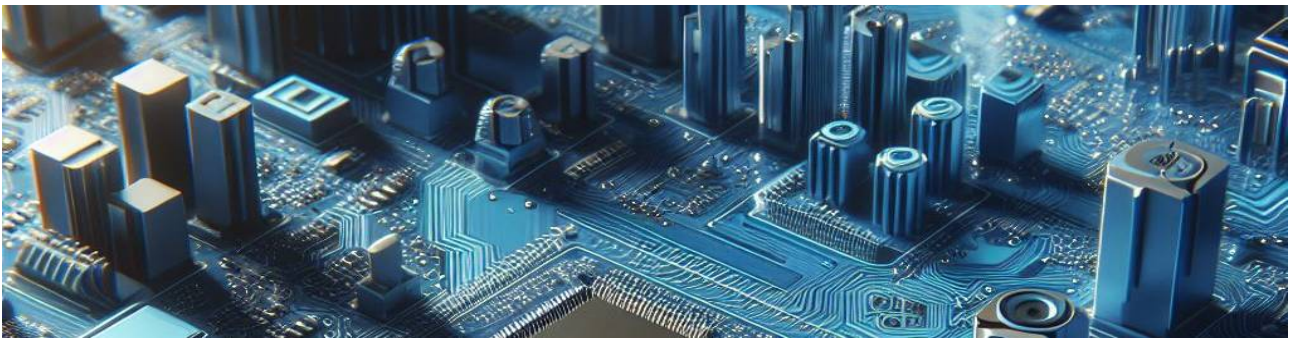


Le grand livre de eFORTH Windows

version 1.3 - 13 novembre 2024



Auteur

- Marc PETREMANN

Table des matières

Pourquoi programmer en langage FORTH sur eForth Windows?	5
Préambule.....	5
Limites entre langage et application.....	5
C'est quoi un mot FORTH?.....	6
Un mot c'est une fonction?.....	6
Le langage FORTH comparé au langage C.....	7
Ce que FORTH permet de faire par rapport au langage C.....	8
Mais pourquoi une pile plutôt que des variables?.....	9
Êtes-vous convaincus?.....	9
Existe-t-il des applications professionnelles écrites en FORTH?.....	9
Un vrai Forth 64 bits avec eForth Windows.....	11
Les valeurs sur la pile de données.....	11
Les valeurs en mémoire.....	11
Traitement par mots selon taille ou type des données.....	12
Conclusion.....	13
Installation sous Windows.....	15
Paramétrer eForth Windows.....	15
Commentaires et mise au point.....	18
Ecrire un code FORTH lisible.....	18
Indentation du code source.....	19
Les commentaires.....	20
Les commentaires de pile.....	20
Signification des paramètres de pile en commentaires.....	21
Commentaires des mots de définition de mots.....	22
Les commentaires textuels.....	22
Commentaire en début de code source.....	23
Outils de diagnostic et mise au point.....	23
Le décompilateur.....	23
Dump mémoire.....	24
Moniteur de pile.....	24
Dictionnaire / Pile / Variables / Constantes.....	26
Étendre le dictionnaire.....	26
Gestion du dictionnaire.....	26
Piles et notation polonaise inversée.....	27
Manipulation de la pile de paramètres.....	28
La pile de retour et ses utilisations.....	28
Utilisation de la mémoire.....	29
Variables.....	29
Constantes.....	30
Valeurs pseudo-constantes.....	30
Outils de base pour l'allocation de mémoire.....	30

Les variables locales avec eForth Windows.....	32
Introduction.....	32
Le faux commentaire de pile.....	32
Action sur les variables locales.....	33
Structures de données pour eForth Windows.....	36
Préambule.....	36
Les tableaux en FORTH.....	36
Tableau de données à une dimension.....	36
Mots de définition de tableaux.....	37
Lire et écrire dans un tableau.....	37
Exemple pratique de gestion d'écran.....	38
Gestion de structures complexes.....	40
Règles de nommage des structures et accesseurs.....	41
Choix de la taille des champs dans une structure.....	42
Définition de sprites.....	44
Les nombres réels avec eForth Windows.....	47
Les réels avec eForth Windows.....	47
Precision des nombres réels avec eForth Windows.....	47
Constantes et variables réelles.....	48
Opérateurs arithmétiques sur les réels.....	48
Opérateurs mathématiques sur les réels.....	48
Opérateurs logiques sur les réels.....	49
Transformations entiers ↔ réels.....	49
Affichage des nombres et chaînes de caractères.....	51
Changement de base numérique.....	51
Définition de nouveaux formats d'affichage.....	52
Affichage des caractères et chaînes de caractères.....	54
Variables chaînes de caractères.....	56
Code des mots de gestion de variables texte.....	56
Ajout de caractère à une variable alphanumérique.....	58
Les vocabulaires avec eForth Windows.....	60
Liste des vocabulaires.....	60
Les vocabulaires essentiels.....	60
Liste du contenu d'un vocabulaire.....	61
Utilisation des mots d'un vocabulaire.....	61
Chainage des vocabulaires.....	61
Les mots à action différée.....	63
Définition et utilisation de mots avec defer.....	64
Définition d'une référence avant.....	64
Un cas pratique.....	65
Les mots de création de mots.....	67
Utilisation de does>.....	67
Exemple de gestion de couleur.....	68
Exemple, écrire en pinyin.....	69
Etendre le vocabulaire graphics pour Windows.....	71

Définition des fonctions dans graphics internals.....	72
Définition des mots dans graphics.....	73
Contenu détaillé des vocabulaires eForth Windows.....	75
Version v 7.0.7.15.....	75
FORTH.....	75
windows.....	76
Ressources.....	79
En anglais.....	79
En français.....	79
GitHub.....	79
Facebook.....	79

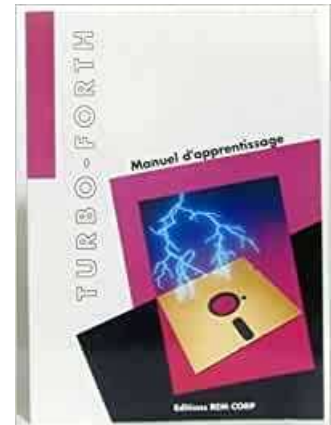
Pourquoi programmer en langage FORTH sur eForth Windows?

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 Loisetech - 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, il fallait faire tenir MS-DOS 3.0 et l'application !

Cela 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.

La compilation des programmes écrits en langage FORTH s'effectue dans eForth Windows. 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){  
    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 Windows:

```
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 WINDOWS, 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 de WINDOWS. 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 WINDOWS, 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 Windows

eForth Windows 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 Windows, 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**.

Avec eForth Windows, un caractère ASCII sera désigné par un entier 64 bits, mais dont la valeur sera bornée [32..127[. Exemple :

```
67 emit \ display C
```

Les valeurs en mémoire

eForth Windows 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....

Installation sous Windows

Vous trouverez les dernières versions de eFORTH pour WINDOWS ici:

<https://eforth.appspot.com/windows.html>

La version de programme à télécharger se trouve en STABLE RELEASE ou Beta Release.

Depuis µEforth version 7.0.7.21 seule la version 64 reste disponible.

Le programme téléchargé est directement exécutable. Une fois le programme téléchargé, commencez par le copier dans un dossier de travail. Ici, j'ai choisi de mettre le programme téléchargé dans un dossier nommé **eforth**.

Pour exécuter µEforth Windows, cliquez sur le programme téléchargé et copié dans ce dossier eforth. Si Windows émet un message d'alerte:

- cliquez sur Informations complémentaires
- puis cliquez sur *Exécuter quand même*

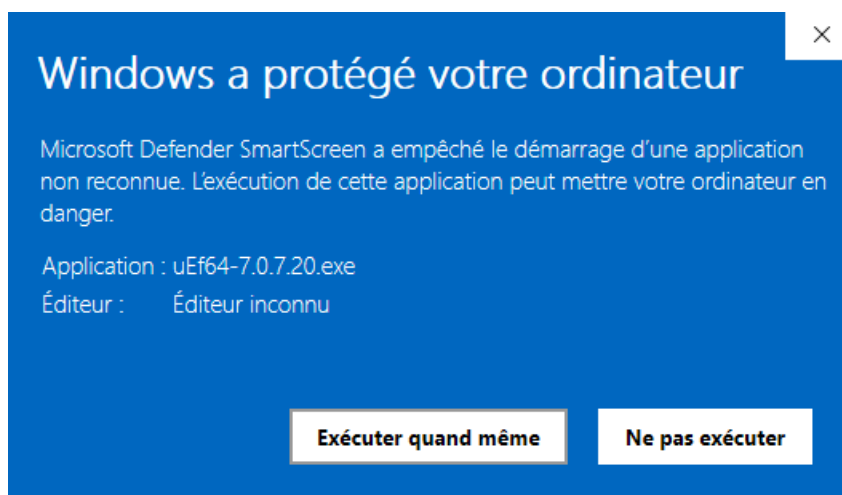


Figure 1: passer le message d'alerte Windows

Une fois ce choix validé, vous pourrez exécuter eForth comme n'importe quel autre programme Windows.

Paramétrer eForth Windows

eFORTH n'a pas besoin d'être installé pour fonctionner. On exécute simplement le fichier téléchargé, ici **uEf64-7.0.7.21.exe**. Voici la fenêtre µEforth :

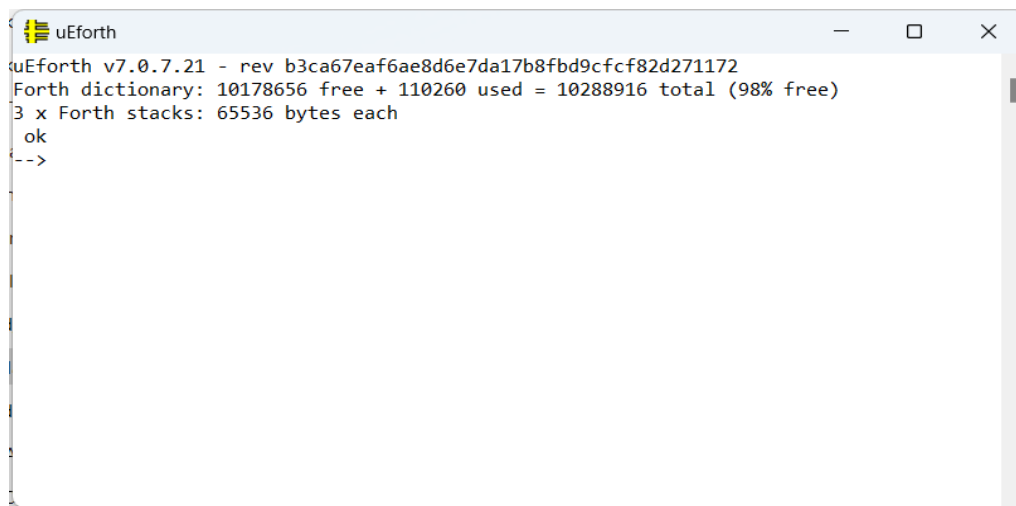


Figure 2: La fenêtre *μEforth* sous Windows

Pour tester le bon fonctionnement de eForth, tapez **words**.

Pour quitter eForth, tapez **bye**.

Quand eForth est ouvert, vous pouvez créer un raccourci à épingler à la barre des tâches, ce qui facilitera un nouveau lancement de eForth.

Pour modifier les couleurs de fond et de texte de eForth, posez le pointeur de la souris sur le logo *μEforth*, puis clic droit et sélectionnez *Propriétés* :

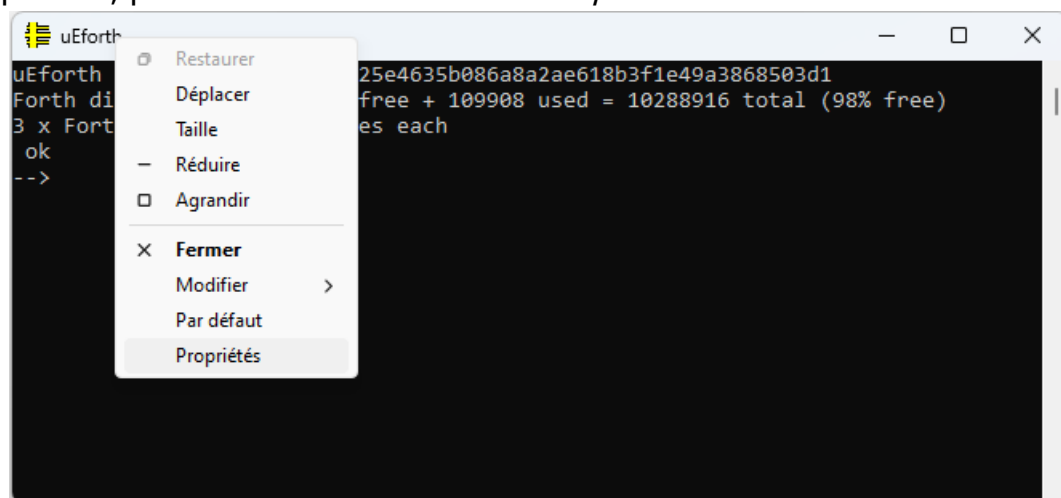


Figure 3: Sélection des propriétés d'affichage

Dans les Propriétés, cliquez sur l'onglet *Couleurs* :

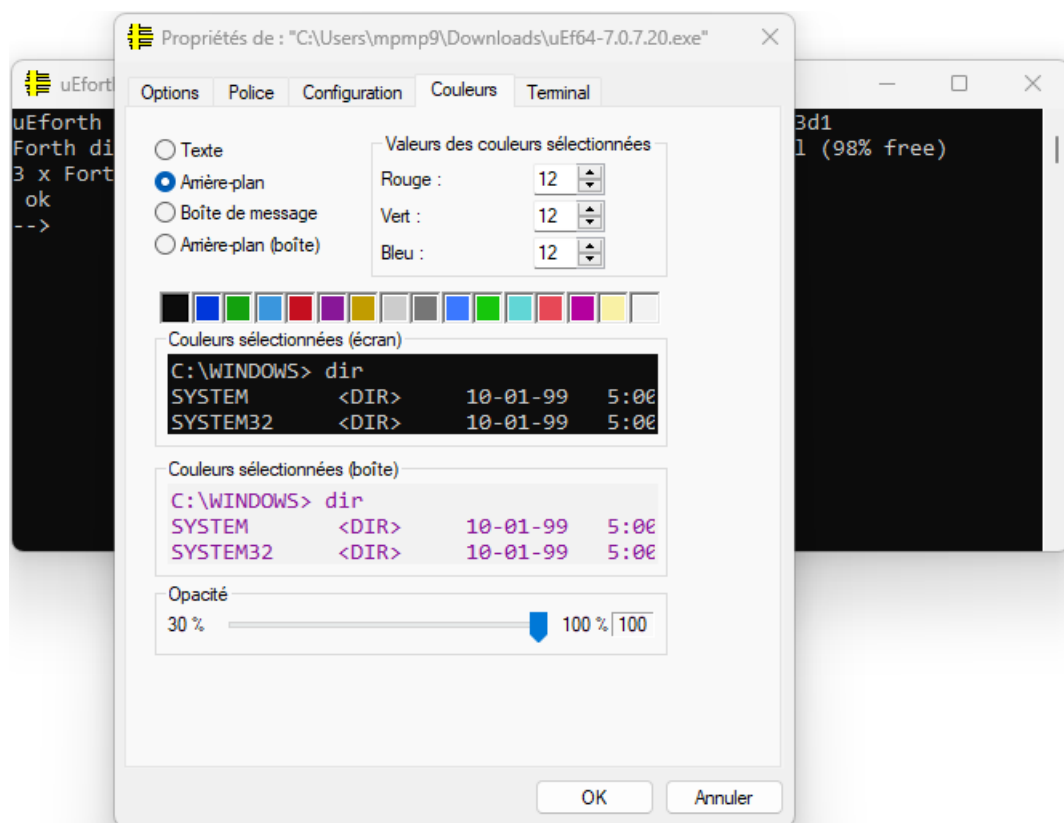


Figure 4: Choix des couleurs d'affichage

Pour ma part, j'ai choisi d'afficher le texte en noir sur fond blanc. Cliquez sur OK pour valider ce choix. Au prochain lancement de eForth, vous retrouverez les couleurs sélectionnées comme paramètres par défaut pour l'affichage dans la fenêtre eForth.

Commentaires et mise au point

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

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

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

```
: inGrid? { n gridPos -- f1 } 0 { f1 } gridPos getGridAddr for aft  
getNumber n = if -1 to f1 then then next drop f1 ;
```

Ce code sera parfaitement compilé par eForth Windows. 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 Windows 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 puissiez 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!**
- Mot d'affichage de donnée **.date**

1 Integrated Development Environment = Environnement de Développement Intégré

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 Windows 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      \ 海报今天的日期
...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 -- f1 }
  0 { f1 }
  gridPos getGridAddr
  for
    aft
      getNumber n =
      if
        -1 to f1
      then
    then
  next
  drop
  f1
;
```

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

```
: inGrid? { n gridPos -- f1 }
  0 { f1 }   gridPos getGridAddr
  for aft
    getNumber n =
    if -1 to f1 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 Windows, 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 Windows, 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'intervalle [0..255].

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

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

```
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 Windows qui est déjà au format 32 ou 64 bits ;
- **fl** indique une valeur booléenne, 0 ou non zéro ;
- **n** indique un entier. Entier signé 32 ou 64 bits pour eForth Windows;
- **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
    sendDataToSSD1306
;
```

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

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

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

Les commentaires textuels

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

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

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

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

;

Commentaire en début de code source

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

```
\ *****
\ key word for UT8 characters
\   Filename:      uekey.fs
\   Date:          29 nov 2023
\   Updated:       29 nov 2023
\   File Version:  1.1
\   MCU:           Linux / Web / Windows
\   Forth:         eForth Windows
\   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 Windows, le développeur peut décompiler ses définitions. Pour décompiler un mot, il suffit de taper **see** suivi du mot à décompiler :

```
: C>F ( °C --- °F) \ Conversion Celsius in Fahrenheit
```

```

    9 5 */ 32 +
;
see c>f
\ display:
: C>F
    9 5 */ 32 +
;

```

Beaucoup de mots du dictionnaire FORTH de eForth Windows 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.

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 Windows 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 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 Windows 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
```

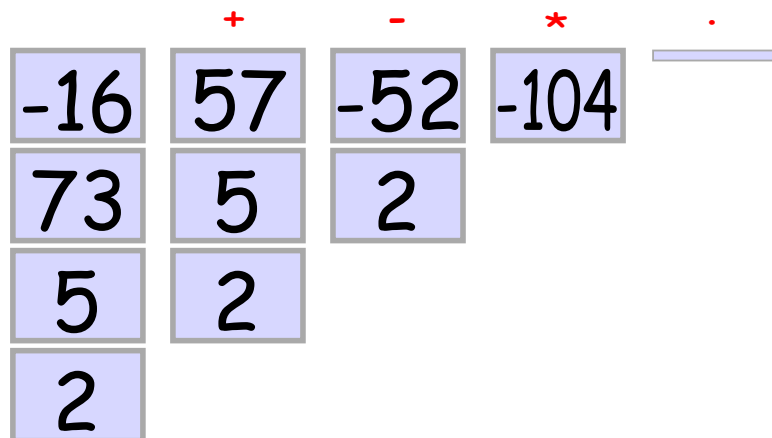
laisse la pile de paramètres dans l'état

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
```

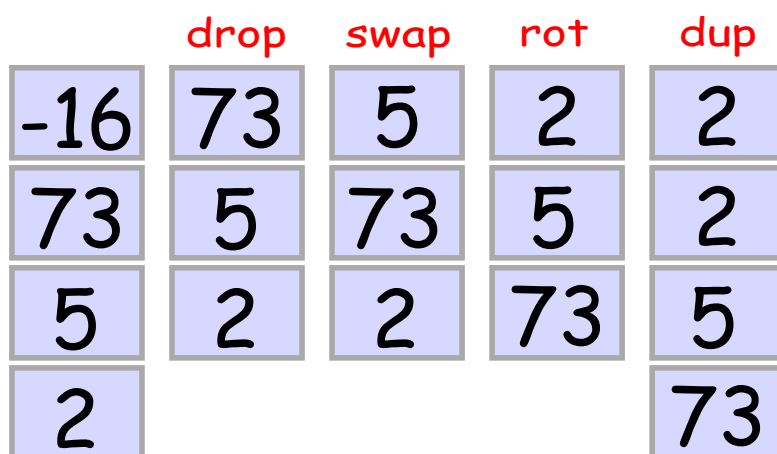
Notez que eForth Windows affiche commodément les éléments de la pile lors de l'interprétation de chaque ligne et que la valeur de -16 est affichée sous la forme d'entier non signé 32 ou 64 bits. En outre, le mot `.` consomme la valeur de données -104, laissant la pile vide. Si nous exécutons `.` sur 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 Windows 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 Windows établit des liens entre le mot appelant et les mots définis précédemment qui doivent être invoqués par l'exécution du nouveau mot. Ce mécanisme de liaison, lors de l'exécution, utilise la pile de retour (rstack). L'adresse du mot suivant à invoquer est placée sur la pile de retour de sorte que, lorsque le mot courant est terminé en cours d'exécution, le système sait où passer au mot suivant. Comme les mots peuvent être imbriqués, il doit y avoir une pile de ces adresses de retour.

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

stockage temporaire, vous devez la remettre dans son état d'origine, sinon vous ferez probablement planter le système eForth Windows. 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 Windows, les nombres 32 ou 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 une constante est la valeur des données restant sur la pile.

```
\ defines extrem values for alpha channel
255 constant SDL_ALPHA_OPAQUE
0   constant SDL_ALPHA_TRANSPARENT
```

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

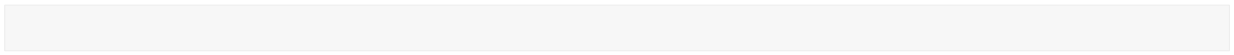
Les mots **create** et **allot** sont les outils de base pour réserver un espace mémoire et y attacher une étiquette. Par exemple, la transcription suivante montre une nouvelle entrée de dictionnaire **graphic-array** :

```
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 Windows

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 adresse pour dump
\ END_ADDR        \ dernière adresse pour dump
\ OSTART_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 * { OSTART_ADDR } \ calc. addr for loop start
  16 / 1+ { LINES }
  base @ { myBASE }           \ save current base
  hex
  \ outer loop
  LINES 0 do
    OSTART_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
      OSTART_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
      OSTART_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 Windows

Préambule

eForth Windows 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 :

```
cell . \ affiche 8
```

La valeur 8 signifie que la taille des éléments déposés sur la pile de données est de 8 octets, soit $8 \times 8 \text{ bits} = 64 \text{ bits}$.

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 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.

Lire et écrire dans un tableau

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

```

arrayC myCTemps
    6 allot \ allocate 6 bytes
    0 myCTemps 6 0 fill \ fill this 6 bytes with value 0
32 0 myCTemps c! \ store 32 in myCTemps[0]
25 5 myCTemps c! \ store 25 in myCTemps[5]

```

```
0 myCTemps c@ .          \ display 32
```

Dans notre exemple, le tableau contient 6 éléments. Avec eFORTH, il y a assez d'espace mémoire pour traiter des tableaux bien plus grands, avec 1.000 ou 10.000 éléments par exemple. Il est facile de créer des tableaux à plusieurs dimensions. Exemple de tableau à deux dimensions :

```
63 constant SCR_WIDTH
16 constant SCR_HEIGHT
create mySCREEN
  SCR_WIDTH SCR_HEIGHT * allot          \ allocate 63 * 16 bytes
  mySCREEN SCR_WIDTH SCR_HEIGHT * bl fill \ fill this memory with 'space'
```

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

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

```
: xySCRaddr { x y -- addr }
  SCR_WIDTH y *
  x + mySCREEN +
;
: SCR@ ( x y -- c )
  xySCRaddr c@
;
: SCR! ( c x y -- )
  xySCRaddr c!
;
char X 15 5 SCR!    \ store char X at col 15 line 5
15 5 SCR@ emit      \ display X
```

Exemple pratique de gestion d'écran

Voici comment afficher la table des caractères disponibles :

```
: tableChars ( -- )
  base @ >r hex
  128 32 do
    16 0 do
      j i + dup . space emit space space
    loop
    cr
  16 +loop
  256 160 do
    16 0 do
      j i + dup . space emit space space
    loop
    cr
  16 +loop
```

```

cr
r> base !
;
tableChars

```

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

```

--> tableChars
20  21  !  22  "  23  #  24  $  25  %  26  &  27  '  28  (  29  )  2A  *  2B  +  2C  ,  2D  -  2E  .  2F  /
30  0  31  1  32  2  33  3  34  4  35  5  36  6  37  7  38  8  39  9  3A  :  3B  ;  3C  <  3D  =  3E  >  3F  ?
40  @  41  A  42  B  43  C  44  D  45  E  46  F  47  G  48  H  49  I  4A  J  4B  K  4C  L  4D  M  4E  N  4F  O
50  P  51  Q  52  R  53  S  54  T  55  U  56  V  57  W  58  X  59  Y  5A  Z  5B  [  5C  \  5D  ]  5E  ^  5F  _
60  `  61  a  62  b  63  c  64  d  65  e  66  f  67  g  68  h  69  i  6A  j  6B  k  6C  l  6D  m  6E  n  6F  o
70  p  71  q  72  r  73  s  74  t  75  u  76  v  77  w  78  x  79  y  7A  z  7B  {  7C  |  7D  }  7E  ~  7F  ¨
A0  á  A1  â  A2  ã  A3  ü  A4  ñ  A5  ñ  A6  ¢  A7  ¢  A8  ¢  A9  ¢  AA  ¢  AB  ¢  AC  ¢  AD  ¢  AE  ¢  AF  ¢
B0  ¢  B1  ¢  B2  ¢  B3  ¢  B4  ¢  B5  ¢  B6  ¢  B7  ¢  B8  ¢  B9  ¢  BA  ¢  BB  ¢  BC  ¢  BD  ¢  BE  ¢  BF  ¢
C0  ¢  C1  ¢  C2  ¢  C3  ¢  C4  ¢  C5  ¢  C6  ¢  C7  ¢  C8  ¢  C9  ¢  CA  ¢  CB  ¢  CC  ¢  CD  ¢  CE  ¢  CF  ¢
D0  ¢  D1  ¢  D2  ¢  D3  ¢  D4  ¢  D5  ¢  D6  ¢  D7  ¢  D8  ¢  D9  ¢  DA  ¢  DB  ¢  DC  ¢  DD  ¢  DE  ¢  DF  ¢
E0  ¢  E1  ¢  E2  ¢  E3  ¢  E4  ¢  E5  ¢  E6  ¢  E7  ¢  E8  ¢  E9  ¢  EA  ¢  EB  ¢  EC  ¢  ED  ¢  EE  ¢  EF  ¢
F0  -  F1  ±  F2  =  F3  %  F4  ¢  F5  ¢  F6  ¢  F7  ¢  F8  ¢  F9  ¢  FA  ¢  FB  ¢  FC  ¢  FD  ¢  FE  ¢  FF  ¢

```

Figure 5: exécution de `tableChars`

Ces caractères sont ceux du jeu ASCII MS-DOS. Certains de ces caractères sont semi-graphiques. Voici une insertion très simple d'un de ces caractères dans notre écran virtuel :

```

$db dup 5 2 SCR!      6 2 SCR!
$b2 dup 7 3 SCR!      8 3 SCR!
$b1 dup 9 4 SCR!     10 4 SCR!

```

Voyons maintenant comment afficher le contenu de notre écran virtuel. Si on considère chaque ligne de l'écran virtuel comme chaîne alphanumérique, il suffit de définir ce mot pour afficher une des lignes de notre écran virtuel:

```

: dispLine { numLine -- }
  SCR_WIDTH numLine *
  mySCREEN + SCR_WIDTH type
;

```

Au passage, on va créer une définition permettant d'afficher n fois un même caractère :

```

: nEmit ( c n -- )
  for
    aft dup emit then
  next
  drop
;

```

Et maintenant, on définit le mot permettant d'afficher le contenu de notre écran virtuel. Pour bien voir le contenu de cet écran virtuel, on l'encadre avec des caractères spéciaux :

```

: dispScreen
  0 0 at-xy
  \ display upper border
  $da emit    $c4 SCR_WIDTH nEmit    $bf emit    cr
  \ display content virtual screen

```

```

SCR_HEIGHT 0 do
  $b3 emit    i dispLine    $b3 emit    cr
loop
\ display bottom border
$c0 emit    $c4 SCR_WIDTH nEmit    $d9 emit    cr
;

```

L'exécution de notre mot **dispScreen** affiche ceci :

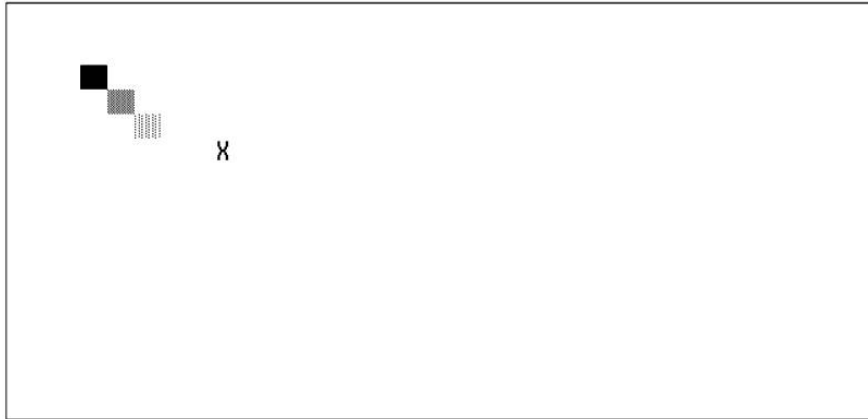


Figure 6: exécution de *dispScreen*

Dans notre exemple d'écran virtuel, nous montrons que la gestion d'un tableau à deux dimensions a une application concrète. Notre écran virtuel est accessible en écriture et en lecture. Ici, nous affichons notre écran virtuel dans le fenêtre du terminal.

Gestion de structures complexes

eForth 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 ->YMDHMS-year
  ptr field ->YMDHMS-month
  ptr field ->YMDHMS-day
  ptr field ->YMDHMS-hour
  ptr field ->YMDHMS-min
  ptr field ->YMDHMS-sec

```

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

Le mot **YMDHMS** a comme seule utilité d'initialiser les accesseurs. Voici comment sont utilisés ces accesseurs :

```

create DateTime
  YMDHMS allot

```



```

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

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

DateTime .date

```

Règles de nommage des structures et accesseurs

Une structure est définie par le mot **struct**. Le nom choisi dépend du contexte d'utilisation. Il ne doit pas être trop long, pour rester lisible. Ici, une structure définissant une couleur dans la librairie SDL :

```
struct SDL_Color ( -- n )
```

Cette structure a comme nom **SDL_Color**. Ce nom a été choisi car cette structure porte le même nom dans la librairie en langage C.

Voici la définition, en Forth, des accesseurs correspondant à cette structure **SDL_Color**:

```

struct SDL_Color ( -- n )
  i8 field ->Color-r
  i8 field ->Color-g
  i8 field ->Color-b
  i8 field ->Color-a

```

Chaque définition commence par un mot indiquant la taille du champ de donnée dans la structure, ici **i8**.

Ce mot **i8** est suivi du mot **field** qui va nommer l'accesseur, **->Color-r** par exemple.

On peut choisir un nom d'accesseur à sa convenance, exemple :

```
i8 field red-color
```

ou

```
i8 field colorRed
```

Mais ces noms finissent rapidement par rendre un programme difficile à déchiffrer. Pour cette raison, il est souhaitable de précéder un nom d'accesseurs par **->**. On fait suivre par le nom de la structure, ici **Color**, suivi d'un tiret et un nom discriminant, ici **r**:

```
i8 field ->Color-r
Autre exemple:
struct SDL_GenericEvent
    i32 field ->GenericEvent-type
    i32 field ->GenericEvent-timestamp
```

Ici, on définit la structure **SDL_GenericEvent** et deux accesseurs 32 bits: -
>GenericEvent-type et **>GenericEvent-timestamp**.

Par la suite, si on retrouve l'accesseur **>GenericEvent-type** dans un programme, on saura immédiatement que c'est un accesseur associé à la structure **GenericEvent**.

Choix de la taille des champs dans une structure

La taille d'un champ dans une structure est définie par un de ces mot :

- **ptr** pour pointer une donnée sur la taille de une cellule, 8 octets pour eForth Windows
- **i64** pour pointer une donnée sur 8 octet (64 bits)
- **i32** pour pointer une donnée sur 4 octet (32 bits)
- **i16** pour pointer une donnée sur 2 octet (16 bits)
- **i8** pour pointer une donnée sur 1 octet (8 bits)

Pour eForth Windows, les mots **ptr** et **i64** ont la même action.

On ne peut pas gérer des champs de taille variable, pour une chaîne de caractères par exemple.

Si on doit définir un champ d'une taille spéciale, on définira ce type ainsi :

```
structures definition
    10 10 typer phoneNum
    5  5 typer zipCode
```

On peut aussi s'en passer en utilisant la taille de données directement. Exemple :

```
structures
struct City
    5  field ->City-zip
    64 field ->City-name
```

Si on exécute **City**, ce mot empilera la taille totale de la structure, ici la valeur 69. On utilisera cette valeur pour réserver le nombre de caractères requis pour des données:

```
create BORDEAUX
  City allot
```

Nous n'allons pas entrer dans le détail de la gestion des champs de notre structure **City**.

Pour ce qui concerne les champs définis par **i8** à **i64**, on ne peut pas utiliser les seuls mots **@** et **!** pour lire et écrire des valeurs numériques dans ces champs.

Voici les mots permettant d'accéder aux données en fonction de leur taille :

	i8	i16	i32	i64
fetch	C@	UW@	UL@	@
store	C!	W!	L!	!

Figure 7: --

On avait 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 de l'accesseur correspondant. On peut redéfinir l'espace alloué de notre structure YMDHMS en utilisant **i8** et **i16** :

```
structures
struct cYMDHMS
  i16 field ->cYMDHMS-year
  i8  field ->cYMDHMS-month
  i8  field ->cYMDHMS-day
  i8  field ->cYMDHMS-hour
  i8  field ->cYMDHMS-min
  i8  field ->cYMDHMS-sec

create cDateTime
  cYMDHMS allot

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

Il est conseillé de factoriser l'emploi des accesseurs dans une définition globale :

```
: date! { year month day hour min sec addr -- }
  year  addr ->cYMDHMS-year  w!
  month addr ->cYMDHMS-month c!
  day   addr ->cYMDHMS-day   c!
  hour  addr ->cYMDHMS-hour  c!
  min   addr ->cYMDHMS-min   c!
  sec   addr ->cYMDHMS-sec   c!
;
2024 11 09 18 25 40 cDateTime date!
```

Avec **date!**, on ne s'occupe plus de savoir quels champs sont sur un ou deux octets dans la structure **cYMDHMS**.

Si on doit changer la taille d'un champ, seule la définition de **date!** devra être modifiée.

Voici comment lire les données dans **cDateTime** :

```
: .date { date -- }
  ." YEAR: " date ->cYMDHMS-year   uw@ . cr
  ." MONTH: " date ->cYMDHMS-month c@ . cr
  ." DAY: " date ->cYMDHMS-day     c@ . cr
  ." HH: " date ->cYMDHMS-hour    c@ . cr
  ." MM: " date ->cYMDHMS-min     c@ . cr
  ." SS: " date ->cYMDHMS-sec     c@ . cr
;

cDateTime .date \ display:
\ YEAR: 2024
\ MONTH: 11
\ DAY: 9
\ HH: 18
\ MM: 25
\ SS: 40
```

Définition de sprites

On avait précédemment défini un écran virtuel comme tableau à deux dimensions. Les dimensions de ce tableau sont définies par deux constantes. Rappel de la définition de cet écran virtuel:

```
63 constant SCR_WIDTH
16 constant SCR_HEIGHT
create mySCREEN
  SCR_WIDTH SCR_HEIGHT * allot \ allocate 63 * 16 bytes
  mySCREEN SCR_WIDTH SCR_HEIGHT * bl fill \ fill this memory with 'space'
```

L'inconvénient, avec cette méthode de programmation, les dimensions sont définies dans des constantes, donc en dehors du tableau. Il serait plus intéressant d'embarquer les dimensions du tableau dans le tableau. Pour ce faire, on va définir une structure adaptée à ce cas :

```
structures
struct cARRAY
  i8 field ->cARRAY-width
  i8 field ->cARRAY-height
  i8 field ->cARRAY-content

: cArray-size@ { addr -- datas-size }
  addr ->cARRAY-width c@
  addr ->cARRAY-height c@ *
;
```

```

create myVscreen      \ define a screen 8x32 bytes
    32 c,              \ compile width
    08 c,              \ compile height
    myVscreen cArray-size@ allot

```

Pour définir un sprite logiciel, on va mutualiser très simplement cette définition :

```

structures
struct cARRAY
    i8 field ->cARRAY-width
    i8 field ->cARRAY-height
    i8 field ->cARRAY-content

: cArray-width@ { addr -- width }
    addr ->cARRAY-width c@
;

: cArray-height@ { addr -- height }
    addr ->cARRAY-height c@
;

: cArray-size@ { addr -- datas-size }
    addr cArray-width@
    addr cArray-height@ *
;

```

Voici comment définir un sprite 5 x 7 octets:

```

create char3
    5 c, 7 c,      \ compile width and height
    $20 c, $db c, $db c, $db c, $20 c,
    $db c, $20 c, $20 c, $20 c, $db c,
    $20 c, $20 c, $20 c, $20 c, $db c,
    $20 c, $db c, $db c, $db c, $20 c,
    $20 c, $20 c, $20 c, $20 c, $db c,
    $db c, $20 c, $20 c, $20 c, $db c,
    $20 c, $db c, $db c, $db c, $20 c,

```

Pour l'affichage du sprite, à partir d'une position x y dans la fenêtre du terminal, une simple boucle suffit :

```

: .sprite { xpos ypos sprite-addr -- }
    sprite-addr cArray-height@ 0 do
        xpos ypos at-xy
        sprite-addr cArray-width@ i *      \ calculate offset in sprite
    datas
        sprite-addr ->cARRAY-content +      \ calculate real address for
    line n in sprite datas
        sprite-addr cArray-width@ type      \ display line
        1 +to ypos                          \ increment y position

```

```

    loop
;

0 constant blackColor
1 constant redColor
4 constant blueColor
10 02 char3 .sprite
redColor fg
16 02 char3 .sprite
blueColor fg
22 02 char3 .sprite
blackColor fg
cr cr

```

```

ok
-->
ok
-->
ok
-->
ok
-->
ok
-->
ok
--> blackColor fg
ok

```

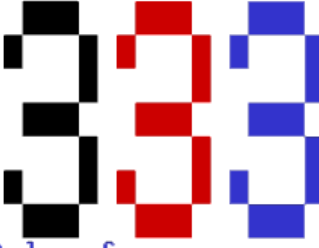


Figure 8: affichage de sprite

Résultat de l'affichage de notre sprite :

Voilà. C'est tout.

Les nombres réels avec eForth Windows

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

Ce n'est pas surprenant. De base, eForth Windows n'utilise que des nombres entiers 32 ou 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 Windows intègre des opérateurs traitant des nombres réels.

Les nombres réels sont aussi dénommés nombres à virgule flottante.

Les réels avec eForth Windows

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 Windows

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 Windows 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 Windows 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 Windows 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 Windows 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 Windows 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 $r1 \leq r2$.
- **f<>** (r1 r2 -- fl) fl est vrai si $r1 \neq r2$.
- **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 $r1 \geq r2$.

Transformations entiers ↔ réels

eForth Windows 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 Windows 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 Windows;
- **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 Windows, 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
  LOOP
```

```

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

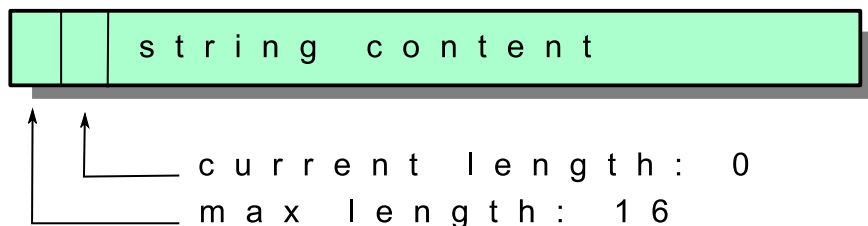
Les variables alpha-numérique texte n'existent pas nativement dans eForth Windows. Voici le premier essai de définition du mot **string** :

```
\ 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 --- f1)
  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 from 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 chaîne :

```

--> 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 vocabulaires avec eForth Windows

En FORTH, la notion de procédure et de fonction n'existe pas. Les instructions FORTH s'appellent des MOTS. A l'instar d'une langue traditionnelle, FORTH organise les mots qui le composent en VOCABULAIRES, ensemble de mots ayant un trait commun.

Programmer en FORTH consiste à enrichir un vocabulaire existant, ou à en définir un nouveau, relatif à l'application en cours de développement.

Liste des vocabulaires

Un vocabulaire est une liste ordonnée de mots, recherchés du plus récemment créé au moins récemment créé. L'ordre de recherche est une pile de vocabulaires. L'exécution du nom d'un vocabulaire remplace le haut de la pile d'ordre de recherche par ce vocabulaire.

Pour voir la liste des différents vocabulaires disponibles dans ESP32forth, on va utiliser le mot **voclist**:

```
--> internals voclist      \ affiche
internals
graphics
ansi
editor
streams
tasks
windows
structures
recognizers
internalized
internals
FORTH
```

Cette liste n'est pas limitée. Des vocabulaires supplémentaires peuvent apparaître si on compile certaines extensions.

Le principal vocabulaire s'appelle **FORTH**. Tous les autres vocabulaires sont rattachés au vocabulaire **FORTH**.

Les vocabulaires essentiels

Voici la liste des principaux vocabulaires disponibles dans ESP32forth :

- **ansi** gestion de l'affichage dans un terminal ANSI ;
- **editor** donne accès aux commandes d'édition des fichiers de type bloc ;
- **structures** gestion de structures complexes ;

Liste du contenu d'un vocabulaire

Pour voir le contenu d'un vocabulaire, on utilise le mot **vlist** en ayant préalablement sélectionné le vocabulaire adéquat:

```
graphics vlist
```

Sélectionne le vocabulaire **graphics** et affiche son contenu:

```
--> graphics vlist    \ affiche:
flip poll wait 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
```

La sélection d'un vocabulaire donne accès aux mots définis dans ce vocabulaire.

Par exemple, le mot **voclist** n'est pas accessible sans invoquer d'abord le vocabulaire **internals**.

Un même mot peut être défini dans deux vocabulaires différents et avoir deux actions différentes.

Utilisation des mots d'un vocabulaire

Pour compiler un mot défini dans un autre vocabulaire que FORTH, il y a deux solutions. La première solution consiste à appeler simplement ce vocabulaire avant de définir le mot qui va utiliser des mots de ce vocabulaire.

Ici, on définit un mot **SDL2.dll** qui utilise le mot **dll** défini dans le vocabulaire **windows**:

```
\ Entry point to SDL2.dll library
windows
z" SDL2.dll" dll SDL2.dll
```

Chainage des vocabulaires

L'ordre de recherche d'un mot dans un vocabulaire peut être très important. En cas de mots ayant un même nom, on lève toute ambiguïté en maîtrisant l'ordre de recherche dans les différents vocabulaires qui nous intéressent.

Avant de créer un chaînage de vocabulaires, on restreint l'ordre de recherche avec le mot **only**:

```
windows
order    \ affiche:      windows >> FORTH
only
order    \ affiche:      FORTH
```

On duplique ensuite le chaînage des vocabulaires avec le mot **also**:

```
only
```

```

order    \ affiche:      FORTH
windows also
order    \ affiche:      windows >> FORTH
structures
order    \ affiche:
        \                structures >> FORTH
        \                windows >> FORTH

```

Voici une séquence de chaînage compacte:

```

vocabulary SDL2
only FORTH also windows also structures also
SDL2 definitions

```

Le dernier vocabulaire ainsi chaîné sera le premier exploré quand on exécutera ou compilera un nouveau mot.

```

order      \ affiche:      SDL2 >> FORTH
           \                structures >> FORTH
           \                windows >> FORTH
           \                FORTH

```

L'ordre de recherche, ici, commencera par le vocabulaire **SDL2**, puis **structures**, puis **windows** et pour finir, le vocabulaire **FORTH**:

- si le mot recherché n'est pas trouvé, il y a une erreur de compilation;
- si le mot est trouvé dans un vocabulaire, c'est ce mot qui sera compilé, même s'il est défini dans le vocabulaire suivant;

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 abrégons 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'initialisation 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 précedence 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 précedence 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" Saturday" ;
: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 eForth Windows, 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 eForth Windows, 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 Windows, 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 eForth Windows. 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 Windows plus lisible. On détecte facilement toutes les constantes nommant un registre ;
- 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 Windows. 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 Windows :

```

\ Work well in eForth Windows
: 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 Windows et tapez :

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

collez le caractère 早. Eforth Windows 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 ?

Etendre le vocabulaire graphics pour Windows

eFORTH permet l'accès aux librairies des fonctions Windows grâce au mot **dll**.

Dans le code source de eForth, voici comment s'effectue la connexion à la librairie **Gdi32** :

```
windows definitions
z" Gdi32.dll" dll Gdi32
```

Ici, le mot **Gdi32** devient le point d'entrée pour définir les mots donnant accès à cette librairie **Gdi32.dll**.

A partir de ce moment, chaque mot défini pour eFORTH utilisant cette librairie **Gdi32** se réfère à la documentation Microsoft :

<https://learn.microsoft.com/en-us/windows/win32/api/wingdi/>

Ici, on va chercher la documentation de la fonction **LineTo** :

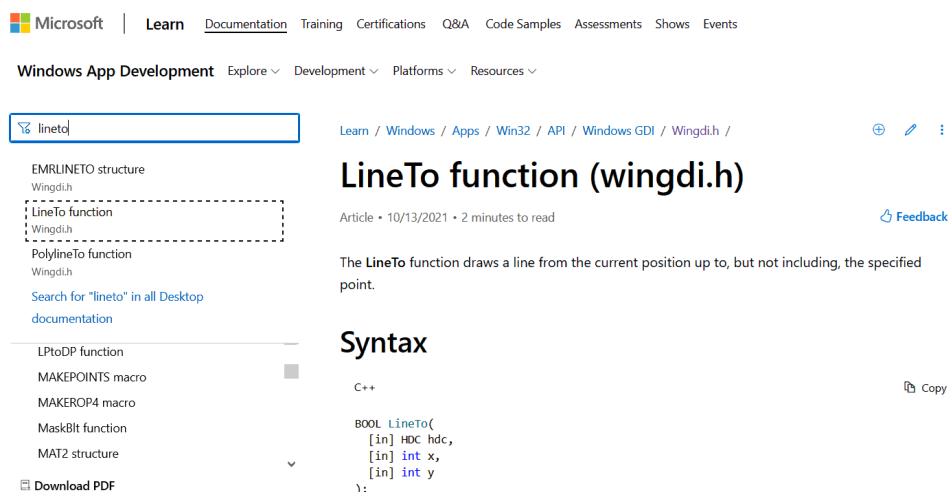
The image is a screenshot of the Microsoft Learn website. At the top, there's a navigation bar with 'Microsoft | Learn' and various links like 'Documentation', 'Training', 'Certifications', etc. Below this, a breadcrumb trail shows 'Windows App Development' > 'Explore' > 'Development' > 'Platforms' > 'Resources'. A search bar on the left contains the text 'lineto'. Below the search bar, a list of search results is shown, including 'EMRLINEETO structure', 'LineTo function', 'PolylineTo function', and 'LPTODP function'. The 'LineTo function' result is highlighted. To the right of the search results, the main content area displays the title 'LineTo function (wingdi.h)' with a sub-header 'Article • 10/13/2021 • 2 minutes to read'. Below the title, a brief description states: 'The LineTo function draws a line from the current position up to, but not including, the specified point.' Further down, the 'Syntax' section is visible, showing the C++ signature: 'BOOL LineTo([in] HDC hdc, [in] int x, [in] int y);'. A 'Copy' button is located next to the syntax.

Figure 9: documentation de LineTo

Dans cette documentation, pour la fonction **LineTo**, il est indiqué:

- accepte trois paramètres en entrée: hdc, x et y
- rend un paramètre en sortie: fl

Le premier réflexe, serait donc de définir le mot eFORTH **LineTo** comme ceci :

```
z" LineTo"      3 Gdi32 LineTo ( hdc x y -- fl )
```

La valeur 3 qui précède le mot **Gdi32** indique que la fonction appelée doit utiliser trois paramètres.

Si on accepte d'utiliser le mot **LineTo** de cette manière, nous serions obligé, à chaque utilisation de procéder ainsi :

```
graphics internals
: drawLines ( -- )
  hdc 20 20 LineTo drop
  hdc 50 20 LineTo drop
  hdc 50 50 LineTo drop
  hdc 20 50 LineTo drop
  hdc 45 45 LineTo drop
;
```

L'appel systématique au ticket **hdc** n'est pas nécessaire si on gère une seule fenêtre Windows. De même, l'utilisation de **drop** après **LineTo** alourdit le code eFORTH. La solution, pour simplifier ces mots consiste à les définir sous deux formes dans deux vocabulaires différents.

Définition des fonctions dans graphics internals

Les mots eFORTH réalisant un appel direct aux fonctions de la librairie **Gdi32** seront renommés en les préfixant avec Gdi:

- **Gdi.LineTo** accède à **LineTo**;
- **Gdi.Rectangle** accède à **Rectangle**, etc.

Tous les mots seront définis dans le vocabulaire **graphics internals** :

```
graphics internals definitions
windows also

\ The LineTo function draw a line.
z" LineTo"      3 Gdi32 Gdi.LineTo ( hdc x y -- fl )

z" Rectangle"   5 gdi32 Gdi.Rectangle ( hdc left top right bottom -- fl )

z" Ellipse"     5 gdi32 Gdi.Ellipse ( hdc left top right bottom -- fl )

\ The CloseFigure function close a figure in a path.
z" CloseFigure" 1 gdi32 Gdi.CloseFigure ( hdc -- fl )

\ The GetPixel function retrieves the red, green, blue (RGB) color value
\ of the pixel at the specified coordinates.
z" GetPixel"    3 gdi32 Gdi.GetPixel ( hdc x y -- color )

\ The SetPixel function sets the pixel at the specified coordinates
\ to the specified color.
z" SetPixel"    4 gdi32 Gdi.SetPixel ( hdc x y colorref -- colorref )
```


Il est aisé de vérifier la bonne compilation de ces mots dans le vocabulaire **graphics internals** :

```
Gdi.SetPixel Gdi.GetPixel Gdi.CloseFigure Gdi.Ellipse Gdi.Rectangle
Gdi.LineTo
GrfWindowProc msg>pressed msg>button rescale binfo msgbuf ps hdc hwnd
GrfClass
hinstance GrfWindowTitle GrfClassName 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
```

Ici, on a mis en évidence les nouveaux mots eFORTH connectés aux fonctions de la librairie **Gdi32**.

Définition des mots dans graphics

Nous allons maintenant définir des mots graphiques simplifiés dans le vocabulaire **graphics**:

```
only forth
graphics definitions internals

: lineTo ( x y -- )
  hdc -rot Gdi.LineTo drop ;
: rectangle ( left top right bottom -- )
  >r >r >r >r hdc r> r> r> r> Gdi.Rectangle drop ;
: closeFigure ( -- )
  hdc Gdi.CloseFigure drop ;
: getPixel ( x y -- colorref )
  hdc -rot Gdi.GetPixel ;
: setPixel ( x y color -- )
  hdc -rot Gdi.SetPixel ;
```

Pour utiliser ces mots, reprenons notre exemple :

```
graphics
: drawLines ( -- )
  20 20 lineTo
  50 20 lineTo
  50 50 lineTo
  20 50 lineTo
  45 45 lineTo
;
```

Pour conclure, ne cherchez pas à étendre eFORTH avec toutes les fonctions de chaque librairie. Vous y passeriez des années!

La stratégie la plus rapide et la plus simple consiste à définir exclusivement les mots exploitant les fonctions qui vous intéressent. La programmation Windows est très complexe et nécessite l'acquisition de bases solides.

Contenu détaillé des vocabulaires eForth Windows

Eforth Windows 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

=	-1	-rot	_	.	:	:noname	!
?	?do	?dup	.	."	.s	'	(local)
[[']	[char]	[ELSE]	[IF]	[THEN]	l	{
}transfer	@	*	*/	*/MOD	/	/mod	#
#!	#>	#fs	#s	#tib	+	+!	+loop
+to	<	<#	<=	<>	=	>	>=
>BODY	>flags	>flags&	>in	>link	>link&	>name	>params
>R	>size	0	0<	0<>	0=	1	1-
1/F	1+	10	2!	2@	2*	2/	2drop
2dup	3dup	4*	4/	41	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	CELL	cell/	cell+	cells
char	CLOSE-FILE	cmove	cmove>	CONSTANT	context	copy	cp
cr	CREATE	CREATE-FILE	current	decimal	default-key	default-key?	
default-type		default-use	defer	DEFINED?	definitions	DELETE-FILE	depth
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	FORTH	forth-
builtins							
FOVER	FP!	FP@	fp0	free	FROT	FSIN	FSINCOS
FSORT	FSWAP	fvariable	graphics	here	hex	hld	hold
I	if	IMMEDIATE	include	included	included?	internals	invert
is	J	K	key	key?	L!	latestxt	leave
list	literal	load	loop	LSHIFT	max	min *	mod
ms	ms-ticks	mv	n.	needs	negate	next	nip
nl	NON-BLOCK	normal	octal	OF	ok	only	open-
blocks							
OPEN-FILE	OR	order	OVER	pad	page	PARSE	pause
pause?	PI	postpone	postpone,	precision	previous	prompt	quit
r"	R@	R/O	R/W	R>	r 	r~	rdrop
READ-FILE	recognizers	recurse	refill	remaining	remember	RENAME-FILE	repeat
REPOSITION-FILE		required	reset	resize	RESIZE-FILE	restore	revive
rm	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@	SP!	SP@	sp0	space	spaces	start-task	
startswith?							
startup:	state	str	str=	streams	structures	SW@	SWAP
task	tasks	terminate	then	throw	thru	tib	to
touch	transfer	transfer{	type	u.	U/MOD	U<	UL@
UNLOOP	until	update	use	used	UW@	value	VARIABLE
visual	vlist	vocabulary	W!	W/O	while	windows	words
WRITE-FILE	XOR	z"	z>s				

windows

[process-heap](#) [HeapReAlloc](#) [HeapFree](#) [HeapAlloc](#) [GetProcessHeap](#) [WM_>name](#) [WM_PENWINLAST](#)
[WM_PENEVENT](#) [WM_CTLINIT](#) [WM_PENMISC](#) [WM_PENCTL](#) [WM_HEDITCTL](#) [WM_SKB](#) [WM_PENMISCINFO](#)
[WM_GLOBALRCCHANGE](#) [WM_HOOKRCRESULT](#) [WM_RCRESULT](#) [WM_PENWINFIRST](#) [WM_AFXLAST](#)
[WM_AFXFIRST](#) [WM_HANDHELDLAST](#) [WM_HANDHELDFIRST](#) [WM_APPCOMMAND](#) [WM_PRINTCLIENT](#)
[WM_PRINT](#) [WM_HOTKEY](#) [WM_PALETTECHANGED](#) [WM_PALETTEISCHANGING](#) [WM_QUERYNEWPALETTE](#)
[WM_HSCROLLCLIPBOARD](#) [WM_CHANGECHAIN](#) [WM_ASKCBFORMATNAME](#) [WM_SIZECLIPBOARD](#)
[WM_VSCROLLCLIPBOARD](#) [WM_PAINTCLIPBOARD](#) [WM_DRAWCLIPBOARD](#) [WM_DESTROYCLIPBOARD](#)
[WM_RENDERALLFORMATS](#) [WM_RENDERFORMAT](#) [WM_UNDO](#) [WM_CLEAR](#) [WM_PASTE](#) [WM_COPY](#) [WM_CUT](#)
[WM_MOUSELEAVE](#) [WM_NCMOUSELEAVE](#) [WM_MOUSEHOVER](#) [WM_NCMOUSEHOVER](#) [WM_IME_KEYUP](#)
[WM_IMEKEYUP](#) [WM_IME_KEYDOWN](#) [WM_IMEKEYDOWN](#) [WM_IME_REQUEST](#) [WM_IME_CHAR](#) [WM_IME_SELECT](#)
[WM_IME_COMPOSITIONFULL](#) [WM_IME_CONTROL](#) [WM_IME_NOTIFY](#) [WM_IME_SETCONTEXT](#) [WM_IME_REPORT](#)
[WM_MDIREFRESHMENU](#) [WM_DROPFILES](#) [WM_EXITSIZEMOVE](#) [WM_ENTERSIZEMOVE](#) [WM_MDISETMENU](#)
[WM_MDIGETACTIVE](#) [WM_MDIICONARRANGE](#) [WM_MDICASCADE](#) [WM_MDITILE](#) [WM_MDIMAXIMIZE](#)
[WM_MDINEXT](#) [WM_MDIRESTORE](#) [WM_MDIACTIVATE](#) [WM_MDIDESTROY](#) [WM_MDICREATE](#) [WM_DEVICECHANGE](#)
[WM_POWERBROADCAST](#) [WM_MOVING](#) [WM_CAPTURECHANGED](#) [WM_SIZING](#) [WM_NEXTMENU](#) [WM_EXITMENULOOP](#)
[WM_ENTERMENULOOP](#) [WM_PARENTNOTIFY](#) [WM_MOUSEHWHEEL](#) [WM_XBUTTONDOWNBLCLK](#) [WM_XBUTTONUP](#)
[WM_XBUTTONDOWN](#) [WM_MOUSEWHEEL](#) [WM_MOUSELAST](#) [WM_MBUTTONDOWNBLCLK](#) [WM_MBUTTONUP](#)
[WM_MBUTTONDOWN](#) [WM_RBUTTONDOWNBLCLK](#) [WM_RBUTTONUP](#) [WM_RBUTTONDOWN](#) [WM_LBUTTONDOWNBLCLK](#)
[WM_LBUTTONUP](#) [WM_LBUTTONDOWN](#) [WM_MOUSEMOVE](#) [WM_MOUSEFIRST](#) [CB_MSGMAX](#) [CB_GETCOMBOBOXINFO](#)
[CB_MULTIPLEADDSTRING](#) [CB_INITSTORAGE](#) [CB_SETDROPPEDWIDTH](#) [CB_GETDROPPEDWIDTH](#)
[CB_SETHORIZONTALEXTENT](#) [CB_GETHORIZONTALEXTENT](#) [CB_SETTOPINDEX](#) [CB_GETTOPINDEX](#)

CB_GETLOCALE CB_SETLOCALE CB_FINDSTRINGEXACT CB_GETDROPPEDSTATE CB_GETEXTENDEDUI
CB_SETEXTENDEDUI CB_GETITEMHEIGHT CB_SETITEMHEIGHT CB_GETDROPPEDCONTROLRECT
CB_SETITEMDATA CB_GETITEMDATA CB_SHOWDROPDOWN CB_SETCURSEL CB_SELECTSTRING
CB_FINDSTRING CB_RESETCONTENT CB_INSERTSTRING CB_GETLBTEXTLEN CB_GETLBTEXT
CB_GETCURSEL CB_GETCOUNT CB_DIR CB_DELETETEXT CB_ADDSTRING CB_SETEXTITSEL
CB_LIMITTEXT CB_GETEDITSEL WM_CTLCOLORSTATIC WM_CTLCOLORSCROLLBAR WM_CTLCOLORDLG
WM_CTLCOLORBTN WM_CTLCOLORLISTBOX WM_CTLCOLOREDIT WM_CTLCOLORMSGBOX WM_LBTRACKPOINT
WM_QUERYUISTATE WM_UPDATEUISTATE WM_CHANGEUISTATE WM_MENUCOMMAND WM_UNINITMENUPOPUP
WM_MENUGETOBJECT WM_MENUDRAG WM_MENUBUTTONUP WM_ENTERIDLE WM_MENUCAR
WM_MENUSELECT WM_SYSTIMER WM_INITMENUPOPUP WM_INITMENU WM_VSCROLL WM_HSCROLL
WM_TIMER WM_SYSCOMMAND WM_COMMAND WM_INITDIALOG WM_IME_KEYLAST WM_IME_COMPOSITION
WM_IME_ENDCOMPOSITION WM_IME_STARTCOMPOSITION WM_INTERIM WM_CONVERTRESULT
WM_CONVERTREQUEST WM_WNT_CONVERTREQUESTEX WM_UNICHAR WM_SYSDEADCHAR WM_SYSCHAR
WM_SYSKEYUP WM_SYSKEYDOWN WM_DEADCHAR WM_CHAR WM_KEYUP WM_KEYDOWN WM_INPUT
BM_SETDONTCLICK BM_SETIMAGE BM_GETIMAGE BM_CLICK BM_SETSTYLE BM_SETSTATE
BM_GETSTATE BM_SETCHECK BM_GETCHECK SBM_GETSCROLLBARINFO SBM_GETSCROLLINFO
SBM_SETSCROLLINFO SBM_SETRANGEREDRAW SBM_ENABLE_ARROWS SBM_GETRANGE SBM_SETRANGE
SBM_GETPOS SBM_SETPOS EM_GETIMESTATUS EM_SETIMESTATUS EM_CHARFROMPOS EM_POSFROMCHAR
EM_GETLIMITTEXT EM_GETMARGINS EM_SETMARGINS EM_GETPASSWORDCHAR EM_GETWORDBREAKPROC
EM_SETWORDBREAKPROC EM_SETREADONLY EM_GETFIRSTVISIBLELINE EM_EMPTYUNDOBUFFER
EM_SETPASSWORDCHAR EM_SETTABSTOPS EM_SETWORDBREAK EM_LINEFROMCHAR EM_FMTLINES
EM_UNDO EM_CANUNDO EM_SETLIMITTEXT EM_LIMITTEXT EM_GETLINE EM_SETFONT EM_REPLACESEL
EM_LINELENGTH EM_GETTHUMB EM_GETHANDLE EM_SETHANDLE EM_LINEINDEX EM_GETLINECOUNT
EM_SETMODIFY EM_GETMODIFY EM_SCROLLCARET EM_LINESCROLL EM_SCROLL EM_SETRECTNP
EM_SETRECT EM_GETRECT EM_SETSEL EM_GETSEL WM_NCXBUTTONDBLCLK WM_NCXBUTTONUP
WM_NCXBUTTONDOWN WM_NCMBUTTONDBLCLK WM_NCMBUTTONUP WM_NCMBUTTONDOWN
WM_NCRBUTTONDBLCLK
WM_NCRBUTTONUP WM_NCRBUTTONDOWN WM_NCLBUTTONDBLCLK WM_NCLBUTTONUP WM_NCLBUTTONDOWN
WM_NCMOUSEMOVE WM_SYNCPAINT WM_GETDLGCODE WM_NCACTIVATE WM_NCPAINT WM_NCHITTEST
WM_NCCALCSIZE WM_NCDESTROY WM_NCCREATE WM_SETICON WM_GETICON WM_DISPLAYCHANGE
WM_STYLECHANGED WM_STYLECHANGING WM_CONTEXTMENU WM_NOTIFYFORMAT WM_USERCHANGED
WM_HELP WM_TCARD WM_INPUTLANGCHANGE WM_INPUTLANGCHANGEREQUEST WM_NOTIFY
WM_CANCELJOURNAL WM_COPYDATA WM_COPYGLOBALDATA WM_POWER WM_WINDOWPOSCHANGED
WM_WINDOWPOSCHANGING WM_COMMNOTIFY WM_COMPACTING WM_GETOBJECT WM_COMPAREITEM
WM_QUERYDRAGICON WM_GETHOTKEY WM_SETHOTKEY WM_GETFONT WM_SETFONT WM_CHARTOITEM
WM_VKEYTOITEM WM_DELETEITEM WM_MEASUREITEM WM_DRAWITEM WM_SPOOLERSTATUS
WM_NEXTDLGCTL WM_ICONERASEBKGD WM_PAINTICON WM_GETMINMAXINFO WM_QUEUESYNC
WM_CHILDACTIVATE WM_MOUSEACTIVATE WM_SETCURSOR WM_CANCELMODE WM_TIMECHANGE
WM_FONTCHANGE WM_ACTIVATEAPP WM_DEVMODECHANGE WM_WININICHANGE WM_CTLCOLOR
WM_SHOWWINDOW WM_ENDSESSION WM_SYSCOLORCHANGE WM_ERASEBKGD WM_QUERYOPEN
WM_QUIT WM_QUERYENDSESSION WM_CLOSE WM_PAINT WM_GETTEXTLENGTH WM_GETTEXT
WM_SETTEXT WM_SETREDRAW WM_ENABLE WM_KILLFOCUS WM_SETFOCUS WM_ACTIVATE
WM_SIZE WM_MOVE WM_DESTROY WM_CREATE WM_NULL SRCCOPY DIB_RGB_COLORS BI_RGB
->bmiColors ->bmiHeader BITMAPINFO ->biClrImportant ->biClrUsed ->biYpelsPerMeter
->biXPelsPerMeter ->biSizeImage ->biCompression ->biBitCount ->biPlanes
->biHeight ->biWidth ->biSize BITMAPINFOHEADER ->rgbReserved ->rgbRed ->rgbGreen
->rgbBlue RGBQUAD StretchDIBits DC_PEN DC_BRUSH DEFAULT_GUI_FONT SYSTEM_FIXED_FONT
DEFAULT_PALETTE DEVICE_DEFAULT_PALETTE SYSTEM_FONT ANSI_VAR_FONT ANSI_FIXED_FONT
OEM_FIXED_FONT BLACK_PEN WHITE_PEN NULL_BRUSH BLACK_BRUSH DKGRAY_BRUSH
GRAY_BRUSH LTGRAY_BRUSH WHITE_BRUSH GetStockObject COLOR_WINDOW RGB
CreateSolidBrush
DeleteObject Gdi32 dpi-aware SetThreadDpiAwarenessContext VK_ALT GET_X_LPARAM
GET_Y_LPARAM IDI_INFORMATION IDI_ERROR IDI_WARNING IDI_SHIELD IDI_WINLOGO

```

IDI_ASTERISK IDI_EXCLAMATION IDI_QUESTION IDI_HAND IDI_APPLICATION LoadIconA
IDC_HELP IDC_APPSTARTING IDC_HAND IDC_NO IDC_SIZEALL IDC_SIZENS IDC_SIZEWE
IDC_SIZENESW IDC_SIZENWSE IDC_ICON IDC_SIZE IDC_UPARROW IDC_CROSS IDC_WAIT
IDC_IBEAM IDC_ARROW LoadCursorA PostQuitMessage FillRect ->rgbReserved
->fIncUpdate ->fRestore ->rcPaint ->fErase ->hdc PAINTSTRUCT EndPaint BeginPaint
GetDC PM_NOYIELD PM_REMOVE PM_NOREMOVE ->lPrivate ->pt ->time ->lParam
->wParam ->message ->hwnd MSG DispatchMessageA TranslateMessage PeekMessageA
GetMessageA ->bottom ->right ->top ->left RECT ->y ->x POINT CW_USEDEFAULT
IDI_MAIN_ICON DefaultInstance WS_TILEDWINDOW WS_POPUPWINDOW WS_OVERLAPPEDWINDOW
WS_CAPTION WS_TILED WS_ICONIC WS_CHILDWINDOW WS_GROUP WS_TABSTOP WS_POPUP
WS_CHILD WS_MINIMIZE WS_VISIBLE WS_DISABLED WS_CLIPSIBLINGS WS_CLIPCHILDREN
WS_MAXIMIZE WS_BORDER WS_DLGFRAME WS_VSCROLL WS_HSCROLL WS_SYSMENU WS_THICKFRAME
WS_MINIMIZEBOX WS_MAXIMIZEBOX WS_OVERLAPPED CreateWindowExA callback DefWindowProcA
SetForegroundWindow SW_SHOWMAXIMIZED SW_SHOWNORMAL SW_FORCEMINIMIZE SW_SHOWDEFAULT
SW_RESTORE SW_SHOWNA SW_SHOWNOACTIVE SW_MINIMIZE SW_SHOW SW_SHOWNOACTIVATE
SW_MAXIMIZED SW_SHOWMINIMIZED SW_NORMAL SW_HIDE ShowWindow ->lpszClassName
->lpszMenuName ->hbrBackground ->hCursor ->hIcon ->hInstance ->cbWndExtra
->cbClsExtra ->lPFNWndProc ->style WINDCLASSA RegisterClassA MB_CANCELTRYCONTINUE
MB_RETRYCANCEL MB_YESNO MB_YESNOCANCEL MB_ABORTRETRYIGNORE MB_OKCANCEL
MB_OK MessageBoxA User32 win-key win-key? raw-key win-type init-console
console-mode stderr stdout stdin console-started FlushConsoleInputBuffer
SetConsoleMode GetConsoleMode GetStdHandle ExitProcess AllocConsole
ENABLE_LVB_GRID_WORLDWIDE
DISABLE\_NEWLINE\_AUTO\_RETURN ENABLE_VIRTUAL_TERMINAL_PROCESSING
ENABLE_WRAP_AT_EOL_OUTPUT
ENABLE_PROCESSED_OUTPUT ENABLE_VIRTUAL_TERMINAL_INPUT ENABLE_QUICK_EDIT_MODE
ENABLE\_INSERT\_MODE ENABLE_MOUSE_INPUT ENABLE_WINDOW_INPUT ENABLE_ECHO_INPUT
ENABLE_LINE_INPUT ENABLE\_PROCESSED\_INPUT STD_ERROR_HANDLE STD_OUTPUT_HANDLE
STD_INPUT_HANDLE invalid?ior d0NULL wargs-convert wz>sz wargv
wargc CommandLineToArgvW Shell32 GetModuleHandleA GetCommandLineW GetLastError
WaitForSingleObject GetTickCount Sleep ExitProcess Kernel32 contains? dll
sofunc GetProcAddress LoadLibraryA WindowProcShim SetupCtrlBreakHandler
windows-builtins calls

```

Ressources

En anglais

- **ESP32forth** page maintenue par Brad NELSON, le créateur de ESP32forth. Vous y trouverez toutes les versions (ESP32, Windows, Web, Linux...) <https://esp32forth.appspot.com/ESP32forth.html>

En français

- **eForth** site en deux langues (français, anglais) avec plein d'exemples <https://eforth.com.tw/academy/library.htm>

GitHub

- **Ueforth** ressources maintenues par Brad NELSON. Contient tous les fichiers sources en Forth et en langage C de ESP32forth et ueForth Windows, Linux et web. <https://github.com/flagxor/ueforth>
- **eForth Windows** codes sources et documentation pour eForth Windows. Ressources maintenues par Marc PETREMANN <https://github.com/MPETREMANN11/eForth-Windows>
- **eForth SDL2 project** pour eForth Windows <https://github.com/MPETREMANN11/SDL2-eForth-windows>

Facebook

- **Eforth** groupe pour eForth Windows <https://www.facebook.com/groups/785868495783000>

Index lexical

1/F.....	48	FATAN2.....	49	SF@.....	48
ansi.....	60	fconstant.....	48	SPACE.....	56
BASE.....	51	FCOS.....	49	struct.....	40
BINARY.....	51	field.....	40	structures.....	40, 60
c!.....	29	forget.....	26	to.....	33
c@.....	29	FORTH.....	75	value.....	30
cell.....	36	fsqrt.....	48	variable.....	29
constant.....	30	fvariable.....	48	variables locales.....	32
create.....	67	graphics.....	72	voclist.....	60
DECIMAL.....	51	HEX.....	51	26
defer.....	64	HOLD.....	52	26
dll.....	71	i8.....	41	:noname.....	65
DOES>.....	67	is.....	64	55
drop.....	28	mémoire.....	29	.s.....	24
dump.....	24	order.....	62	{.....	32
dup.....	28	pile de retour.....	28	}.....	32
editor.....	60	pinyin.....	69	@.....	29
EMIT.....	54	S".....	55	#.....	52
EXECUTE.....	63	S>F.....	50	#>.....	52
f.....	47	see.....	24	#S.....	52
F**.....	48	set-precision.....	47	+to.....	33
F>S.....	49	SF!.....	48	<#.....	52