des mots (mots de vocabulaire) personnalisés avec des comportements spécifiques.

Voici comment cela fonctionne en Forth :

- `CREATE` **:** ce mot-clé est utilisé pour créer un nouvel espace de données dans le dictionnaire ESP32Forth. Il prend en charge un argument, qui est le nom que vous donnez à votre nouveau mot ;
- `DOES>` **:** ce mot-clé est utilisé pour définir le comportement du mot que vous venez de créer avec `CREATE`. Il est suivi d'un bloc de code qui spécifie ce que le mot devrait faire lorsqu'il est rencontré pendant l'exécution du programme.

Ensemble, cela ressemble à quelque chose comme ceci :

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

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

```
\ define a register, similar as constant
: defREG:
    create ( addr1 -- <name> )
        ,
    does>  ( -- regAddr )
        @
  ;
```

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

```
$00 value DB2INSTANCE
```

ou

```
$00 defREG: DB2INSTANCE
```

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

- un code source eForth Linux plus lisible. On détecte facilement toutes les constantes nommant un registre ESP32 ;
- on se laisse la possibilité de modifier la partie **does>** de **defREG:** sans avoir ensuite à réécrire les lignes de code qui n'utiliseraient pas **defREG:**

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

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

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

Exemple de gestion de couleur

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

```
0 value currentCOLOR
```

```
\ define word as COLOR constant
: color: ( n -- <name> )
    create
        ,
    does>
        @ to currentCOLOR
  ;

$00 color: setBLACK
$ff color: setWHITE
```

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

```
$00 currentCOLOR !
```

Ou

```
$00 variable 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 fonctionne parfaitement dans eForth Linux :

```
\ Work well in eForth Linux
: 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 eForth Linux et tapez :

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

collez le caractère 早. Eforth Linux 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 ?

Traitement des caractères UTF8

C'est en réalisant quelques tests de saisie de caractères au clavier qu'il est apparu un petit souci. Si on fait ceci :

```
key      \ and press a key, push 97 on stack
```

Jusque là, tout est normal. Mais sur le clavier, on dispose aussi, en France sur clavier AZERTY, de caractères accentués et certains caractères comme **€**. Réessayons **key** en tentant de récupérer le code de ce caractère :

```
key
€
 ok
226 --> ��
ERROR: ��  NOT FOUND!
```

Le premier code récupéré a la valeur 226. mais il y a deux autres codes qui perturbent l'interpréteur FORTH. Voyons cette solution :

```
key key key
€
 ok
226 130 172
```

Ah… ?!?!? Trois codes ?

Le codage UTF8

Reprenons les trois codes **226 130 172** en hexadécimal : **E2 82 AC**. Si on fait ceci :

```
$e2 emit
```

Ca affiche **  ok**. Mmmm…. Vérifions dans un boucle qui est dans l'intervalle 32-255 :

```
: dispChars  ( -- )
    256 32 do
        i emit
    loop
  ;
```

L'exécution de **dispChars** affiche ceci :

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~��������
������������������������������������������������
������������������������������������������������
������������������������ ok
```

eForth Linux a quelques soucis pour afficher des caractères ayant un code ASCII supérieur à 127. Si on refait ce test avec eForth Windows, le même mot **dispChars** affiche ceci :

```
 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~Çüéâäàåçêëèï
îìÄÅÉæÆôöòûùÿÖÜø£Ø×ƒáíóúñÑª°¿®¬½¼¡«»░▒▓│┤ÁÂÀ©╣║╗╝¢
¥┐└┴┬├─┼ãÃ╚╔╩╦╠═╬¤ðÐÊËÈ₁ÍÎÏ┘┌█▄¦Ì▀ÓßÔÒõÕµþÞÚÛÙýÝ¯´±‗¾¶§÷¸°¨·¹³²■  ok
```

Pour les caractères dont le code ASCII est dans l'intervalle [32..127], les caractères sont identiques. Pour les caractères dont le code ASCII est supérieur à 127 (7F en hexadécimal), eForth Linux ne sait pas afficher de caractères valides.

Pour afficher le caractère **€** sous eForth Linux, nous avons des solutions simples :

```
: .eur ( -- )
    ." €" ;
```

ou

```
: .€ ( -- )
    ." €" ;
```

ou

```
: .eur ( -- )
    226 emit  130 emit  172 emit ;
```

Mais nous ne réglons pas le problème des caractères dont le code ASCII est supérieur à 127. Pour résoudre ce problème, il faut s'intéresser au codage UTF8, celui utilisé par eForth Linux.

En codage UTF8, les caractères ASCII sont codés sur 7 bits :

- **0***bbb-bbbb* codage sur 1 octet

Pour tous les autres caractères, le codage est sur 2 ou 3 ou 4 octets :

- **110***b-bbbb* **10***bb-bbbb* codage sur 2 octets
- **1110**-*bbbb* **10***bb-bbbb* **10***bb-bbbb* codage sur 3 octets
- **1111**-**0***bbb* **10***bb-bbbb* **10***bb-bbbb* **10***bb-bbbb* codage sur 4 octets

Pour tous les codes supérieurs à $7F, les premiers bits de poids fort déterminent le nombre d'octets codant un caractère UTF8. Revenons à notre caractère **€**. Le premier code qui remonte en exécutant **key** est $E2. En binaire : **11100010**. Ici on a trois bits à 1. Ceci signifie que le caractère **€** est codé sur 3 octets.

Effectuons un test avec le caractère UTF8 橙. Une exécution de **key** fait remonter le code 240, en binaire : **11110000**. On a 4 bits à 1. Le caractère 橙 est codé sur quatre octets.

Récupérer le code de caractères UTF8 entrés au clavier

L'idée est de détecter le nombre d'exécution de **key** à exécuter en fonction du caractère saisi au clavier :

- on exécute un premier **key**
- si le code est supérieur à 127, on fait glisser ce code de 1 bit vers la gauche, puis on teste le bit b7. Si ce bit est à 1 on re-exécute **key**.

Voici le code capable de saisir n'importe quel caractère UTF8 :

```
0 value keyUTF8

: toKeyUTF8  ( c -- )
    keyUTF8 8 lshift or to keyUTF8
  ;
```

Le mot **toKeyUTF8** reçoit un code clavier sur 8 bits et le concatène au contenu de la valeur **keyUTF8**. L'idée est de récupérer le codage UTF8 en une seule valeur numérique finale.

```
\ execute key recursively
: getKeys ( n -- )
    1 lshift dup $80 and    \ test if bit b7 is not null
    if   recurse            \ re-execute xkey
    else drop  then         \ otherwise, drop n
    key toKeyUTF8           \ and execute key 1 or may times
  ;
```

Le mot **getkeys** traite le code remonté par la première exécution de **key**. Il exécute un glissement d'un octet vers la gauche et teste le bit b7 (séquence **1 lshift dup $80 and**). Si ce bit est à 1, le mot se ré-exécute (séquence **if recurse**).

La récursivité permet de contrôler le nombre d'itérations de **getKeys** sans faire appel à des boucles et tests complexes. La récursivité s'interrompt dès qu'un bit b7 est à 0. La sortie de récursivité s'effectue après **then**. Le mot **getKeys** exécutera la séquence **key toKeyUTF8** autant de fois qu'il y a d'appels récursifs.

```
\ key version for UTF8 characters
: ukey
    key to keyUTF8
    keyUTF8 $7F > if                \ if 1st key code > $7F
        keyUTF8 1 lshift getKeys    \ execute xkey
    then
    keyUTF8
  ;
```

Le mot **ukey** peut maintenant se substituer au mot **key** pour récupérer le code UTF8 de n'importe quel caractère du jeu de caractères UTF8 :

```
hex
ukey  .    \ paste €   and <enter>, display : E282AC
```

Ce que confirme la documentation UTF8 en ligne.

Affichage de caractères UTF8 depuis leur code

Si on regarde la définition du mot **emit**, on trouve ceci :

```
: emit
    >R RP@ 1 type rdrop
  ;
```

La séquence de code **RP@ 1 type** limite strictement l'affichage d'un caractère code sur un seul octet. Cette séquence **hex E282AC emit** ne fonctionnera pas. De même :

```
: uemit
    >R RP@ 4 type rdrop
  ;
\  hex e282ac uemit   display : ��� ok
```

Le souci vient de l'ordre des octets d'une valeur numérique. Un dump mémoire de la pile donne ceci :

```
--addr---  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  ------chars-----
0802-00FA  00 00 00 00 00 00 AC 82 E2 00 00 00 00 00 08 01  ......���......
```

Il faut donc *retourner* les octets comme une chaussette :

```
\ reverse integer bytes, example:
\  hex 1a2b3C --> 3c2b1a
: reverse-bytes  ( n0 -- )
    0 { result }
    3 for
        result 100 * to result
        100 u/mod swap +to result
    next
    drop
    result
  ;
```

on peut maintenant réécrire notre mot **uemit** :

```
\ emit UTF8 encoded character
: uemit ( n -- )
    reverse-bytes
    >r rp@ 4 type
    rdrop
  ;
```

L'exécution de **hex E282AC uemit** affiche : **€**.

En conclusion, avec **ukey** et **uemit**, on dispose maintenant de mots permettant de traiter des caractères non-ASCII. Ainsi, avec un clavier grec :

```
hex
ukey     \ press key Σ display : CEA3
uemit    \ display Σ
```

Encodage depuis le point de code des caractères UTF8

Chaque caractère abstrait se voit associer un nombre unique. Ce nombre est appelé point de code. Le point de code est un nombre compris entre 1 et 17×2^{16}, soit potentiellement 1.114.112 signes. Un point de code est noté U+ suivi de la valeur hexadécimale du point de code.
Exemple : **U+00E9** pour le caractère `é`.

Le souci, vous l'avez déjà compris, est qu'on ne peut pas faire `hex e9 emit` avec eForth Linux.

Pour avoir la bonne séquence d'encodage UTF8 b1-b0 (pour byte1 byte0), il faut basculer les deux bits de poids fort de l'octet e9 vers b1. On découpe e9 comme ceci :

```
hex
e9 40 /mod
```

Ce qui nous laisse sur la pile de données `r` et `q` résultant de l'exécution de `/mod`, soit dans notre exemple les valeurs `29` et `3`. Pour transformer ceci en une valeur sur deux octets, on exécute ensuite :

```
100 * +
```

Ce qui nous laisse maintenant sur la pile de données la valeur hexadécimale `329`.

Revenons maintenant au format de codage UTF8 sur deux octets :

- `110b-bbbb 10bb-bbbb`

Ici, en jaune, on a une valeur de masquage, `1100000010000000` en binaire, `c080` en hexadécimal. C'est cette valeur de masque `c080` qu'on va appliquer au résultat de notre calcul précédent. Voici la séquence complète de codage depuis le point de code `e9` :

```
e9 40 /mod
100 * +
c080 or
```

Ce qui nous laisse le code final `c3a9`, qui est maintenant utilisable avec `uemit` :

```
c3a9 uemit   \ display char : é
```

On va maintenant automatiser cela….

Ré-encodage par récursivité

Dans la séquence `n 40 /mod`, on récupère à chaque itération un reste et un quotient. Quand une itération donne un quotient nul, on arrête. Ceci se prête merveilleusement à un traitement par récursivité :

```
$40 constant BYTE_DIVISOR

\ split n modulo BYTE_DIVISOR
```

```
: mod40Recombine ( n -- )
    BYTE_DIVISOR /mod
    dup 0 > if
        recurse
    then
    $100 * +
  ;
```

Le mot `mod40Recombine` découpe la valeur n en paires `r q`. Si `q` est égal à 0, on quitte la récursivité après `then` et on exécute `$100 * +` autant de fois que n a été découpé.

Il reste à appliquer un masque en fonction de la taille du point de code ré-encodé. Voici les valeurs limites pour les encodages de deux, trois, ou quatre octets :

```
$8000     constant LIMIT_2_BYTES
$10000    constant LIMIT_3_BYTES
$200000   constant LIMIT_4_bytes
```

Pour chacune de ces valeurs limites, voici les masques à appliquer :

```
$C080       constant MASK_2_BYTES
$E08080     constant MASK_3_BYTES
$F0808080   constant MASK_4_BYTES
```

Et pour finir, voici le mot `bytesToUTF8` qui applique le masque adapté à la taille du numéro de point de code :

```
: bytesToUTF8 ( n -- n' )
    >r
    r@ LIMIT_2_BYTES < if
        r> mod40Recombine
        MASK_2_BYTES OR
        exit
    then
    r@ LIMIT_3_BYTES < if
        r> mod40Recombine
        MASK_3_BYTES OR
        exit
    then
    r@ LIMIT_4_BYTES < if
        r> mod40Recombine
        MASK_4_BYTES OR
        exit
    then
    abort" UTF8 conversion failed"
  ;
```

Il y a certainement moyen de faire plus élégant. Cette définition a le mérite de fonctionner. En entrée, on empile le point de code à ré-encoder. En sortie, on obtient le code UTF8 utilisable par `uemit`.

Générer une table de caractères UTF8

L'idée est de prendre les premier et dernier numéros de point de code dans une table de caractères. Ces valeurs sont traitées dans une boucle permettant de générer une table des caractères.

Commençons par quelques mots utiles :

```
8 constant LINE_LIMIT

\ CR only if i MOD = 0
: cr? ( i -- )
    1+ LINE_LIMIT mod
    0= if
        cr
    then
  ;

\ display hex value format NNNN
: .###### ( n -- )
    <# # # # # # # #> type
  ;
```

Le mot `cr?` exécute un retour à la ligne si l'affichage atteint n colonnes. Le mot `.######` affiche une valeur n sur 6 chiffres. Voici enfin la boucle d'affichage :

```
: utf8Set { start stop -- }
    base @ { currentBase }
    hex
    stop 1+ start do
        i .######
        space
        i bytesToUTF8 uemit
        2 spaces
        i cr?
    loop
    currentBase base !
  ;
```

Voici la définition permettant d'afficher la table des caractères UTF8 du jeu de caractères grecs et coptes :

```
: greekAndCopt ( -- )
    $370 $3ff utf8Set
  ;
```

Son exécution affiche ceci :

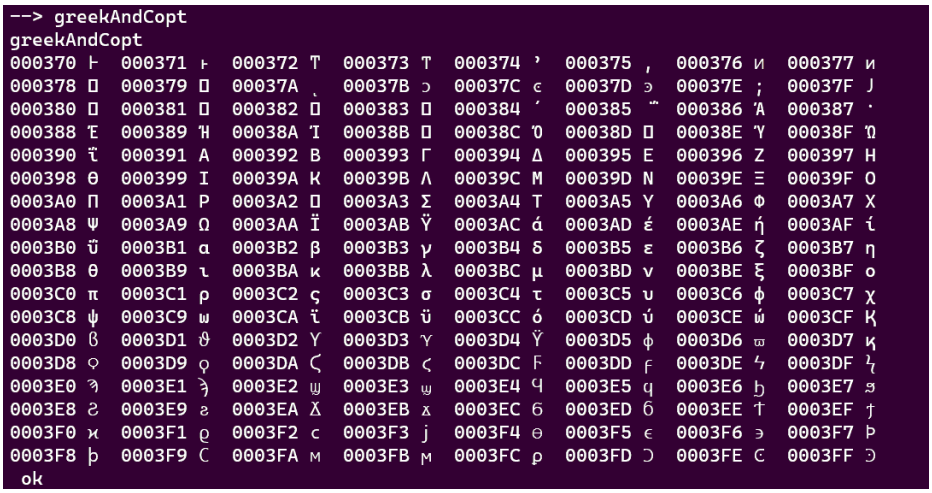


Figure 15: table des caractères grecs et coptes

Les chiffres indiquent le point de code de chaque caractère. Ici, on voit par exemple que le caractère φ a le point de code 3D5.

En conclusion, à quoi ça sert ?

Premièrement, ça sert à comprendre l'encodage UTF8.

Ensuite, on peut s'inspirer d'une partie de ce code pour compter le nombre de caractères. Exemple :

```
s" nb: φ"
\ display :
134495848 6
```

A vue d'œil, on a pourtant 5 caractères, alors que eForth indique une longueur de chaîne de 6 caractères !

Dans une procédure de tri alphanumérique, il peut s'avérer nécessaire de transformer certains caractères accentués en leur équivalent sans accent : à → a, é → e, etc.

Je vous laisse toute liberté pour trouver une application pratique.

Maitriser X11 avec eForth linux

X11 est le système graphique le plus populaire du système d´exploitation *UNIX*. Sa large distribution, le fait qu'ils soit libre de tous droits de distribution et surtout ses qualités techniques exceptionnelles en ont fait un standard de l'industrie du logiciel. Ses caractéristiques principales sont :

- l'utilisation d'une architecture *client/serveur* et cela dans une totale hétérogénéité (le serveur et le client peuvent fonctionner sur des architectures totalement différentes). Le dialogue entre le serveur et les clients se fait conformément au *protocole X* ce qui permet d'assurer la *transparence du dialogue*.
- la *portabilité* des librairies et des applications X11. Avec eForth Linux, on communique avec le serveur X sous Linux. On peut monter un serveur X sous Windows et utiliser le même code eForth sans changer une ligne de code.

En dépits des avantages de X11, le nombre de fonctions à connaître est important et la documentation titanesque et parfois difficile à aborder. Ce chapitre va aborder en priorité l'aspect graphique du serveur X11.

Un programme X peut se diviser en 3 parties :

- l'ouverture d'une connexion à un *serveur X* (serveur local ou terminal).
- les initialisations des fenêtres (cas de la Xlib) ou des objets (cas d'un toolkit)
- l'attente d'évènements (souris, clavier, ou autres...) dans une boucle infinie dont on ne sort qu'à la fin du programme.

Avec eForth Linux, le serveur X11 est accessible au travers du vocabulaire **x11** :

```
x11 vlist     \ display content of x11 vocabulary
```

Les concepts de base

Le display

Le *display* définit la connexion de l'application à un serveur X. Une fois initialisée, la valeur du *display* sera utilisée dans tous les appels aux fonctions X. **display** est calculé ainsi :

```
also x11
0 XOpenDisplay constant display
```

L´écran (screen)

Pour un **display** donné, on peut avoir plusieurs unités d'affichage. Initialisation d'un **screen** :

```
display XDefaultScreen constant screen
```

A partir des éléments `display` et `screen`, on récupère les paramètres de pixel :

```
display screen XBlackPixel constant black
display screen XWhitePixel constant white
```

Les fenêtres

La *fenêtre* est un des concepts les plus importants sous X. Voici l'initialisation d'une fenêtre racine :

```
display screen XRootWindow constant root-window
```

Le type X correspondant à la fenêtre est le type *Window*. Pour créer simplement une fenêtre, on utilisera `XcreateSimpleWindow` :

```
display root-window 0 0 640 480 0 black white
    XCreateSimpleWindow constant window
```

La fenêtre window nécessite les paramètres suivants :

- `display` qui est la connexion au serveur X
- `root-window` est la fenêtre parent
- `0 0` qui indique la position de départ de la nouvelle fenêtre dans la fenêtre parent
- `640 480` indiquent la taille de la fenêtre `window`
- `0 black white` sélectionnent respectivement l'épaisseur de la bordure, la couleur de bordure, la couleur de fond générale de la fenêtre `window`.

Pour afficher la fenêtre, on utilisera `XMapWindow` :

```
display window XMapWindow drop
```

Et pour terminer le paramétrage de fenêtre, il nous faut encore définir la constante `gc` (pour Graphic Content) :

```
display window 0 NULL XCreateGC constant gc
```

Voici une manière élégante de factoriser l'initialisation de l'environnement des fenêtres dans X11 :

```
also x11

0 value display
0 value screen
0 value black
0 value white
0 value root-window
0 value window
0 value gc

: new-window { width height -- }
```

```
    0 XOpenDisplay to display
    display XDefaultScreen to screen
    display screen XBlackPixel to black
    display screen XWhitePixel to white
    display screen XRootWindow to root-window
    display root-window 0 0
        width height 0 black white XCreateSimpleWindow to window
    display window XMapWindow drop
    display window 0 NULL XCreateGC to gc
;
```

Contenu détaillé des vocabulaires eForth Linux

Eforth Linux met à disposition de nombreux vocabulaires :

- **FORTH** est le principal vocabulaire ;
- certains vocabulaires servent à la mécanique interne pour Eforth Windows, comme **internals**, **asm…**

Vous trouverez ici la liste de tous les mots définis dans ces différents vocabulaires. Certains mots sont présentés avec un lien coloré :

align est un mot FORTH ordinaire ;

CONSTANT est mot de définition ;

begin marque une structure de contrôle ;

key est un mot d'exécution différée ;

LED est un mot défini par **constant**, **variable** ou **value** ;

registers marque un vocabulaire.

Les mots du vocabulaire **FORTH** sont affichés par ordre alphabétique. Pour les autres vocabulaires, les mots sont présentés dans leur ordre d'affichage.

Version v 7.0.7.15

FORTH

-	-rot	,	;	:	:noname	!
?	?do	?dup	.	."	.s	'
(local)	[	[']	[char]	[ELSE]	[IF]	[THEN]
]	{	}transfer	@	*	*/	*/MOD
/	/mod	#	#!	#>	#fs	#s
#tib	+	+!	+loop	+to	<	<#
<=	<>	=	>	>=	>BODY	>flags
>flags&	>in	>link	>link&	>name	>params	>R
>size	0<	0<>	0=	1-	1/F	1+
2!	2@	2*	2/	2drop	2dup	4*
4/	abort	abort"	abs	accept	afliteral	aft
again	ahead	align	aligned	allocate	allot	also
AND	ansi	argc	argv	ARSHIFT	asm	assert
at-xy	base	begin	bg	BIN	binary	bl
blank	block	block-fid	block-id	buffer	bye	c,
C!	C@	CASE	cat	catch	cd	CELL
cell/	cell+	cells	char	CLOSE-DIR	CLOSE-FILE	cmove
cmove>	CONSTANT	context	copy	cp	cr	CREATE
CREATE-FILE	current	decimal	default-key	default-type		default-use
defer	DEFINED?	definitions	DELETE-FILE	depth	DLSYM	do

DOES>	DROP	dump	dump-file	DUP	echo	editor
else	emit	empty-buffers		ENDCASE	ENDOF	erase
evaluate	EXECUTE	EXIT	extract	F-	f.	f.s
F*	F**	F/	F+	F<	F<=	F<>
F=	F>	F>=	F>S	F0<	F0=	FABS
FATAN2	fconstant	FCOS	fdepth	FDROP	FDUP	FEXP
fg	file-exists?		FILE-POSITION		FILE-SIZE	fill
FIND	fliteral	FLN	FLOOR	flush	FLUSH-FILE	FMAX
FMIN	FNEGATE	FNIP	for	forget	form	FORTH
forth-builtins		FOVER	FP!	FP@	fp0	free
FROT	FSIN	FSINCOS	FSQRT	FSWAP	fvariable	graphics
handler	here	hex	hld	hold	httpd	I
if	IMMEDIATE	include	included	included?	internals	invert
is	J	K	key	key?	L!	latestxt
leave	list	literal	load	loop	ls	LSHIFT
max	min	mkdir	mod	ms	ms-ticks	mv
n.	needs	negate	nest-depth	next	nip	nl
NON-BLOCK	normal	octal	OF	ok	only	open-blocks
OPEN-DIR	OPEN-FILE	OR	order	OVER	pad	page
PARSE	pause	PI	posix	postpone	precision	previous
prompt	pwd	quit	r"	R@	R/O	R/W
R>	r|	r~	rdrop	READ-DIR	READ-FILE	recurse
refill	remaining	remember	RENAME-FILE	repeat	REPOSITION-FILE	
required	reset	resize	RESIZE-FILE	restore	revive	rm
rmdir	rot	RP!	RP@	rp0	RSHIFT	s"
S>F	s>z	save	save-buffers		scr	sealed
see	set-precision		set-title	sf,	SF!	SF@
SFLOAT	SFLOAT+	SFLOATS	sign	SL@	sockets	SP!
SP@	sp0	space	spaces	start-task	startswith?	startup:
state	str	str=	streams	structures	SW@	SWAP
task	tasks	telnetd	terminate	termios	then	throw
thru	tib	to	touch	transfer	transfer{	type
u.	U/MOD	UL@	UNLOOP	until	update	use
used	UW@	value	VARIABLE	visual	vlist	vocabulary
W!	W/O	web-interface		while	words	WRITE-FILE
x11	XOR	z"	z>s			

ansi

```
terminal-restore terminal-save show hide scroll-up scroll-down clear-to-eol
bel esc
```

asm

```
end-code code, code4, code3, code2, code1, callot chere reserve code-at
code-start
```

editor

```
a r d e wipe p n l
```

graphics

poll wait flip window window heart vertical-flip viewport scale translate
}g g{ screen>g box color pressed? pixel height width event last-char last-key
mouse-y mouse-x RIGHT-BUTTON MIDDLE-BUTTON LEFT-BUTTON FINISHED TYPED RELEASED
PRESSED MOTION EXPOSED RESIZED IDLE internals

graphics/internals

update-event pending-key? update-key update-mouse image-resize EVENT-MASK
keybuffer-used keybuffer keybuffer-size xevent xevent-type image gc window-handle
root-window white black screen-depth xvisual colormap screen display raw-heart
heart-ratio heart-initialize cmax! cmin! heart-end heart-start heart-size
heart-steps heart-f raw-box g> >g gp gstack hline ty tx sy sx key-state!
key-state key-count backbuffer

httpd

notfound-response bad-response ok-response response send path method hasHeader
handleClient read-headers completed? body content-length header crnl= eat
skipover skipto in@<> end< goal# goal strcase= upper server client-cr client-emit
client-read client-type client-len client httpd-port clientfd sockfd body-read
body-1st-read body-chunk body-chunk-size chunk-filled chunk chunk-size
max-connections

internals

errno CALLCODE CALL0 CALL1 CALL2 CALL3 CALL4 CALL5 CALL6 CALL7 CALL8 CALL9
CALL10 CALL11 CALL12 CALL13 CALL14 CALL15 DOFLIT S>FLOAT? fill32 'heap
'context 'latestxt 'notfound 'heap-start 'heap-size 'stack-cells 'boot
'boot-size 'tib 'argc 'argv 'runner 'throw-handler NOP BRANCH 0BRANCH DONEXT
DOLIT DOSET DOCOL DOCON DOVAR DOCREATE DODOES ALITERAL LONG-SIZE S>NUMBER?
'SYS YIELD EVALUATE1 'builtins internals-builtins autoexec boot-set-title
e' @line grow-blocks use?! common-default-use block-data block-dirty clobber
clobber-line include+ path-join included-files raw-included include-file
sourcedirname sourcefilename! sourcefilename sourcefilename# sourcefilename&
starts../ starts./ dirname ends/ default-remember-filename *remember-filename*
restore-name save-name forth-wordlist setup-saving-base 'cold park-forth
park-heap saving-base crtype cremit cases (+to) (to) --? }? ?room scope-create
do-local scope-clear scope-exit local-op scope-depth local+! local! local@
<>locals locals-here locals-area locals-gap locals-capacity ?ins. ins.
vins. onlines line-pos line-width size-all size-vocabulary vocs. voc. voclist
voclist-from see-all >vocnext see-vocabulary nonvoc? see-xt ?see-flags
see-loop see-one indent+! icr see. indent mem= ARGS_MARK -TAB +TAB NONAMED
BUILTIN_FORK SMUDGE IMMEDIATE_MARK dump-line ca@ cell-shift cell-base cell-mask
#f+s internalized BUILTIN_MARK zplace $place free. boot-prompt raw-ok [SKIP]'
[SKIP] ?stack sp-limit input-limit tib-setup raw.s $@ digit parse-quote
leaving, leaving)leaving leaving(value-bind evaluate&fill evaluate-buffer
arrow ?arrow. ?echo input-buffer immediate? eat-till-cr wascr *emit *key
notfound last-vocabulary voc-stack-end xt-transfer xt-hide xt-find& scope

posix

```
FNDELAY F_SETFL fcntl CLOCK_BOOTTIME_ALARM CLOCK_REALTIME_ALARM CLOCK_BOOTTIME
CLOCK_MONOTONIC_COARSE CLOCK_REALTIME_COARSE CLOCK_MONOTONIC_RAW
CLOCK_THREAD_CPUTIME_ID
CLOCK_PROCESS_CPUTIME_ID CLOCK_MONOTONIC CLOCK_REALTIME timespec clock_gettime
0777 SIGPIPE SIGBUS SIGKILL SIGINT SIGHUP SIG_IGN SIG_DFL EPIPE EAGAIN
d0=ior d0<ior 0=ior 0<ior stdin-key stdout-write O_NONBLOCK O_APPEND O_TRUNC
O_CREAT O_RDWR O_WRONLY O_RDONLY MAP_ANONYMOUS MAP_FIXED MAP_PRIVATE PROT_EXEC
PROT_WRITE PROT_READ PROT_NONE SEEK_END SEEK_CUR SEEK_SET stderr stdout
stdin errno .d_name .d_type readdir closedir opendir getwd rmdir mkdir
chdir signal usleep realloc sysfree malloc rename unlink mprotect munmap
mmap waitpid wait fork sysexit fsync ftruncate lseek write read close creat
open sign-extend shared-library sysfunc sofunc calls dlopen 'dlopen RTLD_NOW
RTLD_LAZY
```

sockets

```
sockaccept ip. ip# ->h_addr ->addr! ->addr@ ->port! ->port@ sockaddr l,
s, bs, SO_REUSEADDR SOL_SOCKET sizeof(sockaddr_in) AF_INET SOCK_RAW SOCK_DGRAM
SOCK_STREAM gethostbyname recvmsg recvfrom recv sendmsg sendto send setsockopt
poll sockaccept connect listen bind socket
```

tasks

```
main-task .tasks task-list
```

telnetd

```
server broker-connection wait-for-connection connection telnet-key
telnet-type telnet-emit broker client-len client telnet-port clientfd
sockfd
```

termios

```
termios-key termios-key? pending form winsize sizeof(winsize) TIOCGWINSZ
normal-mode raw-mode termios! termios@ VMIN VTIME TCSAFLUSH _ECHO ICANON
.c_cc[] .c_lflag new-termios old-termios sizeof(termios) delay-mode nodelay-mode
ioctl tcsetattr tcgetattr
```

web-interface

```
server webserver-task do-serve handle1 serve-key serve-type handle-input
handle-index out-string output-stream input-stream out-size webserver index-html
index-html#
```

x11

```
GenericEvent MappingNotify ClientMessage ColormapNotify SelectionNotify
SelectionRequest SelectionClear PropertyNotify CirculateRequest CirculateNotify
```

ResizeRequest GravityNotify ConfigureRequest ConfigureNotify ReparentNotify
MapRequest MapNotify UnmapNotify DestroyNotify CreateNotify VisibilityNotify
NoExpose GraphicsExpose Expose KeymapNotify FocusOut FocusIn LeaveNotify
EnterNotify MotionNotify ButtonRelease ButtonPress KeyRelease KeyPress
xevent# OwnerGrabButtonMask ColormapChangeMask PropertyChangeMask FocusChangeMask
SubstructureRedirectMask SubstructureNotifyMask ResizeRedirectMask
StructureNotifyMask
VisibilityChangeMask ExposureMask KeymapStateMask ButtonMotionMask
Button5MotionMask
Button4MotionMask Button3MotionMask Button2MotionMask Button1MotionMask
PointerMotionHintMask PointerMotionMask LeaveWindowMask EnterWindowMask
ButtonReleaseMask ButtonPressMask KeyReleaseMask KeyPressMask xmask NoEventMask
xexposure xconfigure xmotion xkey xbutton xany bool time win xevent-size
NULL ZPixmap XYPixmap XYBitmap XFillRectangle XSetBackground XSetForeground
XDrawString XSelectInput XPutImage XNextEvent XMapWindow XLookupString
XFlush XDestroyImage XDefaultVisual XDefaultDepth XCreateSimpleWindow XCreateImage
XCreateGC XCheckMaskEvent XRootWindow XDefaultScreen XDefaultColormap
XScreenOfDisplay
XDisplayOfScreen XWhitePixel XBlackPixel XOpenDisplay xlib

Index lexical