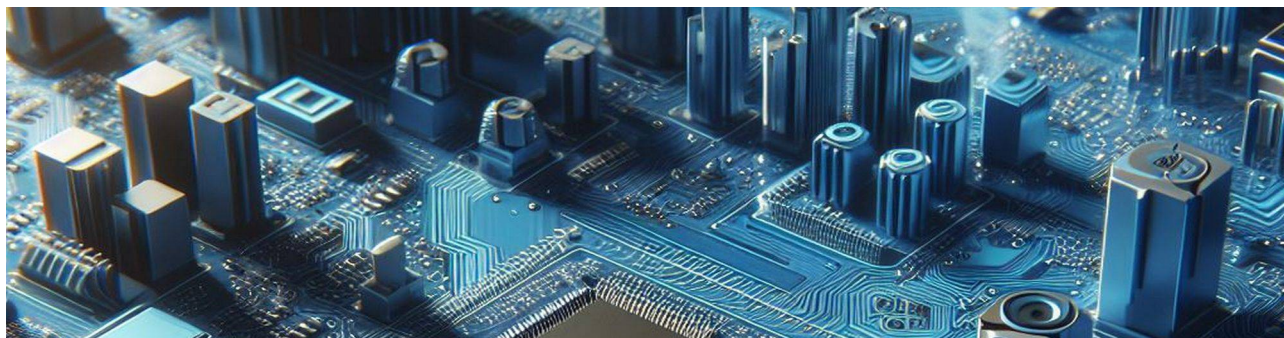


Le grand livre de eForth Windows

version 1.11 - 13 décembre 2024



Auteur

- Marc PETREMANN

Table des matières

Introduction.....	6
Aide à la traduction.....	6
Pourquoi programmer en langage Forth sur eForth Windows?.....	7
Préambule.....	7
Limites entre langage et application.....	7
C'est quoi un mot Forth?.....	8
Un mot c'est une fonction?.....	8
Le langage Forth comparé au langage C.....	9
Ce que Forth permet de faire par rapport au langage C.....	10
Mais pourquoi une pile plutôt que des variables?.....	11
Êtes-vous convaincus?.....	11
Existe-t-il des applications professionnelles écrites en Forth?.....	11
Installation sous Windows.....	13
Paramétrer eForth Windows.....	13
Un vrai Forth 64 bits avec eForth Windows.....	16
Les valeurs sur la pile de données.....	16
Les valeurs en mémoire.....	16
Traitement par mots selon taille ou type des données.....	17
Conclusion.....	18
Edition et gestion des fichiers sources pour eForth Windows.....	20
Les éditeurs de fichiers texte.....	20
Utiliser un IDE.....	21
Stockage sur GitHub.....	23
Quelques bonnes pratiques.....	23
Le fichier main.fs.....	24
Exemple d'organisation d'un projet.....	25
Commentaires et mise au point.....	27
Ecrire un code Forth lisible.....	27
Indentation du code source.....	28
Les commentaires.....	29
Les commentaires de pile.....	29
Signification des paramètres de pile en commentaires.....	30
Commentaires des mots de définition de mots.....	31
Les commentaires textuels.....	31
Commentaire en début de code source.....	32
Outils de diagnostic et mise au point.....	32
Le décompilateur.....	32
Dump mémoire.....	33
Moniteur de pile.....	33
Dictionnaire / Pile / Variables / Constantes.....	35
Étendre le dictionnaire.....	35

Gestion du dictionnaire.....	35
Piles et notation polonaise inversée.....	36
Manipulation de la pile de paramètres.....	37
La pile de retour et ses utilisations.....	37
Utilisation de la mémoire.....	38
Variables.....	38
Constantes.....	38
Valeurs pseudo-constantes.....	39
Outils de base pour l'allocation de mémoire.....	39
Les variables locales avec eForth Windows.....	40
Introduction.....	40
Le faux commentaire de pile.....	40
Action sur les variables locales.....	41
Structures de données pour eForth Windows.....	44
Préambule.....	44
Les tableaux en Forth.....	44
Tableau de données à une dimension.....	44
Mots de définition de tableaux.....	45
Lire et écrire dans un tableau.....	45
Exemple pratique de gestion d'écran.....	46
Gestion de structures complexes.....	48
Règles de nommage des structures et accesseurs.....	49
Choix de la taille des champs dans une structure.....	50
Définition de sprites.....	52
Les structures en détail.....	55
Les tailles de champs.....	55
Accès aux données dans les champs d'une structure.....	56
Gérer les accesseurs de structures.....	56
Les structures imbriquées.....	57
Evolution des structures depuis la version 7.0.7.21.....	59
Les types de champs dans les structures.....	59
Accès simplifié aux données d'un champ de structure.....	60
Les alias de type de données.....	60
Les nombres réels avec eForth Windows.....	63
Les réels avec eForth Windows.....	63
Précision des nombres réels avec eForth Windows.....	63
Constantes et variables réelles.....	64
Opérateurs arithmétiques sur les réels.....	64
Opérateurs mathématiques sur les réels.....	64
Opérateurs logiques sur les réels.....	65
Transformations entiers ↔ réels.....	65
Affichage des nombres et chaînes de caractères.....	67
Changement de base numérique.....	67
Définition de nouveaux formats d'affichage.....	68
Affichage des caractères et chaînes de caractères.....	70

Variables chaînes de caractères.....	72
Code des mots de gestion de variables texte.....	72
Ajout de caractère à une variable alphanumérique.....	74
Comparaisons et branchements.....	76
Branchements conditionnels vers l'avant.....	77
Branchement conditionnel vers l'arrière.....	78
Branchement en avant depuis une boucle indéfinie.....	79
Répétition contrôlée d'une action.....	80
Structure uni-conditionnelle à choix multiples.....	81
La récursivité.....	82
Les tests logiques.....	82
Les vocabulaires avec eForth Windows.....	84
Liste des vocabulaires.....	84
Les vocabulaires essentiels.....	84
Liste du contenu d'un vocabulaire.....	85
Utilisation des mots d'un vocabulaire.....	85
Chainage des vocabulaires.....	85
Les mots à action différée.....	87
Définition et utilisation de mots avec defer.....	88
Définition d'une référence avant.....	88
Un cas pratique.....	89
Les mots de création de mots.....	91
Utilisation de does>.....	91
Exemple de gestion de couleur.....	92
Gestion des paramètres entre eForth et l'API Windows.....	94
Passage des paramètres par la pile de données.....	94
Taille des données dans les structures.....	96
Affichage de boîtes modales.....	99
MessageBoxA.....	99
Contenu et comportement de la boîte de dialogue.....	100
Valeur retournée.....	101
Explications supplémentaires.....	102
Définition du bouton par défaut.....	103
Rajout d'une icône dans la boîte modale.....	103
Etendre le vocabulaire graphics pour Windows.....	106
Définition des mots dans graphics internals.....	107
Trouver les fonctions disponibles dans un fichier dll.....	108
Dependency Walker.....	108
Premiers tracés graphiques.....	111
Ouvrir une fenêtre.....	111
Tracé de lignes.....	112
Coloration des tracés graphiques.....	113
Tracé de formes géométriques.....	116
Colorier l'intérieur des formes.....	116

Tracé de rectangles.....	116
Tracé de polygones.....	118
Tracé d'ellipses.....	119
Tracé d'arcs.....	120
Afficher du texte dans l'environnement graphique.....	122
DrawTextA.....	122
Définition de la zone de tracé du texte.....	123
Formatage du texte.....	124
Changer la couleur du texte.....	126
TextOutA.....	127
Les fontes de caractères.....	129
Fonte et police.....	129
CreateFontA.....	129
COMx: Ouvrir et gérer un port série.....	134
Le port série.....	134
Ouverture du port série.....	135
Récupération des paramètres du port série.....	136
Modification des paramètres du port série.....	137
Initialisation du port série.....	138
Transmission et réception via le port série.....	139
Contenu détaillé des vocabulaires eForth Windows.....	140
Version v 7.0.7.21.....	141
FORTH.....	141
windows.....	142
Liste des fonctions graphiques de la librairie Gdi32.....	145
Bitmaps.....	145
Clip.....	146
Coordonnées et transformation.....	147
Couleurs.....	148
Pinceaux.....	148
Stylos.....	149
Lignes et courbes.....	149
Formes remplies.....	149
Fontes et textes.....	150
Contexte appareil.....	152
Régions.....	153
Ressources.....	154
En anglais.....	154
En français.....	154
GitHub.....	154
Facebook.....	155

Introduction

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

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

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

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

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

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

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

Aide à la traduction

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

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

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

Pour me contacter : petremann@arduino-forth.com

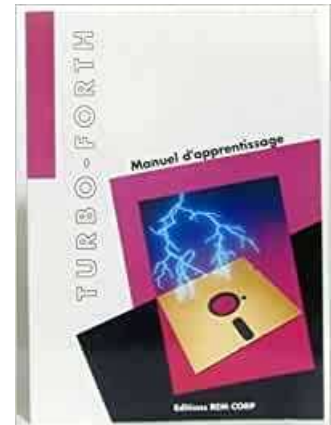
Pourquoi programmer en langage Forth sur eForth Windows?

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

Test avec **if** en langage Forth (extrait de code):

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

Voici l'initialisation de registres en langage C:

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

La même définition en langage Forth:

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

Ce que Forth permet de faire par rapport au langage C

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

```
_0_ __/ \ allume LED
```

Oui! La séquence **_0_ __/** est en langage Forth !

Bonne 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*

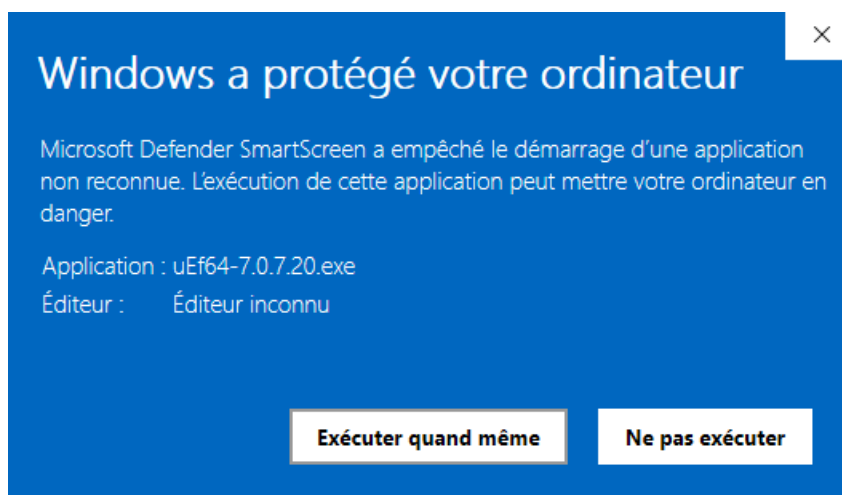


Figure 1: passer le message d'alerte Windows

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 :

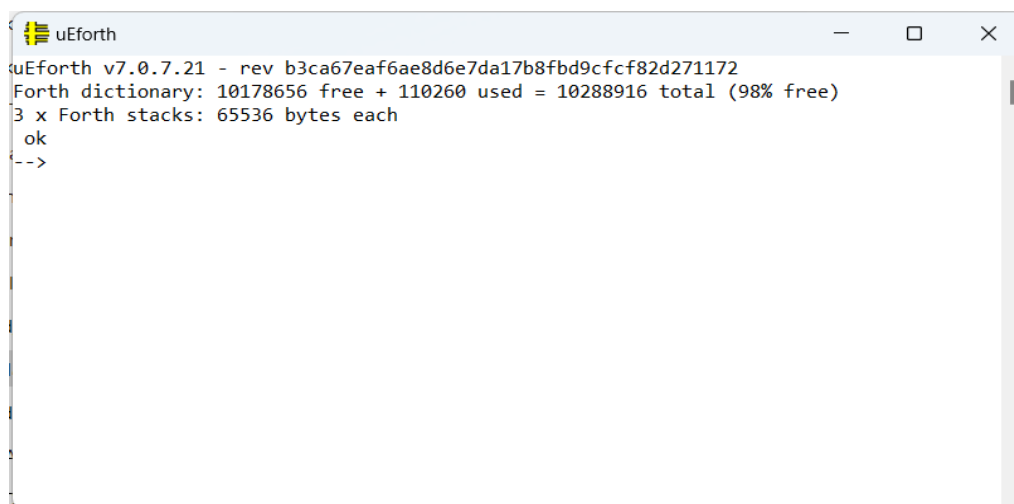


Figure 2: La fenêtre *μEforth* sous Windows

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* :

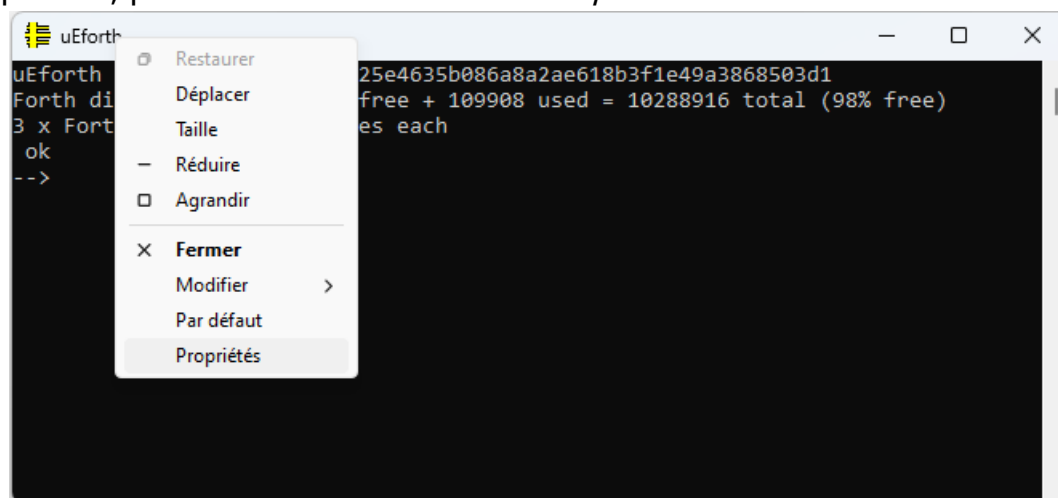


Figure 3: Sélection des propriétés d'affichage

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

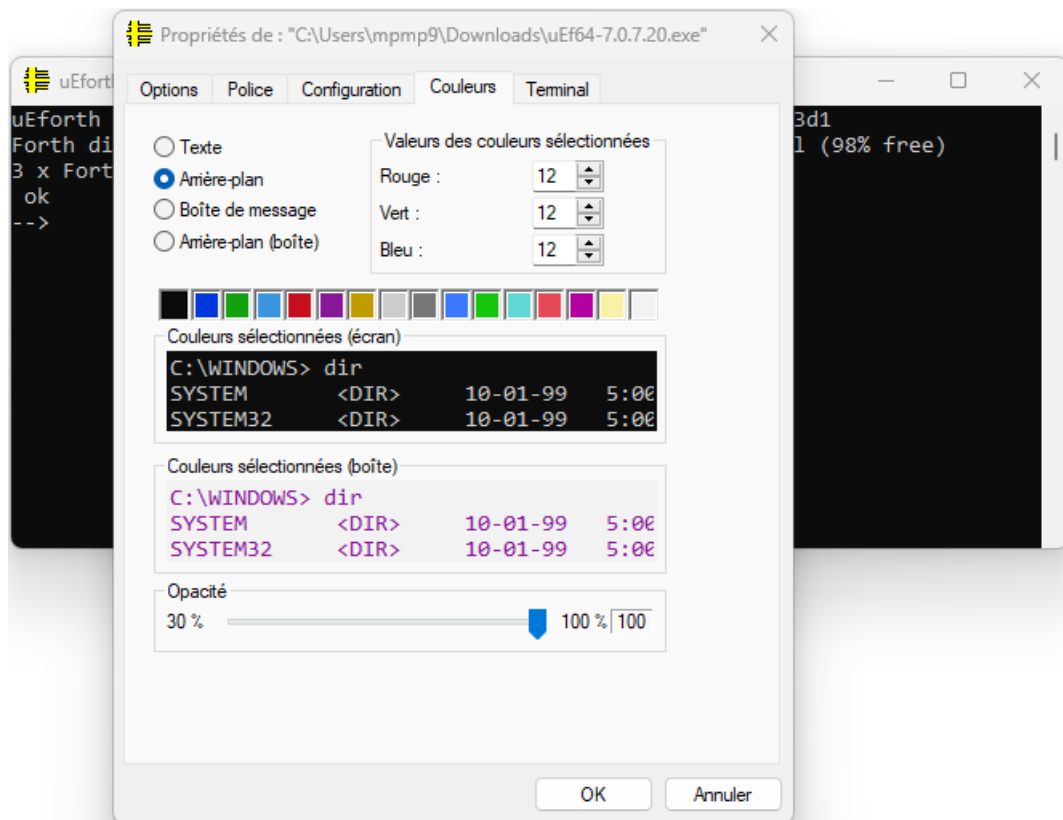


Figure 4: Choix des couleurs d'affichage

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.

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....

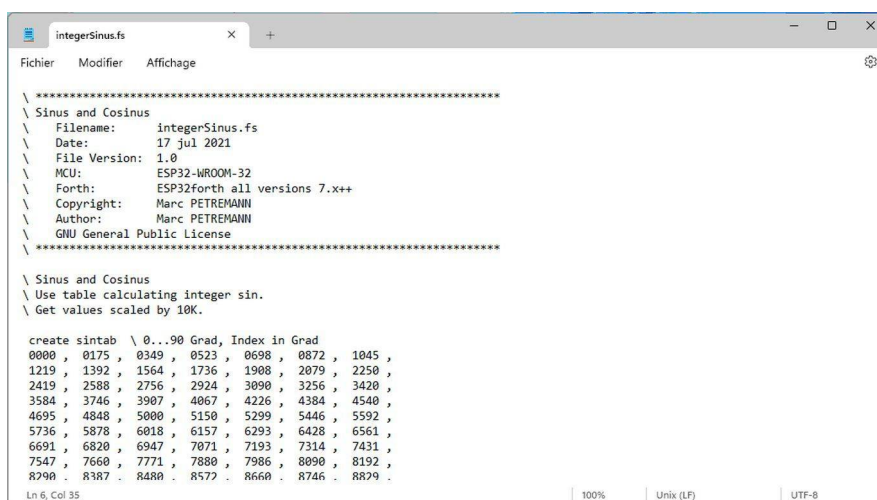
Edition et gestion des fichiers sources pour eForth Windows

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

- **txt** extension générique pour tous les fichiers texte ;
- **forth** utilisé par certains programmeurs Forth ;
- **fth** forme compressée pour Forth ;
- **4th** autre forme compressée pour Forth ;
- **fs** notre extension préférée...

Les éditeurs de fichiers texte

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



The screenshot shows a window titled 'integerSinus.fs' with a menu bar (Fichier, Modifier, Affichage) and a toolbar. The text content is as follows:

```
\ *****  
\ Sinus and Cosinus  
\ Filename:    integerSinus.fs  
\ Date:       17 jul 2021  
\ File Version: 1.0  
\ MCU:        ESP32-WROOM-32  
\ Forth:       ESP32forth all versions 7.x++  
\ Copyright:   Marc PETREMANN  
\ Author:      Marc PETREMANN  
\ GNU General Public License  
\ *****  
  
\ Sinus and Cosinus  
\ Use table calculating integer sin.  
\ Get values scaled by 10K.  
  
create sintab \ 0...90 Grad, Index in Grad  
0000 , 0175 , 0349 , 0523 , 0698 , 0872 , 1045 ,  
1219 , 1392 , 1564 , 1736 , 1908 , 2079 , 2250 ,  
2419 , 2588 , 2756 , 2924 , 3090 , 3256 , 3420 ,  
3584 , 3746 , 3907 , 4067 , 4226 , 4384 , 4540 ,  
4695 , 4848 , 5000 , 5150 , 5299 , 5446 , 5592 ,  
5736 , 5878 , 6018 , 6157 , 6293 , 6428 , 6561 ,  
6691 , 6820 , 6947 , 7071 , 7193 , 7314 , 7431 ,  
7547 , 7660 , 7771 , 7880 , 7986 , 8090 , 8192 ,  
8290 , 8387 , 8480 , 8572 , 8660 , 8746 , 8829 ,
```

At the bottom, it shows 'Ln 6, Col 35', '100%', 'Unix (LF)', and 'UTF-8'.

Figure 5: édition avec edit sous windows 11

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

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

Utiliser un IDE

Rien ne vous empêche d'utiliser un IDE¹. Pour ma part, j'ai une préférence pour **Netbeans** que j'utilise aussi pour PHP, MySQL, Javascript, C, assembleur... C'est un IDE très puissant et aussi performant qu'**Eclipse** :

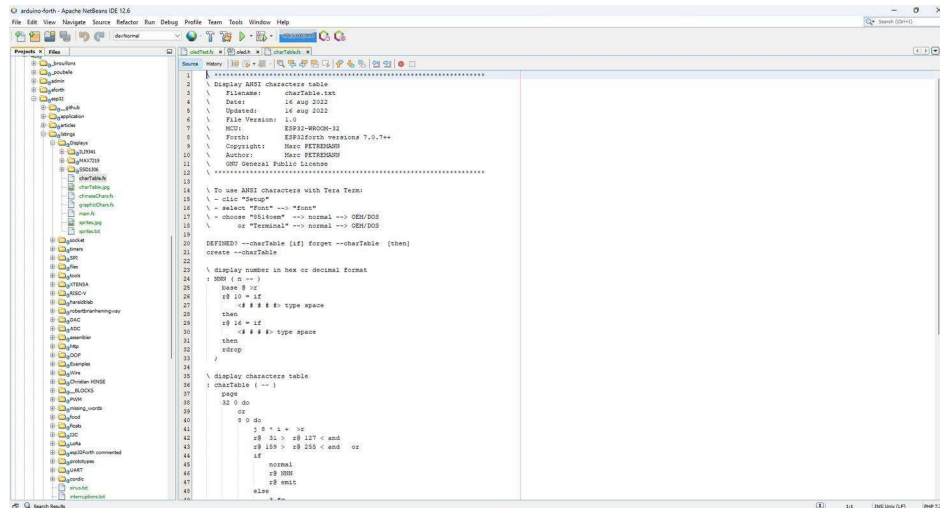


Figure 6: édition avec Netbeans

Netbeans offre plusieurs fonctionnalités intéressantes :

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

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

¹ Integrated Development Environment

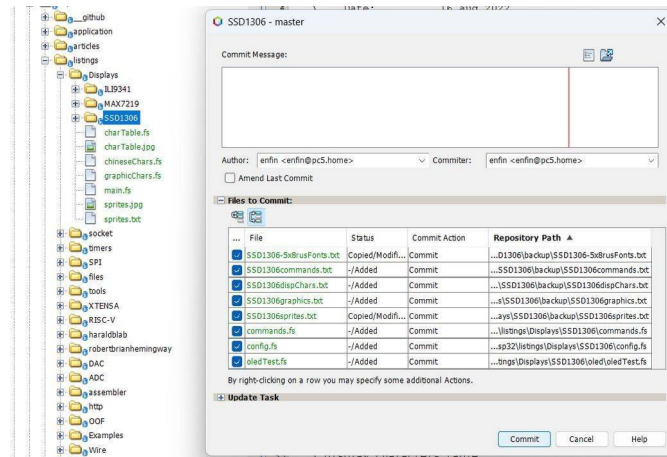


Figure 7: opération GIT dans Netbeans

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

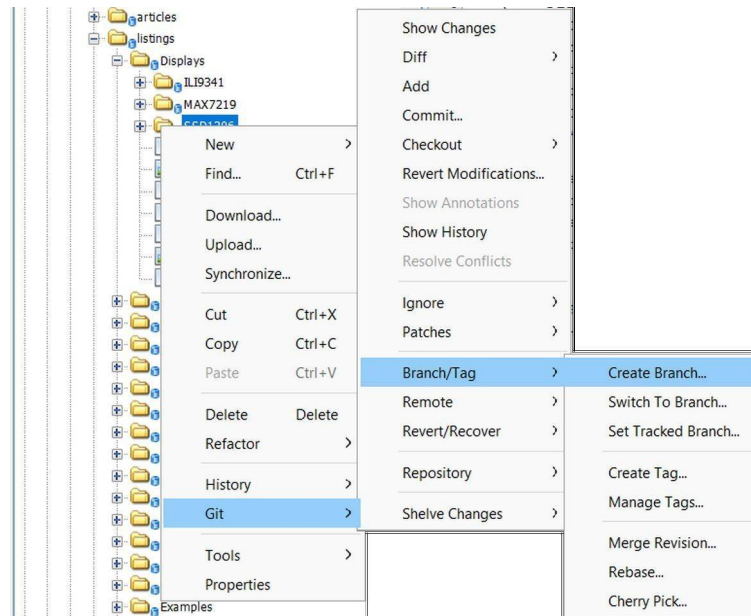


Figure 8: création d'une branche sur un projet

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

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

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

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

Stockage sur GitHub

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

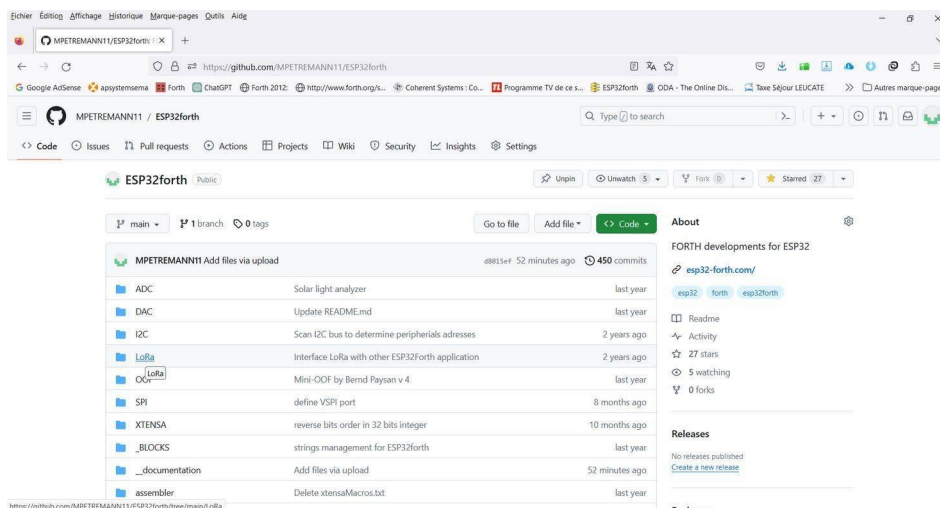


Figure 9: stockage des fichiers sur Github

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

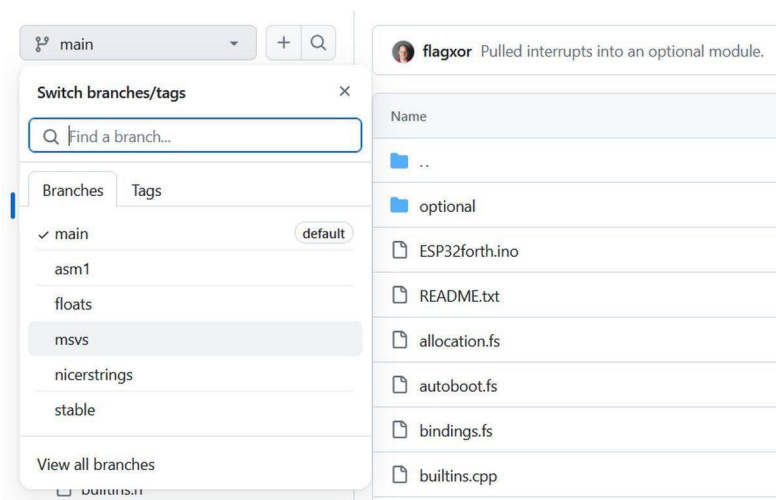


Figure 10: accès à une branche de projet

Quelques bonnes pratiques

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

² <https://github.com/>

³ <https://sourceforge.net/>

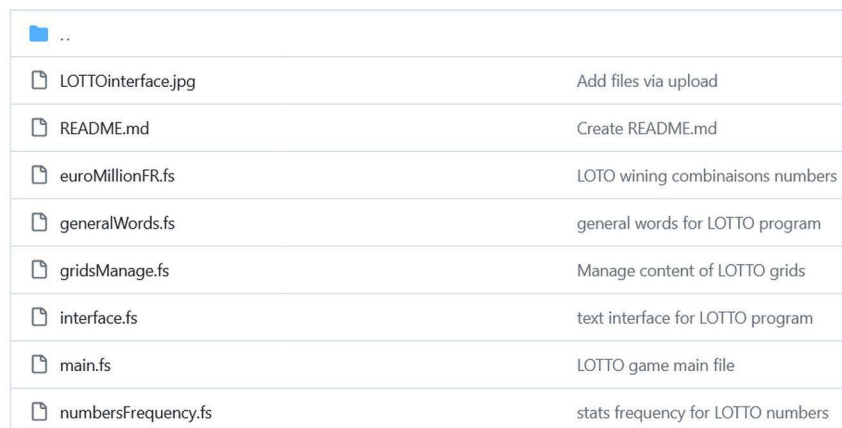
Pour les essais divers, créez dans ce dossier un sous-dossier **sandbox** (bac à sable).

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

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

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

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



..	
LOTTOinterface.jpg	Add files via upload
README.md	Create README.md
euroMillionFR.fs	LOTO wining combinaisons numbers
generalWords.fs	general words for LOTTO program
gridsManage.fs	Manage content of LOTTO grids
interface.fs	text interface for LOTTO program
main.fs	LOTTO game main file
numbersFrequency.fs	stats frequency for LOTTO numbers

Figure 11: exemple de nommage de fichiers source Forth

C'est le contenu de ces fichiers qui sera traité par eForth Windows.

Le fichier main.fs

Les fichiers windows sont stockés dans le dossier **eForth** et peuvent être lus par eForth Windows. Si vous avez écrit un fichier **config.fs**, voici la ligne de code à écrire dans **main.fs** pour accéder au contenu de **config.fs** :

```
include config.fs
```

A compter de ce moment, vous avez deux possibilités pour interpréter le contenu de **config.fs**. Depuis le terminal:

```
include config.fs
```

ou

```
include main.fs
```


L'intérêt est que **main.fs** peut appeler d'autres fichiers. Exemple :

```
\ load FLOG
include flog.fs

\ load FLOG tests
\ include tests/flogTest.fs
```

Le traitement de nombreux fichiers prend moins d'une seconde.

Exemple d'organisation d'un projet

Nous allons partir du dossier **eforth** qui contient notre interpréteur-compileur Forth **uEf64-7.0.7.21.exe** :

```
└─ eforth
   └─ uEf64 7.0.7.21.exe
```

Notre projet s'appelle lotto. On va donc ouvrir un sous-dossier **lotto** :

```
└─ eforth
   └─ uEf64 7.0.7.21.exe
      └─ lotto
```

Dans ce sous-dossier lotto, on crée notre fichier **main.fs** :

```
└─ eforth
   └─ uEf64 7.0.7.21.exe
      └─ lotto
         └─ main.fs
```

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

```
DEFINED? --loto [if] forget --loto [then]
create -loto
include generalWords.fs
include euroMillionFR.fs
include gridsManage.fs
include numbersFrequency.fs
include interface.fs
interface \ run main program
```

Ce fichier **main.fs** appelle d'autres fichiers source via **include**. Voici l'organisation des fichiers dans cet état d'avancement du projet :

```
└─ eforth
   └─ uEf64 7.0.7.21.exe
      └─ lotto
         └─ main.fs
            └─ generalWords.fs
               └─ euroMillionFR.fs
                  └─ gridsManage.fs
                     └─ numbersFrequency.fs
```

```
interface.fs
```

Et enfin, pour des raisons pratiques, on crée le fichier **LOTTO.fs** dans notre dossier **eforth** :

```
eforth
├── uEf64 7.0.7.21.exe
├── lotto
│   ├── main.fs
│   ├── generalWords.fs
│   ├── euroMillionFR.fs
│   ├── gridsManage.fs
│   ├── numbersFrequency.fs
│   └── interface.fs
└── LOTTO.fs
```

Contenu de ce fichier **LOTTO.fs** :

```
include lotto/main.fs
```

Pour tester notre projet, il suffit de lancer eForth, puis dans l'interface eForth de taper **include LOTTO.fs**.

Automatiquement, eforth va aller charger le contenu de **lotto/main.fs** et tous les fichiers appelés depuis **lotto/main.fs**.

Au fil des développements, ne doivent figurer dans le dossier **eforth** que les fichiers d'appel des projets et les sous-dossiers de ces projets.

Commentaires et mise au point

Il n'existe pas d'IDE⁴ 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 nommage 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**

4 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      \ Платат сегоднйшной 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 dictionary pointer on stack  
    0 c,      \ initial lenght data is 0  
  does>  
    dup 1+ swap c@  
    \ send a data array to SSD1306 connected via I2C bus  
    sendDatasToSSD1306  
;
```

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

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

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

Les commentaires textuels

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

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

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

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

```
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 lier 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 ( 0C --- 0F) \ 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
```

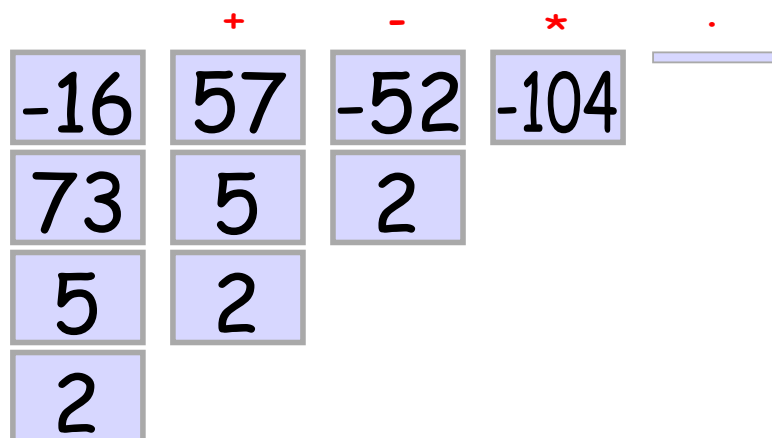
laisse la pile de paramètres dans l'état

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
```

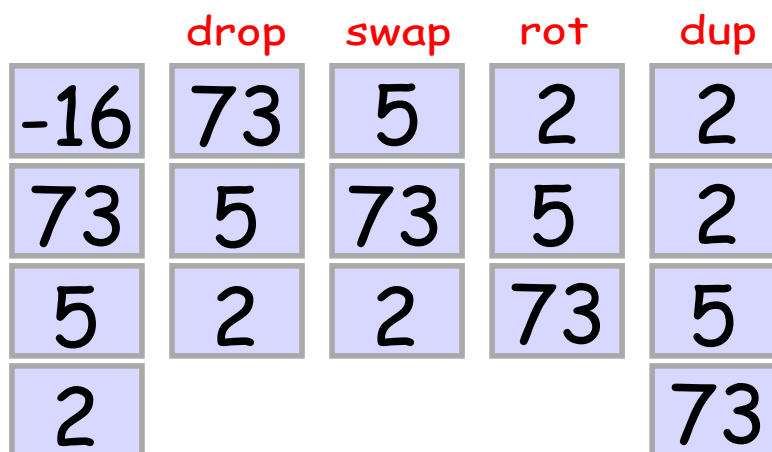
Notez que eForth Windows affiche commodément les éléments de la pile lors de l'interprétation de chaque ligne et que la valeur de -16 est affichée sous la forme d'entier non signé 32 ou 64 bits. En outre, le mot `.` consomme la valeur de données -104, laissant la pile vide. Si nous exécutons `.` sur 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
\ 0START_ADDR     \ première adresse pour la boucle dans dump
\ LINES           \ nombre de lignes pour la boucle dump
\ myBASE          \ base numérique courante
internals
: dump ( start len -- )
  cr cr ." --addr--- "
  ." 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  -----chars-----"
  2dup + { END_ADDR }          \ store latest address to dump
  swap { START_ADDR }         \ store START address to dump
  START_ADDR 16 / 16 * { 0START_ADDR } \ calc. addr for loop start
  16 / 1+ { LINES }
  base @ { myBASE }           \ save current base
  hex
  \ outer loop
  LINES 0 do
    0START_ADDR i 16 * +      \ calc start address for current line
    cr <# # # # # [char] - hold # # # # #> type
    space space              \ and display address
    \ first inner loop, display bytes
    16 0 do
      \ calculate real address
      0START_ADDR j 16 * i + +
      ca@ <# # # # #> type space \ display byte in format: NN
    loop
    space
    \ second inner loop, display chars
```

```

16 0 do
  \ calculate real address
  0START_ADDR j 16 * i + +
  \ display char if code in interval 32-127
  ca@      dup 32 < over 127 > or
  if      drop [char] . emit
  else    emit
  then
  loop
loop
myBASE base !          \ restore current base
cr cr
;
forth

```

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

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

```

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

```

Structures de données pour eForth 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 A4 ü A5 ñ A6 ã A7 º A8 ¿ A9 ® AA ã AB ¼ AC ½ AD ñ AE « AF »
B0 ß B1 ¼ B2 ½ B3 ¾ B4 ¼ B5 ½ B6 ¾ B7 ¼ B8 ½ B9 ¾ BA ¼ BB ½ BC ¾ BD ¼ BE ½ BF ¾
C0 ª C1 ¼ C2 ½ C3 ¾ C4 ¼ C5 ½ C6 ¾ C7 ¼ C8 ½ C9 ¾ CA ¼ CB ½ CC ¾ CD ¼ CE ½ CF ¾
D0 ð D1 ð D2 È D3 È D4 È D5 ù D6 Ì D7 Î D8 Ï D9 Ñ DA Ñ DB Ñ DC Ñ DD Ñ DE Ñ DF
E0 Ó E1 ß E2 Ô E3 Ò E4 õ E5 Ö E6 µ E7 þ E8 þ E9 Û EA Û EB Û EC Û ED Û EE Û EF
F0 ÷ F1 ± F2 = F3 ¼ F4 ½ F5 § F6 ÷ F7 . F8 ° F9 ° FA . FB ¹ FC ³ FD ² FE ■ FF

```

Figure 12: 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 13: 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 accesseurs **->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 14: mot d'accès mémoire en fonction de la taille du champ

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
    loop

```

```

        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

```

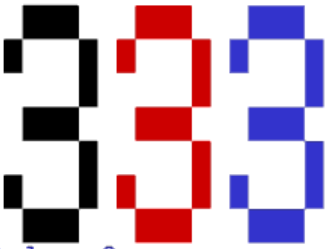


Figure 15: affichage de sprite

Résultat de l'affichage de notre sprite :

Voilà. C'est tout.

Les structures en détail

Dans le précédent chapitre, nous avons abordé les structures construites avec **struct** et **field**. On va analyser plus en détail certaines subtilités sur la manipulation des structures, en particulier sur l'accès aux données, en lecture et écriture.

Les tailles de champs

Windows exploite abondamment les structures pour échanger des données entre processus. Ici un exemple de structure partagée avec l'API Windows :

```
struct POINT
  i32 field ->x
  i32 field ->y

struct RECT
  i32 field ->left
  i32 field ->top
  i32 field ->right
  i32 field ->bottom

struct MSG
  ptr field ->hwnd
  i32 field ->message
  i16 field ->wParam
  i32 field ->lParam
  i32 field ->time
  POINT field ->pt
  i32 field ->lPrivate
```

Ce code est extrait des fichiers source de eForth Windows. Ici, sont définies trois structures : **POINT**, **RECT** et **MSG**.

Pour rappel, avant chaque **field**, on a un mot qui indique la taille du champ dans la structure:

- **i8** pour un champ de 8 bits, c'est à dire 1 octet;
- **i16** pour un champ de 16 bits, donc de 2 octets;
- **i32** pour un champ de 32 bits, donc 4 octets;
- **i64** et **ptr** pour un champ de 64 bits, donc 8 octets;

Le mot **ptr** dépend de la version FORTH. Ici, dans eForth Windows, c'est bien un champ 64 bits. Mais sur un système 32 bits, comme ESP32forth, ce sera un champ 32 bits.

Dans la structure **MSG**, on retrouve des champs de différente taille :

- **ptr field** ->**hwnd** désigne un champ 64 bits
- **i32 field** ->**message** désigne un champ 32 bits
- **i16 field** ->**wParam** désigne un champ 16 bits

Pour chaque taille de champ, il faut utiliser les mots @ ou ! adaptés à la taille du champ cible.

Accès aux données dans les champs d'une structure

On l'a compris, il faut un mot adapté à chaque taille de données :

- **8 bits** : accès avec les mots **C@** et **C!**
- **16 bits** : accès avec les mots **SW@** ou **UW@** et **W!**
- **32 bits** : accès avec les mots **SL@** ou **UL@** et **L!**
- **64 bits** : accès avec les mots **@** et **!**

Exemple :

```
create iconePos
    POINT allot
-5 iconePos ->x W!
3 iconePos ->y W!
```

On stocke la valeur -5 dans le premier champ 16 bits, la valeur 3 dans le second champ 16 bits. Pour récupérer ces valeurs, il faut procéder ainsi :

```
iconePos ->x SW@
iconePos ->y SW@
```

Pour récupérer une valeur 16 bits, il y a deux options. Utiliser **SW@** ou **UW@**. Le mot **SW@** récupère la valeur 16 bits, en tant que valeur signée.

Si on avait utilisé **UW@**, la valeur -5 aurait été remplacée par 65531 qui est son équivalent non signé sur 16 bits.

Le mécanisme est similaire pour les valeurs 32 bits avec **SL@** et **UL@**.

Gérer les accesseurs de structures

Si vous devez accéder aux données d'une structure en différents endroits de votre programme, la meilleure solution est de définir un seul mot d'accès et ensuite passer exclusivement par ce mot. Exemple :

```
: setPoint ( x y addr -- )
    >r
    r@ ->y W!
```



```

    r> ->x W!
;

: getPoint ( addr -- x y )
    >r
    r@ ->x SW@
    r> ->y SW@
;

-15 37 iconePos setPoint
iconePos getPoint \ push -15 37 on stack

```

Dans le reste du programme, l'utilisation de **setPoint** et **getPoint** rendra bien plus facile la manipulation des données d'une structure **POINT**.

Les structures imbriquées

L'API Windows exploite des structures imbriquées :

```

struct MSG
    ptr field ->hwnd
    i32 field ->message
    i16 field ->wParam
    i32 field ->lParam
    i32 field ->time
    POINT field ->pt
    i32 field ->lPrivate

```

Dans cette structure **MSG**, est défini un champ **->pt**. Ce champ a comme taille de données celle d'une structure **POINT**. Pour rappel, l'utilisation d'un nom de structure restitue simplement la taille des données de cette structure. Ici, le mot **POINT** restitue la valeur 8, valeur qui correspond à la taille des deux champs 32 bits définis dans cette structure **POINT**.

Voyons un exemple simple de structure imbriquée. Utilisons **POINT** pour définir une structure **TRIANGLE** :

```

structures
struct TRIANGLE
    POINT field ->triangle-pt1
    POINT field ->triangle-pt2
    POINT field ->triangle-pt3

: setTriangle ( x1 y1 x2 y2 x3 y3 addr -- )
    >r
    r@ ->triangle-pt3 setPoint
    r@ ->triangle-pt2 setPoint
    r> ->triangle-pt1 setPoint

```

```

;

create triAng01
    TRIANGLE ALLOT

10 10  70 10   40 70 triAng01 setTriangle

```

Dans certains cas, c'est un peu moins élégant. Voici l'accès aux données d'une structure de type **TRIANGLE** sans passer par **setPoint** :

```

: setTriangle ( x1 y1 x2 y2 x3 y3 addr -- )
    >r
    r@ ->triangle-pt3 ->y W!
    r@ ->triangle-pt3 ->x W!
    r@ ->triangle-pt2 ->y W!
    r@ ->triangle-pt2 ->x W!
    r@ ->triangle-pt1 ->y W!
    r> ->triangle-pt1 ->x W!
;

```

En résumé, les structures sont à manipuler avec soin. Réservez-les pour les interfaces avec les liaisons logicielles des API Windows.

Evolution des structures depuis la version 7.0.7.21

Dans le précédent chapitre, on a abordé le problème de la taille des champs dans une structure. Il était recommandé de définir des mots spécifiques pour lire ou écrire dans ces champs.

Dans la version eForth Windows 7.0.7.21 les mots **@field** et **!field** apparaissent dans le vocabulaire structures. Ces mots apportent une solution élégante pour lire ou écrire des données depuis des accesseurs.

Les types de champs dans les structures

Il y a quatre types de champs connus: **i8**, **i16**, **i32** et **i64**, gérant respectivement des champs de 8, 16, 32 et 64 bits. Ces quatre types sont complétés par trois nouveaux types :

- **u8** pour un champ 8 bits non signés
- **u16** pour un champ 16 bits non signés
- **u32** pour un champ 32 bits non signés

Il n'y a pas de mot **u64**.

Cette panoplie de champs est complétée du mot **sc@** qui exécute une récupération d'octet signé dans le contenu d'un champ huit bits.

```
structures
struct 8BXY
  i8 field ->8bx
  i8 field ->8by

create XY-offset
  -3 c,
  12 c,

XY-offset ->8bx sc@ . \ display -3
XY-offset ->8by sc@ . \ display 12
```

Le typage des données 8, 16, 32 ou 64 bits, signés ou non signés, rajoute de la complexité dans le choix adaptés des mots effectuant une lecture ou écriture dans les champs d'une structure.

Accès simplifié aux données d'un champ de structure

Heureusement, les deux nouveaux mots **@field** et **!field** apportent une solution qui simplifie l'accès au contenu des champs en s'adaptant automatiquement à la taille et au type des champs définis :

```
XY-offset @field ->8bx .  
XY-offset @field ->8by .
```

Les mots **@field** et **!field** doivent être suivis de l'accessor correspondant.

Utilisons **@field** dans une définition :

```
: getXY ( -- x y )  
  XY-offset @field ->8bx  
  XY-offset @field ->8by  
;  
getXY \ leave -3 12 on stack
```

Si on décompile notre mot **getXY**, on retrouve le mot **sc@** au lieu de **@field** :

```
see getXY \ display:  
: getXY  
  XY-offset ->8bx sc@ XY-offset ->8by sc@  
;
```

Les mots **@field** et **!field** sont donc particulièrement utiles pour gérer les accès à des champs de taille hétérogène dans des structures. La modification du type dans une structure ne nécessite pas la modification des définitions qui utilisent **@field** ou **!field**.

Les alias de type de données

Partons d'une structure écrite en langage C :

```
typedef struct _DCB {  
  DWORD DCBlength;  
  DWORD BaudRate;  
  DWORD fBinary : 1;  
  DWORD fParity : 1;  
  DWORD fOutxCtsFlow : 1;  
  DWORD fOutxDsrFlow : 1;  
  DWORD fDtrControl : 2;  
  DWORD fDsrSensitivity : 1;  
  DWORD fTXContinueOnXoff : 1;  
  DWORD fOutX : 1;  
  DWORD fInX : 1;  
  DWORD fErrorChar : 1;  
  DWORD fNull : 1;  
  DWORD fRtsControl : 2;  
  DWORD fAbortOnError : 1;  
  DWORD fDummy2 : 17;  
};
```

```

WORD  wReserved;
WORD  XonLim;
WORD  XoffLim;
BYTE  ByteSize;
BYTE  Parity;
BYTE  StopBits;
char  XonChar;
char  XoffChar;
char  ErrorChar;
char  EofChar;
char  EvtChar;
WORD  wReserved1;
} DCB, *LPDCB;

```

Pour utiliser cette même structure avec eForth Windows, il est nécessaire de traduire la taille de chaque type de donnée (DWORD, WORD, BYTE...) en son type de données **i8 i16 i32**. C'est long, fastidieux, et sujet à erreurs.

La solution est de définir des alias :

```

( Windows handles bottom out as void pointers. )
: HANDLE      ptr ;
: DWORD       u32 ;
: WINLONG     i32 ;
: UINT        u32 ;
: WPARAM      ptr ;
: LPARAM      ptr ;
: WINBOOL     i32 ;
: WINWORD     u16 ;
: WORD        u16 ;
: BYTE        u8  ;
\ char use u8 or BYTE

```

Il n'y a que pour le type *char* pour lequel on ne définit pas d'alias, car il y a risque de collision avec le mot char du vocabulaire FORTH. Voici comment on peut réécrire en Forth la structure **DCB** si on utilise ces alias :

```

structures
struct DCB
  DWORD field ->DCBlength
  DWORD field ->BaudRate
  DWORD field ->fBinary
  DWORD field ->fParity
  DWORD field ->fOutxCtsFlow
  DWORD field ->fOutxDsrFlow
  DWORD field ->fDtrControl
  DWORD field ->fDsrSensitivity
  DWORD field ->fTXContinueOnXoff
  DWORD field ->fOutX

```

```

DWORD field ->fInX
DWORD field ->fErrorChar
DWORD field ->fNull
DWORD field ->fRtsControl
DWORD field ->fAbortOnError
DWORD field ->fDummy2
WORD field ->wReserved
WORD field ->XonLim
WORD field ->XoffLim
BYTE field ->ByteSize
BYTE field ->Parity
BYTE field ->StopBits
BYTE field ->XonChar
BYTE field ->XoffChar
BYTE field ->ErrorChar
BYTE field ->EofChar
BYTE field ->EvtChar
WORD field ->wReserved1

```

Dans la version de **DCB** en langage C, nous avons un certain nombre de champs avec des valeurs par défaut. Voici comment initialiser une structure **DCB** en Forth avec **!field** :

```

: DCB.init ( DCBaddr -- )
  >r
  1  r@ !field ->fBinary
  1  r@ !field ->fParity
  1  r@ !field ->fOutxCtsFlow
  1  r@ !field ->fOutxDsrFlow
  2  r@ !field ->fDtrControl
  1  r@ !field ->fDsrSensitivity
  1  r@ !field ->fTXContinueOnXoff
  1  r@ !field ->fOutX
  1  r@ !field ->fInX
  1  r@ !field ->fErrorChar
  1  r@ !field ->fNull
  2  r@ !field ->fRtsControl
  1  r@ !field ->fAbortOnError
  17 r> !field ->fDummy2
;
create COM5-DCB
  DCB allot
  COM5-DCB DCB.init

```

En résumé, avec cette évolution, la gestion des champs dans les structures devient infiniment plus simple et beaucoup plus souple. Cette évolution reste compatible avec les codes sources Forth plus anciens exploitant les structures sans nécessiter la réécriture de ces fichiers source.

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
  LOOP

```

```

4 #out+!
I EMIT 2 SPACES      \ affiche caractère
3 #out+!
#out @ 77 =
IF
    CR    0 #out !
THEN
LOOP ;

```

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

```

hex jeu-ascii
20      21 !    22 "    23 #    24 $    25 %    26 &    27 '    28 (    29 )    2A *
2B +    2C ,    2D -    2E .    2F /    30 0    31 1    32 2    33 3    34 4    35 5
36 6    37 7    38 8    39 9    3A :    3B ;    3C <    3D =    3E >    3F ?    40 @
41 A    42 B    43 C    44 D    45 E    46 F    47 G    48 H    49 I    4A J    4B K
4C L    4D M    4E N    4F O    50 P    51 Q    52 R    53 S    54 T    55 U    56 V
57 W    58 X    59 Y    5A Z    5B [    5C \    5D ]    5E ^    5F _    60 `    61 a
62 b    63 c    64 d    65 e    66 f    67 g    68 h    69 i    6A j    6B k    6C l
6D m    6E n    6F o    70 p    71 q    72 r    73 s    74 t    75 u    76 v    77 w
78 x    79 y    7A z    7B {    7C |    7D }    7E ~    7F   ok

```

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

```

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

```

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

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

```

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

```

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

```

LIGNE1 TYPE      \ affiche E..Enregistrement de données

```

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

```

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

```

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

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

Variables chaînes de caractères

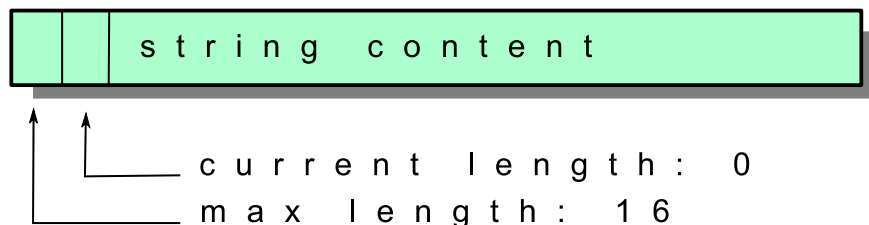
Les variables alpha-numérique texte n'existent pas nativement dans eForth Windows. Voici le premier essai de définition du mot **string** :

```
\ 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 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
;
```

Comparaisons et branchements

eForth Windows vous permet de comparer deux nombres sur la pile, en utilisant les opérateurs relationnels **>**, **<** et **=** :

```
2 3 = .      \ display: 0
2 3 >        \ display: 0
2 3 <        \ display: -1
```

Ces opérateurs consomment les deux paramètres et laissent un indicateur, pour représenter le résultat booléen. Ci-dessus, on voit que «2 est égal à 3» est faux (valeur 0), «2 est supérieur à 3» est également faux, tandis que "2 est inférieur à 3" est vrai. Le drapeau vrai a tous les bits mis à 1, d'où la représentation **-1**. eForth Windows fournit également les opérateurs relationnels **0=** et **0<** qui testent si le contenu du sommet de la pile est nul ou négatif.

Les mots relationnels sont utilisés pour le contrôle. Exemple :

```
: test
  0= invert
  if
    cr ." Not zero!"
  then
;
0 test      \ no display
-14 test    \ display: Not zero!
```

Le sommet de pile est comparé à zéro. Si le sommet de pile est différent de zéro, le mot **if** consomme le drapeau et s'exécute tous les mots entre lui-même et la terminaison **then**. Si le sommet de pile est nul, l'exécution saute au mot qui suit **then**. Le mot **cr** émet un retour chariot (newline). Le mot **else** peut être utilisé pour fournir un autre chemin d'exécution, comme illustré ici :

```
: truth
  0=
  if
    ." false"
  else
    ." true"
  then
;
1 truth      \ display: true
0 truth      \ display: false
```

Un sommet de pile non nul entraîne l'exécution de mots entre **if** et **else**, et les mots entre **else** et **then** sont ignorés. Une valeur nulle produit le comportement inverse.

Le langage Forth ne gère aucun autre type de données que des entiers traités par la pile de paramètres.

Les entiers de la version eForth Windows sont au format 64 bits. Il n'existe donc pas de type booléen en Forth, à fortiori avec eForth Windows.

Pour rompre le déroulement linéaire d'une séquence d'instructions, on utilisera des structures de contrôle dont le rôle est de compiler des branchements. Ces branchements sont de deux sortes :

- les branchements conditionnels
- les branchements inconditionnels

Une suite d'instructions sera répétée à l'aide d'une boucle. Celles-ci sont également de deux sortes:

- boucles itératives ou répétitives contrôlées par des index de début et de fin de boucle
- boucles indéfinies

Le langage Forth exploite à travers différentes structures de base, toutes ces possibilités de branchement et de boucles.

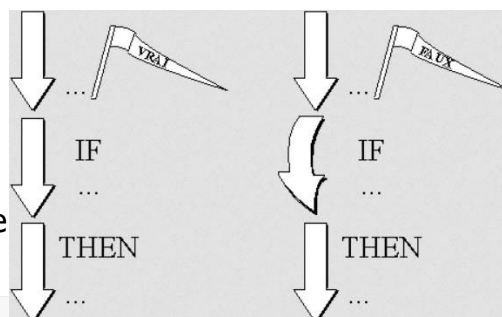
Branchements conditionnels vers l'avant

Au cours du déroulement d'un programme, le résultat d'un test ou le contenu d'une variable peut être amené à modifier l'ordre d'exécution des instructions. Cette valeur, nommée flag booléen, est à l'état vrai si elle est non nulle, à l'état faux si elle est nulle. Le flag booléen est utilisé par une instruction de branchement marquant le début d'une structure de contrôle :

- **IF ... THEN**
- **IF ... ELSE ... THEN**

Le mot **if** est un mot d'exécution immédiate et compile un branchement conditionnel. Il n'est utilisable qu'en compilation. Exemple:

```
variable chaleur
: meteo ( -- )
  chaleur @ 25 > if
    cr ." Hot weather"
  then ;
```



et en exécution :

```
15 CHALEUR ! METEO \ n'affiche rien
35 CHALEUR ! METEO \ affiche " hot weather"
```

Le mot **METEO** teste le contenu de la variable **CHALEUR** et exécute la partie de définition comprise entre **IF** et **THEN** si cette valeur est supérieure à 25. Mais s'il fait 25 degrés ou moins, le mot **METEO** n'annonce rien.

Pour y remédier, on peut le redéfinir :

```
: meteo ( -- )
  chaleur @ 25 > if
    cr ." Hot weather"
  else
    cr ." Cold weather"
  then ;
```

Maintenant :

```
15 CHALEUR ! METEO2 \ affiche cold weather
```

Si le résultat du test exécuté avant **IF** délivre un flag booléen faux, c'est la partie de définition située entre **ELSE** et **THEN** qui sera exécutée.

Un flag booléen peut être le résultat de diverses opérations:

- lecture d'une adresse mémoire,
- empilage du contenu d'un registre en sortie d'exécution d'une définition écrite en code machine,
- résultat d'un calcul arithmétique 64 bits, signé ou non,
- résultat d'une opération logique (**OR**, **AND**...) ou d'une combinaison d'opérations logiques,
- paramètre empilé par l'exécution d'une définition exécutée avant **IF**,

Le mot **IF** teste et consomme la valeur située au sommet de la pile de données. Voici un autre exemple affichant si un nombre est pair ou non :

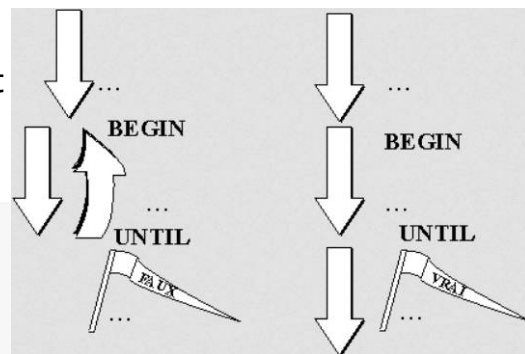
```
: is-even? ( n -- )
  dup 2 mod
  if      ." is even"
  else    ." is odd"
  then ;
```

Branchement conditionnel vers l'arrière

Avec la structure de type **IF.. THEN** et **IF.. ELSE.. THEN**, on ne peut exécuter qu'une partie de définition non répétitive. Pour réitérer une séquence de type faire.. tant-que, il faut exploiter un nouveau type de structure de contrôle: la boucle répétitive indéfinie **BEGIN.. UNTIL**.

Dans une boucle **BEGIN.. UNTIL**, la partie de définition située entre ces deux mots est répétée tant que le résultat du test précédant **UNTIL** délivre un flag booléen faux. Exemple :

```
: dactylo ( -- )
  begin
    key dup emit
    [char] $ =
  until ;
```



L'exécution de **dactylo** affiche tous les caractères tapés au clavier. Seul l'appui sur la touche marquée du signe '\$' peut interrompre la répétition. L'appui sur la touche <return> renvoie en début de ligne.

Si le résultat du test précédant **UNTIL** est toujours faux, on ne peut plus sortir de la boucle **BEGIN.. UNTIL**. Cette situation a été prévue en Forth et est exploitée par la boucle **BEGIN.. AGAIN**. L'utilisation de **AGAIN** équivaut à **BEGIN ... 0 UNTIL**. Si on définit **dactylo** par :

```
: dactylo ( -- )
  begin key emit again ;
```

On ne pourra plus interrompre la boucle lors de son exécution. Le seul moyen pour s'en sortir sera de fermer la fenêtre eForth Windows.

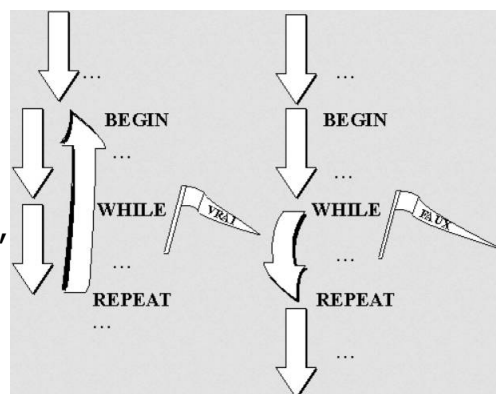
Branchement en avant depuis une boucle indéfinie

Avec une boucle **BEGIN.. UNTIL** ou **BEGIN.. AGAIN**, l'action est répétée en fin de boucle en fonction du résultat d'un test ou de manière inconditionnelle. Mais on peut exploiter un branchement avant conditionnel depuis une boucle dont la structure est **BEGIN.. WHILE.. REPEAT**.

Dans cette structure, le test est exécuté avant **WHILE**. Si le résultat est faux, l'exécution se poursuit après **REPEAT**. Si le résultat est vrai, la partie de définition comprise entre **WHILE** et **REPEAT** est exécutée, puis **REPEAT** effectue un branchement arrière inconditionnel, c'est à dire renvoie l'exécution à **BEGIN**.

Pour exemple, réécrivons **dactylo** avec cette nouvelle structure :

```
: dactylo ( -- )
  begin key dup [char] $ <>
  while emit
  repeat
  drop ;
```



Répétition contrôlée d'une action

Le dernier cas de figure des diverses structures de contrôle disponibles en Forth est la boucle **DO.. LOOP**. Cette structure admet comme paramètres d'entrée deux valeurs qui sont les index initiaux et terminaux contrôlant l'itération.

Exemple :

```
: myLoop ( -- )
  10000 0 do loop ;
```

Le mot **DO**, utilisable seulement en compilation, est toujours précédé de deux valeurs n1 n2, où n2 est l'index initial de la boucle, n1 la valeur terminale. La valeur terminale est en général supérieure à la valeur initiale. Exemple :

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

affiche tous les caractères ASCII situés entre 32 et 127. Dans cette définition, le mot **I** dépose sur la pile la valeur de l'index courant de la boucle. L'index prend successivement toutes les valeurs comprises entre 32 et 128, 128 non inclus. L'incrément et le test de sortie de boucle sont exécutés par **LOOP**.

Pour incrémenter l'index de boucle d'une quantité différente de une unité, il faut remplacer le mot **LOOP** par **+LOOP** et le faire précéder de la valeur de l'incrément de boucle. Exemple :

```
variable table
: .table
  cr 11 table @ * table @
  do i . table @ +loop ;

3 table ! .table
\ affiche 3 6 9 12 15 ... 30

7 table ! .table
\ affiche 7 14 21 28 .... 70
```

L'incrément d'index de boucle située avant **+LOOP** peut être négatif. Dans ce cas, la première valeur placée avant **DO** est inférieure à la seconde. Rien ne nous interdit de ré-utiliser **.table** dans une définition plus générale qui va nous afficher une véritable table de multiplication :

```
: xTable ( -- )
  11 1 do
    i table ! .table
  loop ;
```

Ne soyez pas étonné par la réutilisation du mot **I**, déjà exploité dans la définition de **.table**. L'exécution de **I** ne fait référence qu'à la boucle dans laquelle il est défini et en cours d'exécution. Par contre, dans le cas de deux boucles imbriquées, pour accéder

depuis la boucle imbriquée à l'index de la boucle extérieure, il faut utiliser le mot **J**.

Exemple:

```
: twoLoops ( -- )
  6 0 do
    cr i .
    10 0 do
      i j * .
    loop
  loop
  cr ;
```

Pour interrompre le déroulement d'une boucle de type **DO.. LOOP** ou **DO.. +LOOP**, il faut exécuter le mot **LEAVE**.

Structure uni-conditionnelle à choix multiples

Dans certaines situations, on peut être amené à exécuter une action spécifique au programme en fonction d'une condition précise, parmi d'autres actions également dépendantes de cette condition. La structure de contrôle la plus fréquemment utilisée dans ce cas est la structure de type **CASE.. OF.. ENDOF.. ENDCASE**. Syntaxe:

```
CASE
  valeur1 OF action1 ENDOF
  valeur2 OF action2 ENDOF
  ...
  valeurN OF actionN ENDOF
ENDCASE
```

La valeur à tester est placée devant le mot **OF**. Si la valeur figurant au sommet de la pile avant l'exécution de **CASE** est identique à celle-ci, la partie de définition située entre **OF** et **ENDOF** est exécutée, puis l'exécution se poursuit après **ENDCASE**. Dans le cas contraire, le test ou les tests suivants sont exécutés jusqu'à validité d'un test. Si aucun test n'a pu être vérifié, la partie de définition située le cas échéant entre le dernier **ENDOF** et **ENDCASE** est exécutée. Exemple :

```
: JOUR ( n - addr len )
  7 mod
  case
    0 of s" DIMANCHE"   endof
    1 of s" LUNDI"      endof
    2 of s" MARDI"      endof
    3 of s" MERCREDI"   endof
    4 of s" JEUDI"      endof
    5 of s" VENDREDI"   endof
    6 of s" SAMEDI"     endof
  endcase
;
```

Exemples d'exécution :

```
2 JOUR TYPE    \ affiche MARDI
5 JOUR TYPE    \ affiche VENDREDI
7 JOUR TYPE    \ affiche DIMANCHE
```

La récursivité

Les puristes affirment qu'un langage informatique est incomplet s'il ne peut traiter la récursivité. D'autres prétendent que l'on peut très bien s'en passer. Forth va enthousiasmer les premiers, car il est équipé pour traiter ce type de situation.

Comme Forth ne peut faire explicitement référence au mot en cours de définition, on insérera le mot **RECURSE** partout où l'on voudra faire appel à la définition en cours de compilation. Exemple :

```
: factor ( n - FACTn )
  dup 1 - dup 1 >
  if
    recurse
  then
  * ;
```

La décompilation de **factor**, exécutée en tapant **see factor**, met en évidence la substitution de **recurse** par l'adresse d'exécution de **factor**. Exemples d'exécution de **factor** :

```
2 factor .    \ affiche 2
3 factor .    \ affiche 6
4 factor .    \ affiche 24
```

La récursivité a ses limites : la capacité des piles de données et de retour. Une récursivité mal contrôlée sature la pile de données ou de retour et bloque le système.

Les tests logiques

Les tests peuvent porter sur des comparaisons de grandeurs numériques 64 bits exclusivement. Le résultat d'un test est toujours un flag booléen, représenté par un entier 64 bits :

- 0 pour **faux**
- -1 ou toute autre valeur non nulle pour **vrai**

Un test de comparaison agit de manière identique à un opérateur arithmétique. Ce sont des opérateurs dyadiques :

```
= < > <= >= <>
U< U> U>= U<=
```

Exemples :

```
2 5 = . \ affiche 0
3 8 < . \ affiche -1
```

Les tests de comparaison par rapport à zéro sont déjà définis. Ce sont des opérateurs monadiques :

```
0= 0< 0> 0<> 0<= 0>=
```

Exemple:

```
12 0> . \ affiche -1
```

Les résultats de ces tests peuvent être combinés en une seule expression à l'aide des opérateurs logiques suivants :

- **OR AND XOR** combinent deux valeurs logiques
- **INVERT** inverse l'état d'une valeur logique

Les opérateurs logiques agissent bit à bit sur deux valeurs 64 bits non signés.

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 eForth Windows, 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 eForth Windows:

- **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 ;

- **windows** gestion de l'environnement Windows

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 abrègerons 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 !

```

Gestion des paramètres entre eForth et l'API Windows

La gestion des paramètres entre eForth et l'API Windows est un point sensible. Beaucoup de fonctions du langage C exigent un typage des données: booléen, chaînes de caractères, valeurs en 8, 16, 32 ou 64 bits. Nous allons explorer en détail cette gestion des paramètres.

Passage des paramètres par la pile de données

On le rappelle, la version eForth Windows 7.0.7.21 et suivantes gèrent toutes les données sur les piles de données et de retour au format entiers 64 bits.

eForth Windows peut être étendu en créant de nouveaux mots FORTH interfacés à des fonctions de l'API Windows. Exemple :

```
only forth
windows definitions

\ returns the system's time and date
z" GetLocalTime"      1 Kernel32 GetLocalTime ( addr -- )
```

Ici, on définit un nouveau mot **GetLocalTime** qui utilise la fonction **GetLocalTime** disponible dans le fichier **kernel32.dll**.

Au fil des versions de Windows, les fonctionnalités de **kernel32.dll** ont évolué pour s'adapter aux architectures 64 bits, offrant ainsi une meilleure gestion des grands volumes de données.

La capacité d'une fonction à gérer des données 64 bits dépend en grande partie du type de données qu'elle manipule. Certaines fonctions sont spécifiquement conçues pour travailler avec des entiers 64 bits (int64_t), des pointeurs 64 bits, etc.

On définit la structure associée à **GetLocalTime** :

```
\ Definition in C language:
\ typedef struct _SYSTEMTIME {
\   WORD wYear;
\   WORD wMonth;
\   WORD wDayOfWeek;
\   WORD wDay;
\   WORD wHour;
\   WORD wMinute;
\   WORD wSecond;
```

```

\   WORD wMilliseconds;
\ } SYSTEMTIME, *PSYSTEMTIME, *LPSYSTEMTIME;

struct SYSTEMTIME
  i16 field ->wYear
  i16 field ->wMonth
  i16 field ->wDayOfWeek
  i16 field ->wDay
  i16 field ->wHour
  i16 field ->wMinute
  i16 field ->wSecond
  i16 field ->wMilliseconds

```

Ici on définit la structure **SYSTEMTIME** calquée sur sa définition en langage C, dont le code source a été mis en commentaire. Ce code source en C ne sert que pour information. Une fois la structure testée avec succès, on peut supprimer ce code en C.

Exemple d'utilisation :

```

create sysTime
  SYSTEMTIME allot

: dispTime ( -- )
  sysTime GetLocalTime drop
  systime ->wHour    UW@ .
  systime ->wMinute  UW@ .
  systime ->wSecond  UW@ .
;

```

Ici, le mot **dispTime** va afficher l'heure extraite du système. Dans la définition de **dispTime**, on accède aux champs de la structure **sysTime** avec des accesseurs pointant vers des données au format 16 bits non signées.

L'adresse de la structure **sysTime** est passée au mot **GetLocalTime** au format des données de la pile de données, ici au format 64 bits.

En retour, **GetLocalTime** empile un flag booléen qui sera toujours une donnée 64 bits dans la pile de données Forth.

Donc, sur un système Windows 64 bits, avec eForth Windows 7.0.7.21+, tous les échanges de données par la pile de données seront systématiquement au format 64 bits.

La seule chose qui peut être nécessaire, c'est de limiter l'intervalle de définition d'une donnée. Ici, le code source, en langage C de la fonction **CreateFontA** :

```

HFONT CreateFontA(
  [in] int    cHeight,
  [in] int    cWidth,
  [in] int    cEscapement,
  [in] int    cOrientation,

```

```
[in] int      cWeight,
[in] DWORD    bItalic,
[in] DWORD    bUnderline,
[in] DWORD    bStrikeOut,
[in] DWORD    iCharSet,
[in] DWORD    iOutPrecision,
[in] DWORD    iClipPrecision,
[in] DWORD    iQuality,
[in] DWORD    iPitchAndFamily,
[in] LPCSTR    pszFaceName
);
```

Ici, le paramètre **bItalic** doit être un flag booléen, donc une valeur dans l'intervale [0,1]. En C, la donnée est typée **DWORD**. Un **DWORD** est un raccourci pour "Double Word". Il représente généralement un entier non signé de 32 bits, soit 4 octets.

eForth empile une valeur 64 bits pour ce paramètre **bItalic**, la fonction en C dans l'API va tronquer cette valeur 64 bits au moment d'injecter cette valeur dans la fonction **CreateFontA**.

Taille des données dans les structures

L'API Windows réalise beaucoup d'échange de données en utilisant des structures. Ces structures existent sous Windows et Linux. Pour ce qui nous concerne, on ne s'intéressera qu'aux structures à gérer entre eForth et Windows.

Commençons par une structure très simple, la structure **RECT** :

```
\ typedef struct tagRECT {
\   LONG left;
\   LONG top;
\   LONG right;
\   LONG bottom;
\ } RECT, *PRECT, *NPRECT, *LPRECT;

struct RECT
  i32 field ->left
  i32 field ->top
  i32 field ->right
  i32 field ->bottom
```

Dans cette structure **RECT**, nous n'avons à gérer que quatre champs de type LONG, donc on utilise **i32** pour définir chaque champ de cette structure.

Ici, une structure aux données avec des formats variés :

```
\ typedef struct tagBITMAPINFOHEADER {
\   DWORD biSize;
\   LONG  biWidth;
```



```

\ LONG biHeight;
\ WORD biPlanes;
\ WORD biBitCount;
\ DWORD biCompression;
\ DWORD biSizeImage;
\ LONG biXPelsPerMeter;
\ LONG biYPelsPerMeter;
\ DWORD biClrUsed;
\ DWORD biClrImportant;
\ } BITMAPINFOHEADER

struct BITMAPINFOHEADER
  i32 field ->biSize \ must be i32 ??
  i32 field ->biWidth
  i32 field ->biHeight
  i16 field ->biPlanes
  i16 field ->biBitCount
  i32 field ->biCompression
  i32 field ->biSizeImage
  i32 field ->biXPelsPerMeter
  i32 field ->biYPelsPerMeter
  i32 field ->biClrUsed
  i32 field ->biClrImportant

```

Pour chaque type de données, on utilise le paramètre **i8**, **i16** ou **i32** correspondant à la taille de la donnée à gérer :

C	eForth	taille
<i>pointer</i>	ptr	8 octet
BYTE	i8	1 octet
DWORD	i32	4 octets
LONG	i32	4 octets
UINT	i32	4 octets
WORD	i16	2 octets

Dans certains cas, l'analyse peut s'avérer complexe :

```

\ typedef struct tagMSG {
\   HWND  hwnd;
\   UINT  message;
\   WPARAM wParam;
\   LPARAM lParam;
\   DWORD time;
\   POINT pt;
\   DWORD lPrivate;
\ } MSG, *PMSG, *NPMSG, *LPMSG;

```

```
struct MSG
    ptr field ->hwnd
    i32 field ->message
    i16 field ->wParam
    i32 field ->lParam
    i32 field ->time
    POINT field ->pt
    i32 field ->lPrivate
```

Ici, l'analyse des types **WPARAM** et **LPARAM** doit se faire par analyse du code source en C dans lequel est définie la structure.

Le type **POINT** est une structure. ici, on a affaire à une structure imbriquée.

De nombreuses clés sont expliquées ici:

<https://learn.microsoft.com/fr-fr/windows/win32/winprog/windows-data-types>

Affichage de boîtes modales

Les boîtes modales font partie intégrante de l'environnement Windows. Elles sont très utiles pour afficher une information, émettre un avertissement. A la fermeture de la boîte modale, on peut même récupérer certains choix provoqués par la fermeture de boîte modale.

Une boîte modale, c'est un peu comme un panneau d'arrêt qui surgit sur l'écran de l'ordinateur.

C'est une petite fenêtre apparaissant, recouvrant tout ou partie de la page. Cette fenêtre, c'est une boîte modale:

- pour attirer l'attention. Elle signale quelque chose d'important, comme une confirmation à donner, un choix à faire ou une information à lire;
- pour demander une action. Elle peut demander de remplir un formulaire, de cliquer sur un bouton ou de prendre une décision avant de pouvoir continuer;
- pour informer. Elle peut afficher un message d'erreur, une confirmation de réussite ou simplement donner une information supplémentaire.

C'est le mot "modale" qui est important ici. Il signifie qu'il faut agir avec cette boîte avant de pouvoir continuer à utiliser le reste de l'application. C'est comme répondre à une question avant de pouvoir passer à la suite.

Exemple concret: quand on veut supprimer un fichier sur l'ordinateur, une boîte modale apparaît souvent pour demander de confirmer la décision. Il faut cliquer sur "Oui" ou "Non" avant que le fichier soit effectivement supprimé.

MessageBoxA

Le mot **MessageBoxA** est défini dans le vocabulaire **windows**.

Paramètres:

- **hWnd** Handle pour la fenêtre propriétaire de la boîte de message à créer. Si ce paramètre a la valeur **NULL**, la zone de message n'a pas de fenêtre propriétaire;
- **lpText** Message à afficher. Si la chaîne se compose de plusieurs lignes, vous pouvez séparer les lignes à l'aide d'un retour chariot et/ou d'un caractère de saut de ligne entre chaque ligne.
- **lpCaption** Titre de la boîte de dialogue. Si ce paramètre a la valeur **NULL**, le titre par défaut est *Erreur*.

- **uType** Contenu et comportement de la boîte de dialogue. Ce paramètre peut être une combinaison d'indicateurs des groupes d'indicateurs qu'on va voir plus en détail après.

Contenu et comportement de la boîte de dialogue

Exemple de boite modale :

```
only
windows also
graphics internals

: MSGbox
  NULL z" Beware, message test!" NULL MB_OK MessageBoxA
  ?dup if
    cr ." You have pressed: " .
  then ;

: run
  MSGbox
;
```

L'exécution de **run** affiche ceci:

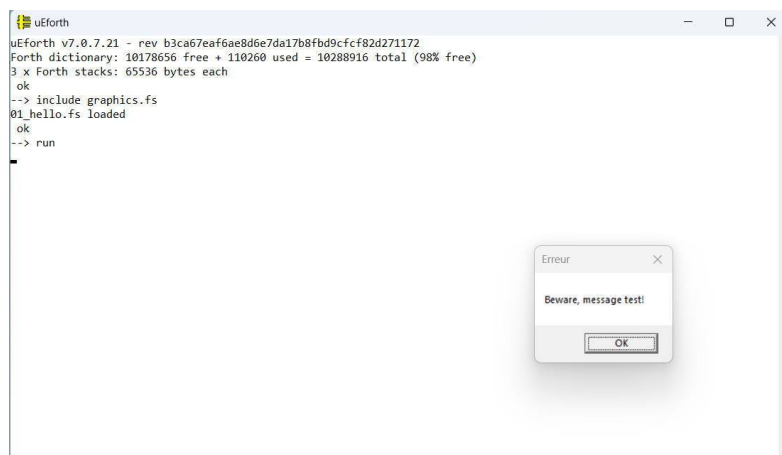


Figure 16: affichage de notre première fenêtre modale

Dans le vocabulaire **windows**, les constantes préfixées MB sont associées à **MessageBoxA**:

```
0 constant MB_OK
1 constant MB_OKCANCEL
2 constant MB_ABORTRETRYIGNORE
3 constant MB_YESNOCANCEL
4 constant MB_YESNO
5 constant MB_RETRYCANCEL
6 constant MB_CANCELTRYCONTINUE
```

Elles correspondent au paramètre **uType** et sélectionnent les boutons affichés dans la boite modale :

- **MB_OK** La boîte de message contient un bouton d'envoi : OK. Il s'agit de la valeur par défaut.
- **MB_OKCANCEL** La boîte de message contient deux boutons pousseurs : OK et Annuler.
- **MB_ABORTRETRYIGNORE** La boîte de message contient trois boutons d'envoi : Abandonner, Réessayer et Ignorer.
- **MB_YESNOCANCEL** La boîte de message contient trois boutons pousseurs : Oui, Non et Annuler.
- **MB_YESNO** La boîte de message contient deux boutons pousseurs : Oui et Non.
- **MB_RETRYCANCEL** La boîte de message contient deux boutons d'envoi : Réessayer et Annuler.
- **MB_CANCELTRYCONTINUE** La boîte de message contient trois boutons d'envoi : Annuler, Réessayer, Continuer. Utilisez ce type de boîte de message au lieu de **MB_ABORTRETRYIGNORE**.

Valeur retournée

Si une zone de message comporte un bouton *Annuler*, le mot retourne la valeur 2 si vous appuyez sur la touche Échap ou si le bouton *Annuler* est sélectionné.

Si la boîte de message n'a pas de bouton *Annuler*, appuyer sur Échap n'aura aucun effet, sauf si un bouton **MB_OK** est présent. Si un bouton **MB_OK** s'affiche et que l'utilisateur appuie sur Échap, la valeur de retour est 1.

Si la fonction échoue, la valeur de retour est égale à zéro. Pour obtenir des informations détaillées sur l'erreur, utiliser **GetLastError**.

Si la fonction réussit, la valeur de retour est l'une des valeurs d'élément de menu suivantes :

- **3** Le bouton Abandonner a été sélectionné.
- **2** Le bouton Annuler a été sélectionné.
- **11** Le bouton Continuer a été sélectionné.
- **5** Le bouton Ignorer a été sélectionné.
- **7** Le bouton Non a été sélectionné.
- **1** Le bouton OK a été sélectionné.
- **4** Le bouton Réessayer a été sélectionné.
- **10** Le bouton Réessayer a été sélectionné.

- **6** Le bouton Oui a été sélectionné.

Voyons ceci avec un exemple pratique.

Explications supplémentaires

Voici quelques informations sur le comportement de **MessageBoxA**.

```
: lpText r" You must make a choice:
  Yes to continue
  No to stop" s>z ;
z" make a choice"      constant lpCaption

: MSGbox ( -- )
  NULL lpText lpCaption MB_YESNO MessageBoxA
  ?dup if
    cr ." You have pressed: "
    case
      1 of ." OK"      endof
      6 of ." Yes"     endof
      7 of ." No"      endof
    endcase
  else
    cr ." You canceled the operation."
  then
  ;
```

Ici, le mot **MessageBoxA** suspend l'exécution de **MSGbox**. La boîte modale agit comme **key** dans une application Forth non graphique.



Figure 17: test de boîte modale OUI / NON

Le mot **lpCaption** définit le message qui sert de titre à la boîte modale.

Le mot **lpText** contient le message à afficher dans la boîte modale.

Définition du bouton par défaut

Vous pouvez désigner un bouton par défaut. Voici les constantes nécessaires pour sélectionner un bouton :

```
$00000000 constant MB_DEFBUTTON1
$00000100 constant MB_DEFBUTTON2
$00000200 constant MB_DEFBUTTON3
$00000300 constant MB_DEFBUTTON4
```

Si vous avez une boîte modale avec un choix entre deux boutons, vous ne pouvez utiliser que **MB_DEFBUTTON1** ou **MB_DEFBUTTON2**.

Exemple:

```
: MSGbox ( -- )
  NULL lpText lpCaption
  MB_YESNO MB_DEFBUTTON2 or
  MessageBoxA
  ?dup if
    cr ." You have pressed: "
    case
      1 of ." OK"      endof
      6 of ." Yes"     endof
      7 of ." No"      endof
    endcase
  else
    cr ." You canceled the operation."
  then
  ;
```

Ici, la séquence **MB_YESNO MB_DEFBUTTON2** or sélectionne le second bouton par défaut.

Rajout d'une icône dans la boîte modale

Vous pouvez rajouter une icône qui accroît l'importance du contenu de la boîte modale, parmi ces options :

```
$00000010 constant MB_ICONHAND
$00000020 constant MB_ICONQUESTION
$00000030 constant MB_ICONEXCLAMATION
$00000040 constant MB_ICONASTERISK
```

	MB_ICONHAND, MB_ICONSTOP ou MB_ICONERROR
	MB_ICONQUESTION
	MB_ICONEXCLAMATION ou MB_ICONWARNING
	MB_ICONASTERISK ou MB_ICONINFORMATION

Figure 18: icônes pour boîte modale

Intégration dans le code Forth :

```
...
    MB_YESNO MB_DEFBUTTON2 or MB_ICONEXCLAMATION or
    MessageBoxA
...
```

Résultat :

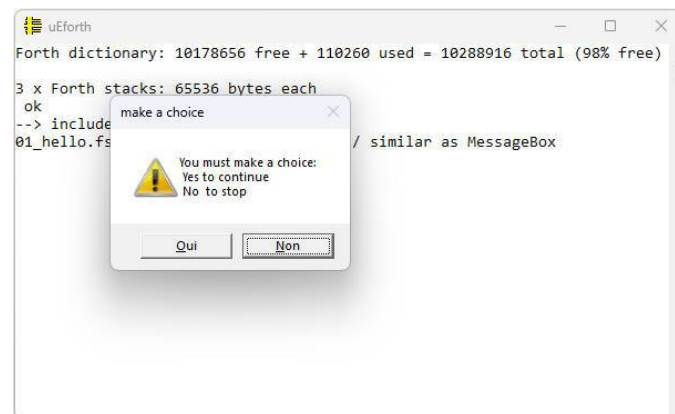


Figure 19: boîte modale avec icône

En résumé, nous avons, avec les fenêtres modales, un moyen simple et efficace pour interagir avec un programme écrit en Forth. Une fenêtre modale ne nécessite pas de réorganiser l'affichage dans la fenêtre parent.

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** :

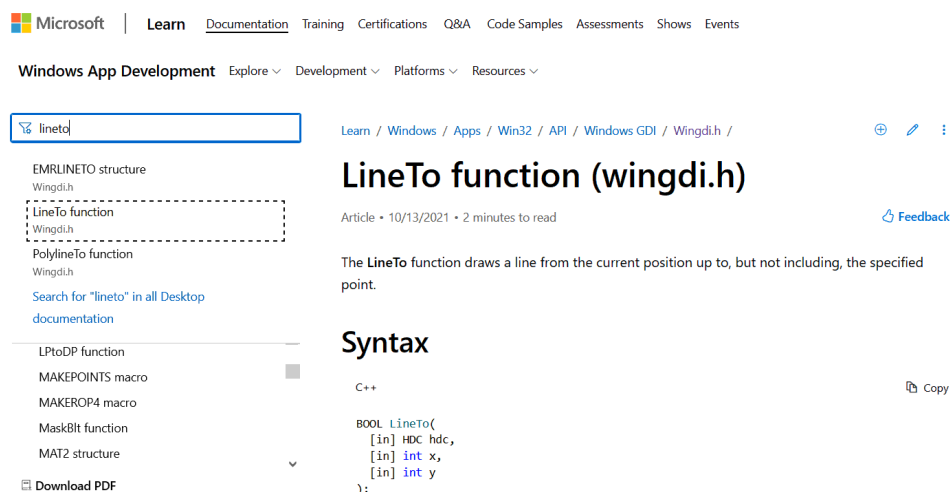


Figure 20: documentation de LineTo

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 mots dans graphics internals

Mots définis dans le vocabulaire **graphics**:

```
graphics definitions
windows also

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

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

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

\ The CloseFigure function close a figure in a path.
z" CloseFigure" 1 gdi32 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 GetPixel ( hdc x y -- color )

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

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

```
gdiError CreateFontA GetCurrentObject SetTextColor TextOutA SetPixel GetPixel
CloseFigure Ellipse Rectangle LineTo MoveToEx 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
```

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

Vous trouverez en ligne tous les mots rajoutés au vocabulaire **graphics** :

<https://github.com/MPETREMANN11/eForth-Windows/blob/main/graphics/Gdi32-definitions.fs>

Trouver les fonctions disponibles dans un fichier dll

eForth Windows fait appel aux fonctions de quatre fichiers DLL:

- **Gdi32.dll** fonctions de gestion graphiques
- **Kernel32.dll** sert d'interface entre les applications et le noyau du système
- **User32.dll** fournit une interface de programmation d'applications
- **Shell32.dll** joue un rôle crucial dans l'interface utilisateur de Windows

Le rôle de eForth Windows est de piocher dans toutes ces fonctions et de proposer des mots à utiliser sans se préoccuper de savoir à quelle librairie est rattaché ce mot.

Les soucis commencent quand on souhaite rajouter un mot lié à une librairie DLL. Prenons comme exemple la fonction **CreateDialog()**, normalement définie dans **User32.dll** si on se fie à la documentation Microsoft. Tentative de définition en Forth :

```
z" CreateDialog" 4 User32 CreateDialog
```

Et là, c'est l'échec!

Dans la documentation Windows, il y a une variante **CreateDialogA()**. Faisons une nouvelle tentative :

```
z" CreateDialogA" 4 User32 CreateDialogA
```

Ca ne passe toujours pas...

La raison est que le système Windows évolue de version en versions. Certaines fonctions disparaissent, d'autres sont réécrites. Il existe un système de prototypes proposant des alias de fonctions. Mais ces mécanismes ne sont disponibles qu'au travers des APIs de développement Windows.

Pour nous, développeur Forth, si on ne veut pas tâtonner, c'est de répertorier toutes les fonctions intégrées à un fichier DLL installé sur notre machine.

Dependency Walker

Ce programme est disponible ici :

<https://www.dependencywalker.com/>

C'est un petit programme, facile à installer et à utiliser.

Au démarrage, cliquer sur *File* et sélectionnez *Open*.

On va analyser le fichier **User32.dll**. Sous windows 11, ce fichier se trouve dans le dossier **Windows --> System32**.

Ne vous inquiétez pas d'éventuels messages d'erreur.

Résultat de l'ouverture de **User32.dll** :

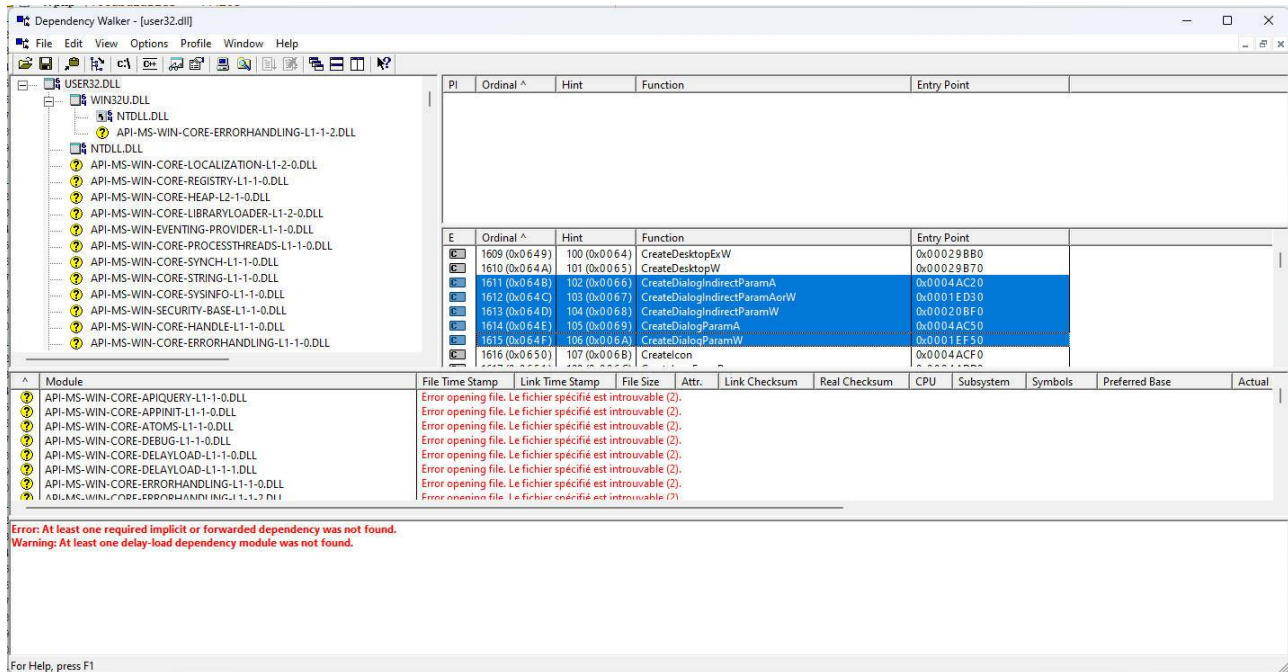


Figure 21: recherche des fonctions de User32.dll

Ici, on a retrouvé des fonctions commençant bien par **CreateDialog**, mais on ne retrouve que des variantes. Aucune chance donc d'effectuer une liaison avec **CreateDialog()** ou **CreateDialogA()**. On devra donc utiliser la variante la plus adaptée.

Cette liste peut être copiée pour éviter d'avoir à ouvrir **User32.dll** en permanence.

Vous pouvez renommer une fonction non managée à votre convenance dans votre code eForth :

```
z" SendMessageA"      4 User32 SendMessage ( hWnd msg wParam iParam -- LRESULT )
```

Ici, la fonction **SendMessageA()** est utilisée sous nom **SendMessage** dans eForth. Mais c'est risqué. L'autre solution est de créer un alias :

```
z" SendMessageA"      4 User32 SendMessageA ( hWnd msg wParam iParam -- LRESULT )

: SendMessage
  SendMessageA ;
```

Certaines fonctions peuvent avoir un nombre de paramètres différent. L'intérêt de passer ensuite par un alias, c'est de traiter les éventuels messages d'erreur :

```
\ write text
z" TextOutA"      5 Gdi32 TextOutA ( hdc x y lpString c -- fl )

: TextOut ( hdc x y lpString c -- fl )
  TextOutA dup 0= if
    abort" TextOut ERROR"
  then ;
```

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.

Premiers tracés graphiques

Les Interfaces de Programmation d'Application (API) de l'environnement Windows sont très complexes. La maîtrise de ces API est essentiel pour réaliser des programmes fonctionnant ailleurs que dans un simple terminal texte. Pour être très cash, c'est une "usine à gaz", pour le dire poliment.

Les fervents zélotes de Linux seraient tentés d'affirmer que Linux serait plus simple. J'exprime quelques doutes sur ce point, au vu de certains codes sources Linux. En réalité, les deux systèmes, Windows et Linux, ont des démarches similaires pour ce qui est de gérer des environnements graphiques.

Je vais donc tenter d'aborder ce vaste sujet par paliers, en espérant ne pas vous perdre en cours de route.

Ouvrir une fenêtre

Le premier mot à connaître, pour ouvrir une fenêtre graphique, est **window**. Ce mot accepte deux paramètres :

- largeur de la fenêtre, exprimée en pixels;
- hauteur de la fenêtre, exprimée en pixels.

Exemple:

```
graphics
600 400 window
```

Ceci ouvre une fenêtre de 640 pixels de large et 400 pixels de haut. Cette fenêtre est indépendante de la fenêtre eForth Windows.

Ici, le mot **graphics** sélectionne le vocabulaire **graphics**. Contenu de ce vocabulaire :

```
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
```

Une partie des primitives sont définies dans le vocabulaire **internals** :

```
graphics internals vlist \ display:
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
```

Dans le chapitre *Etendre le vocabulaire graphics pour Windows*, on a déjà abordé la manière d'étendre le contenu de ces deux vocabulaires.

Tracé de lignes

L'ouverture d'une fenêtre avec **window**, stocke le handle de l'environnement courant dans la valeur **hdc**.

Avant d'effectuer un tracé, on positionne le pointeur graphique avec **MoveToEx**, dont voici la définition :

```
\ MoveToEx updates the current position
z" MoveToEx"      4 Gdi32 MoveToEx ( hdc x y LPPOINT -- fl )
```

Paramètres de **MoveToEx** :

- **hdc** Handle vers un contexte de périphérique
- **x** Spécifie la coordonnée x, en unités logiques, de la nouvelle position
- **y** Spécifie la coordonnée y, en unités logiques, de la nouvelle position
- **LPPOINT** Pointeur vers une structure **POINT** qui reçoit la position actuelle précédente. Si ce paramètre est un pointeur de valeur **NULL** , la position précédente n'est pas retournée

Exemple :

```
hdc 20 20 NULL moveToEx drop
```

Ici on a positionné le pointeur graphique aux coordonnées 20 20. De base, cette position correspond à 20 pixels depuis le bord haut et le bord gauche de la fenêtre courante. On va maintenant faire quelques tracés depuis cette position, avec le mot **LineTo** dont voici d'abord la définition:

```
\ LineTo draws a line from the current position to,
\ but not including, the specified point.
z" LineTo"      3 Gdi32 LineTo ( hdc x y -- fl )
```

Paramètres de **LineTo** :

- **hdc** Handle vers un contexte de périphérique
- **x** Spécifie la coordonnée x, en unités logiques, de la nouvelle position
- **y** Spécifie la coordonnée y, en unités logiques, de la nouvelle position

Exemple :

```
windows also
graphics internals

: draw
```



```

hdc 20 20 NULL moveToEx drop
hdc 90 20 LineTo drop
hdc 90 50 LineTo drop
hdc 20 90 LineTo drop
hdc 20 20 LineTo drop
;

: run09
  600 400 window
  draw
  key drop
  bye
;

```

Résultat :

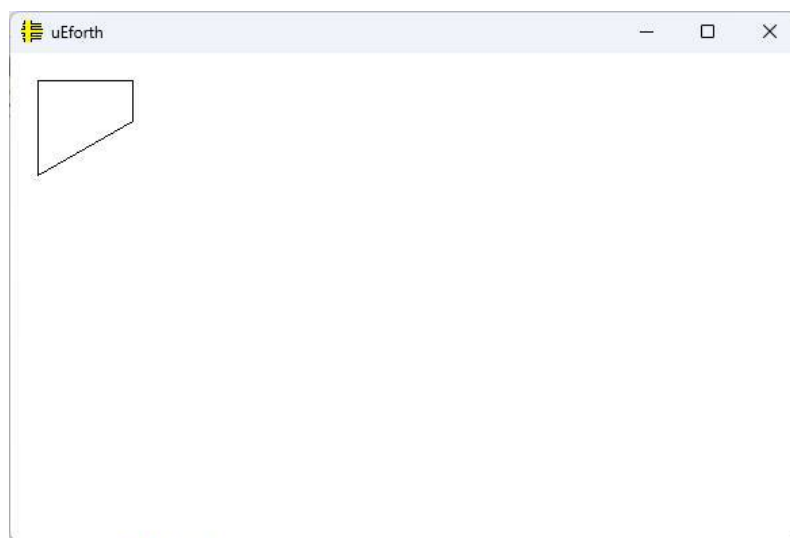


Figure 22: Premier tracé avec LineTo

La couleur de tracé, par défaut, est noir.

Coloration des tracés graphiques

Avec WinCGI, rien n'est simple, mais tout est logique. Et le changement de couleur fait partie de cette immense usine à gaz que l'on va tenter de maîtriser.

Pour modifier la couleur d'une ligne tracée avec **LineTo**, vous devez définir la couleur du stylo (pen) avant d'exécuter **LineTo**.

On définit préalablement un stylo, avec le mot **CreatePen**, dont voici la définition :

```
z" CreatePen"      3 Gdi32 CreatePen ( iStyle cWidth color -- hPen )
```

L'exécution de **CreatePen** nécessite ces paramètres :

- **iStyle** style du tracé
- **cWidth** épaisseur du tracé

- **color** couleur du tracé

En retour, on récupère un handle nécessaire pour sélectionner le stylo. Exemple:

```
0 constant PS_SOLID
0 value HPEN_RED
PS_SOLID 1 $FF $00 $00 RGB CreatePen to HPEN_RED
```

Ici, on a défini une valeur **HPEN_RED** qui stocke le handle issu de l'exécution de **CreatePen**.

On peut facilement définir plusieurs stylos :

```
0 value HPEN_RED
0 value HPEN_BLUE

: setPens ( -- )
  PS_SOLID    3 $FF $00 $00 RGB CreatePen to HPEN_RED
  PS_DOT      1 $00 $00 $FF RGB CreatePen to HPEN_BLUE
;
```

Ici, on définit deux stylos dont les handles sont stockés dans **HPEN_RED** et **HPEN_BLUE**.

Pour sélectionner un stylo, on passe par son handle en utilisant le mot **SelectObject** dont voici la définition :

```
z" SelectObject"    2 Gdi32 SelectObject ( hdc h -- f1 )
```

Paramètres :

- **hdc** handle pour le contrôleur de domaine
- **h** handle de l'objet à sélectionner.

Utilisation :

```
: draw
  setPens
  hdc HPEN_RED SelectObject drop
  hdc 20 20 NULL moveToEx drop
  hdc 90 20 LineTo drop
  hdc 90 50 LineTo drop
  hdc 20 90 LineTo drop
  hdc 20 20 LineTo drop

  hdc HPEN_BLUE SelectObject drop
  hdc 25 25 NULL moveToEx drop
  hdc 95 25 LineTo drop
  hdc 95 55 LineTo drop
  hdc 25 95 LineTo drop
  hdc 25 25 LineTo drop
;
```

Résultat :

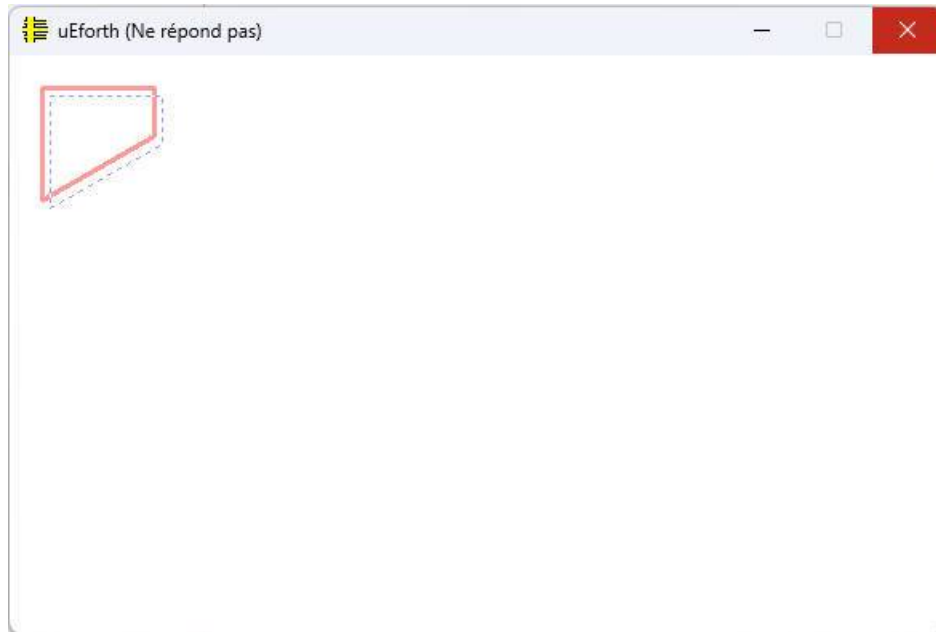


Figure 23: Utilisation de deux stylos

En résumé, voici les étapes pour colorer un tracé simple :

- définir un stylo avec **CreatePen**
- sélectionner un stylo avec **SelectObject**
- faire le tracé graphique

Dans l'exemple ci-dessus, le tracé bleu est en pointillé. En fait, derrière l'apparente complexité sur la manière de gérer les tracés et couleurs de tracés, nous n'avons plus à re-crée une librairie graphique en Forth.

Tracé de formes géométriques

Dans le précédent chapitre, on a appris à définir et utiliser un stylo pour la couleur des tracés. Avant d'aborder le tracé de formes, on va voir comment définir et utiliser un pinceau.

Colorier l'intérieur des formes

Pour colorer l'intérieur d'une forme, on doit d'abord définir le mot **CreateSolidBrush** :

```
z" CreateSolidBrush" 1 Gdi32 CreateSolidBrush ( color -- HBRUSH )
```

Paramètre :

- **color** couleur de remplissage

Exemple :

```
0 value HBRUSH_BLUE

: setBrushes ( -- )
    $00 $00 $FF RGB CreateSolidBrush to HBRUSH_BLUE
;
```

A l'exécution de **setBrushes**, le handle délivré par **CreateSolidBrush** est stocké dans **HBRUSH_BLUE**.

Ce handle **HBRUSH_BLUE** sera ensuite sélectionné par **SelectObject**.

Tracé de rectangles

Le tracé de rectangles est effectué par le mot **Rectangle**, dont voici la définition :

```
\ draw a rectangle
z" Rectangle" 5 gdi32 Rectangle ( hdc left top right bottom -- fl )
```

Paramètres :

- **hdc** handle du contexte graphique
- **left** position du bord gauche
- **top** position du bord haut
- **right** position du bord droit
- **bottom** position du bord bas

Exemple :

```
0 value HPEN_RED
```

```

0 value HPEN_BLUE

: setPens ( -- )
  PS_SOLID    3 $FF $00 $00 RGB CreatePen to HPEN_RED
  PS_DOT      1 $00 $00 $FF RGB CreatePen to HPEN_BLUE
;

create RECT01
  10 L, 10 L, 220 L, 80 L,

create RECT02
  25 L, 25 L, 235 L, 95 L,

: draw
  setPens
  setBrushes
  hdc RECT01 GetRect Rectangle drop
  hdc HPEN_RED   SelectObject drop
  hdc HBRUSH_BLUE SelectObject drop
  hdc RECT02 GetRect Rectangle drop
;

```

Résultat :

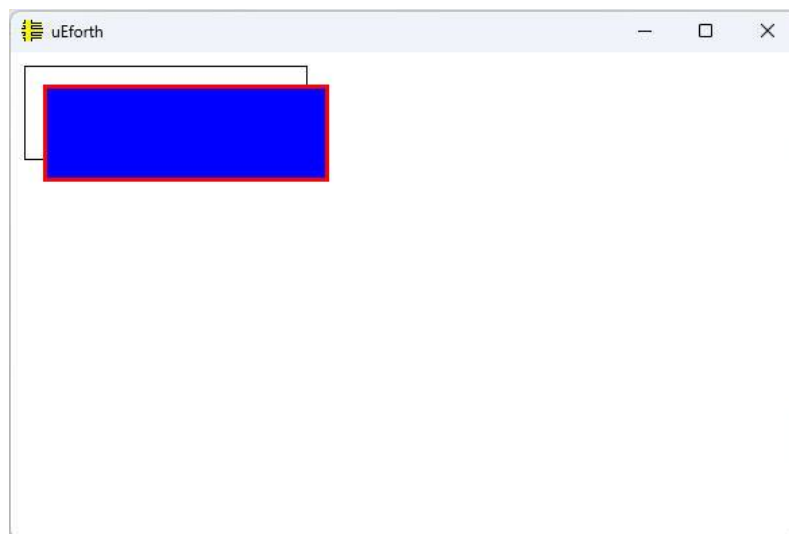


Figure 24: tracé de rectangles

Ici, on a défini deux rectangles via les structures **RECT01** et **RECT02**.

Le second rectangle a son bord et son fond coloré par sélection des objets **HPEN_RED** et **HBRUSH_BLUE**.

Tracé de polygones

Le tracé de polygones utilise une structure de type **POINT**. Pour définir un tracé, il suffit préalablement de créer un tableau contenant des paires de coordonnées x y, chaque paire correspondant à la position d'une extrémité de segment décrivant les arêtes du polygone:

```
create POYGON01
    30 L, 37 L,
    44 L, 22 L,
    58 L, 41 L,
    43 L, 54 L,
    56 L, 68 L,
    43 L, 79 L,
    31 L, 65 L,
    20 L, 75 L,
    5 L 58 L,
    19 L, 47 L,
    6 L, 31 L,
    19 L, 22 L,
```

Ici, on a défini un tableau nommé **POLYGON01** contenant 12 arêtes. Pour tracer le polygone, on utilise le mot **Polygon** dont voici la définition :

```
\ draw a polygone
z" Polygon"    3 Gdi32 Polygon ( hdc *apt cpt -- fl )
```

Paramètres :

- **hdc** handle du contexte graphique
- **apt** pointeur vers le tableau contenant les données du polygone
- **cpt** taille du tableau

Utilisation :

```
: drawPolygon
    setPens
    setBrushes
    hdc HPEN_RED    SelectObject drop
    hdc HBRUSH_BLUE SelectObject drop
    hdc POYGON01 12 Polygon drop
;
```

Résultat :

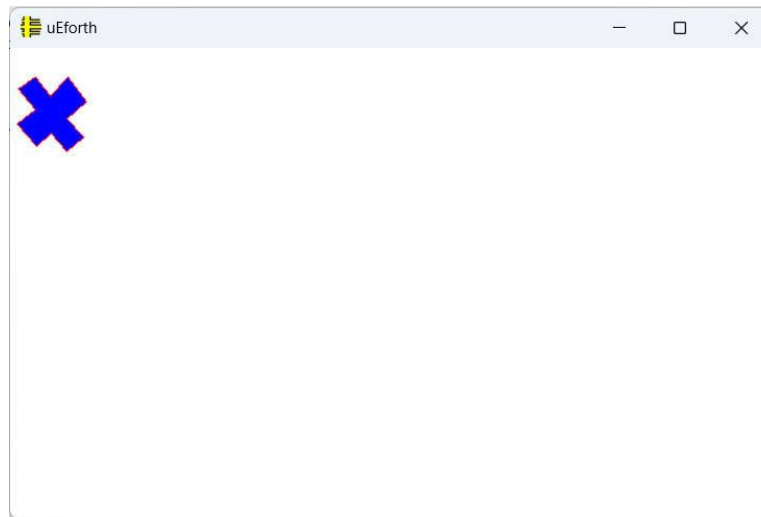


Figure 25: tracé d'un polygone

Il n'est pas nécessaire de revenir au point de départ. Le mot **Polygon** ferme automatiquement la figure en connectant la dernière arête à la première arête tracées.

Tracé d'ellipses

Pour tracer une ellipse, on utilisera le mot **Ellipse** dont voici la définition :

```
\ Draw ellipse
z" Ellipse"      5 gdi32 Ellipse      ( hdc left top right bottom -- fl )
```

Paramètres :

- **hdc** handle du contexte graphique
- **left** position du bord gauche
- **top** position du bord haut
- **right** position du bord droit
- **bottom** position du bord bas

Exemple :

```
: drawEllipse ( -- )
  setPens
  setBrushes
  hdc HPEN_RED    SelectObject drop
  hdc 10 10 400 200 Ellipse drop
  hdc HPEN_BLUE   SelectObject drop
  hdc 30 30 420 220 Ellipse drop
;
```

Résultat :

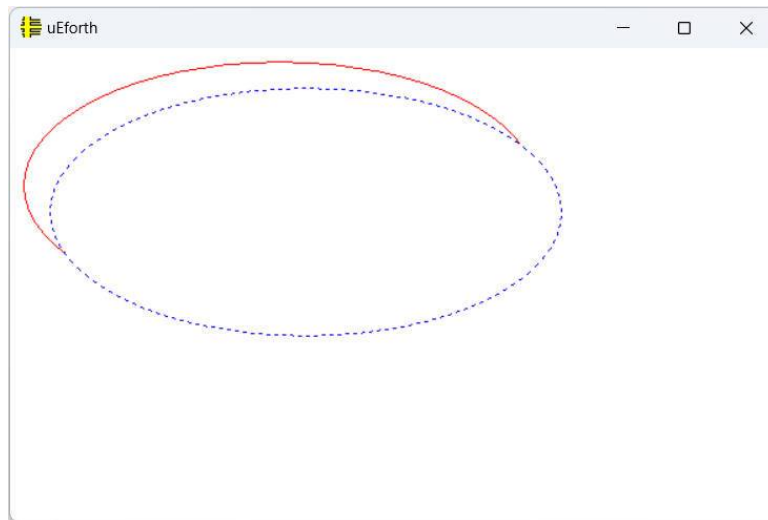


Figure 26: tracé d'ellipse

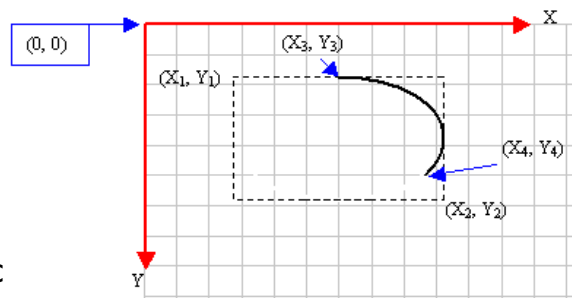
Tracé d'arcs

Les arcs sont des courbes tracées avec le mot **Arc** dont voici la définition :

```
\ Draw arc
z" Arc"      9 gdi32 Arc      ( hdc x1 y1 x2 y2 x3 y3 x4 y4 -- f1 )
```

Paramètres :

- **hdc** handle du contexte graphique
- **x1** et **y1** coin supérieur droit englobant l'arc
- **x2** et **y2** coin inférieur droit englobant l'arc
- **x3** et **y3** position de début du tracé de l'arc
- **x4** et **y4** position de fin du tracé de l'arc



Exemple :

```
0 value HPEN_RED
0 value HPEN_BLUE

: setPens ( -- )
  PS_SOLID    5 $FF $00 $00 RGB CreatePen to HPEN_RED
  PS_SOLID    5 $00 $00 $FF RGB CreatePen to HPEN_BLUE
;

: drawArcs ( -- )
  setPens
  hdc HPEN_RED    SelectObject drop
  hdc 10 10 400 200 10 105 400 105 Arc drop
  hdc HPEN_BLUE    SelectObject drop
  hdc 10 10 400 200 400 105 10 105 Arc drop
```


;

Résultat :

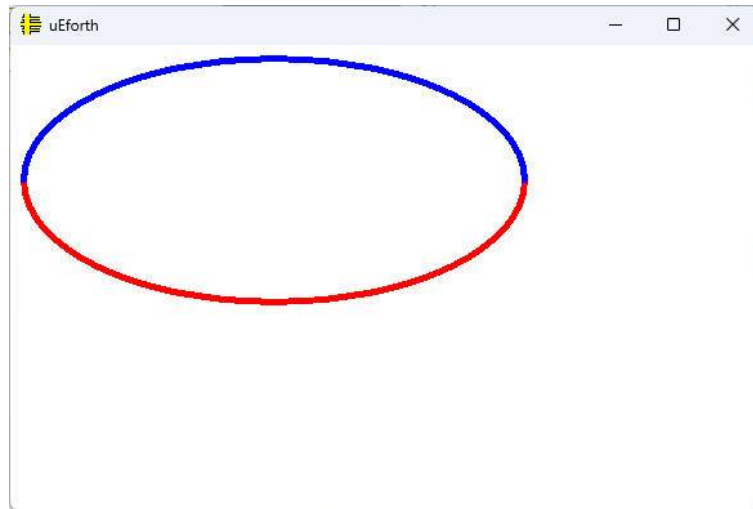


Figure 27: tracé d'arcs

Ici, nous voyons deux demi arcs se suivant en changeant de couleur.

Pour résumer, nous avons abordé les mots de dessin essentiels permettant de créer des fonds graphiques: quadrillages, boîtes, encadrés, etc...

Afficher du texte dans l'environnement graphique

Au lancement de eForth Windows, les textes qui s'affichent sont gérés dans une console de terminal Windows. On peut faire beaucoup de choses dans cette console, sauf changer de fonte, positionner du texte au pixel près, encadrer du texte, intégrer des tracés graphiques...

En affichant du texte dans une fenêtre Windows, on va passer dans un univers complexe, mais riche et plein de possibilités.

DrawTextA

Le mot **DrawTextA** n'est pas défini dans le vocabulaire **graphics**. Voici sa définition:

```
only forth
windows definitions

\ draws formatted text in the specified rectangle.
z" DrawTextA" 5 User32 DrawTextA
```

Ce nouveau mot requiert cinq paramètres :

- **hdc** handle de la fenêtre courante;
- **lpchText** pointeur vers la chaîne à afficher;
- **cchText** longueur de la chaîne à afficher. Si cette valeur est -1, la chaîne doit se terminer par le code zéro;
- **LPRECT** lpch pointeur vers une structure de type RECT;
- **format** formatage du texte.

Entrons dans le vif du sujet avec un premier exemple :

```
only
windows also
graphics internals

: RECT! { left top right bottom addr -- }
  left   addr ->left   L!
  top    addr ->top    L!
  right  addr ->right  L!
  bottom addr ->bottom L!
;

create LPRECT
  RECT allot
```

```

: STR01 ( -- addr len )
  s" This is my first example." ;

: DRAWtext ( -- )
  10 10 300 80 LPRECT RECT!
  hdc STR01 LPRECT DT_TOP DrawTextA
;

: run04
  600 400 window 100 ms
  DRAWtext
  key drop
;

```

Résultat :

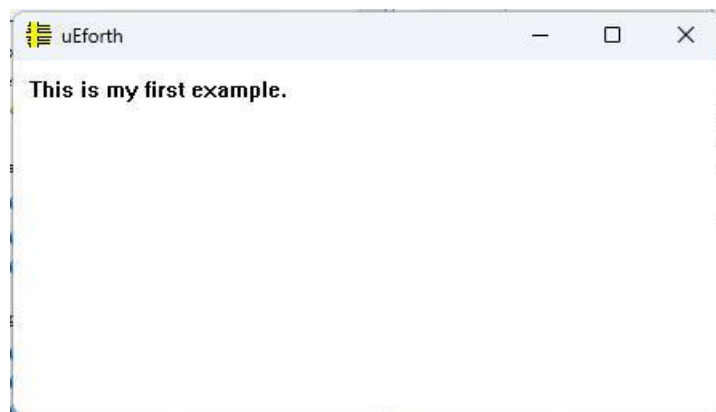


Figure 28: premier exemple d'affichage de texte

Ca a l'air touffu, mais cet exemple intègre l'initialisation de divers paramètres.

Définition de la zone de tracé du texte

L'affichage du texte s'effectue dans une zone rectangulaire. Cette zone rectangulaire est définie dans une structure **RECT**. Si vous ne maîtrisez pas les structures, lisez le chapitre *Structures de données pour eForth*.

Pour charger une structure **RECT**, utiliser **SetRect** qui se charge d'affecter les dimensions du rectangle :

```

create LPRECT
  RECT allot

LPRECT 10 10 300 80 SetRect

```

La séquence **LPRECT 10 10 300 80 SetRect** enregistre les paramètres du rectangle dans **LPRECT**.

Ici, le rectangle fait 290 pixels de large. Que se passe-t-il si on essaie d'afficher un texte trop long?

```
: STR01 ( -- addr len )  
  s" This is my first example.. I try to draw a very long text in this  
  graphic window." ;  
  
: DRAWtext ( -- )  
  10 10 300 80 LPRECT RECT!  
  hdc STR01 LPRECT DT_TOP DrawTextA  
;
```

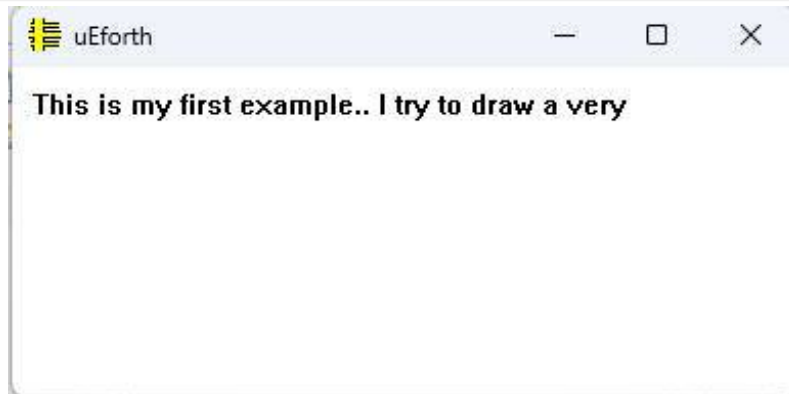


Figure 29: Affichage de texte trop long

Quand le texte dépasse le bord du rectangle, il est tronqué.

Formatage du texte

Le formatage du texte tracé par **DrawText** est contrôlé par son dernier paramètre. Dans notre code, c'est défini via la constante **DT_TOP**.

Voici les principaux formats :

- **DT_TOP** Affichage depuis le haut du rectangle
- **DT_LEFT** Aligne le texte depuis la gauche du rectangle
- **DT_CENTER** Centre le texte horizontalement dans le rectangle
- **DT_RIGHT** Aligne le texte à droite du rectangle
- **DT_VCENTER** Centre le texte verticalement
- **DT_BOTTOM** Justifie le texte depuis le bas du rectangle
- **DT_WORDBREAK** Césure les mots

Il y a encore d'autres formats, mais leur étude sort du cadre de ce chapitre.

Ces formats peuvent se combiner avec l'opérateur **OR**, sous réserve de ne pas combiner des formats incompatibles.

```

: STR01 ( -- addr len )
  s" This is my first example.. I try to draw a very long text in this
  graphic window." ;

: FORMATTING ( -- n )
  DT_TOP          \ draw from top
  DT_WORDBREAK OR \ break words
;

: DRAWtext ( -- )
  10 10 300 80 LPRECT RECT!
  hdc STR01 LPRECT FORMATTING DrawTextA
;

```

Résultat :

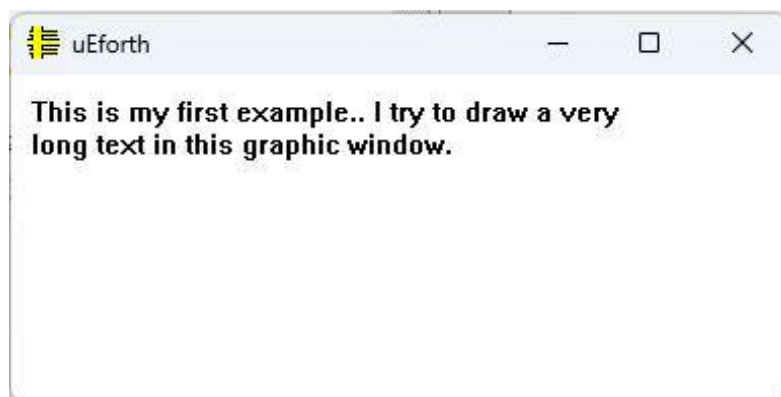


Figure 30: Formatage du texte

On notera que la césure de texte s'effectue entre deux mots. Il n'est donc pas nécessaire de s'occuper de la longueur du texte à afficher. Si un mot risque de déborder, il est automatiquement reporté sur la ligne suivante dans le rectangle. Rajoutons le centrage de texte et dans un rectangle plus étroit :

```

: FORMATTING ( -- n )
  DT_TOP          \ draw from top
  DT_WORDBREAK OR \ break words
  DT_CENTER       OR \ center text
;

: DRAWtext ( -- )
  10 10 200 120 LPRECT RECT!
  hdc STR01 LPRECT FORMATTING DrawTextA
;

```

Résultat :

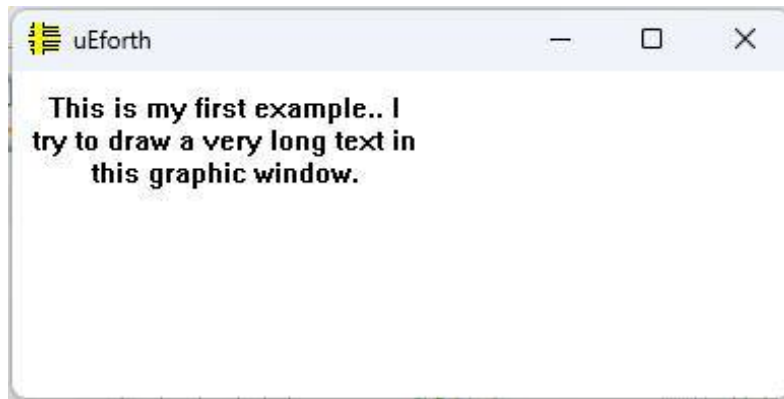


Figure 31: Affichage dans un rectangle plus étroit.

A chaque fois, le mot **DrawTextA** dépose une valeur au sommet de la pile. Dans le dernier exemple, il dépose la valeur 48. Cette valeur correspond à la dimension verticale, en pixels, du texte qui a été affiché.

Si vous n'utilisez pas ce paramètre, éliminez-le avec **DROP**.

Changer la couleur du texte

Pour changer la couleur du texte, il faut au préalable définir le mot **SetTextColor** :

```
only forth
windows also
graphics definitions

\ Set text color
z" SetTextColor"          2 Gdi32 SetTextColor ( hdc color -- f1 )
```

Le mot **SetTextColor** a besoin de deux paramètres :

- **hdc** handle du contexte courant;
- **color** couleur du texte.

La couleur peut être codée en une seule valeur 32 bits, **\$00FF00** par exemple.

Sinon, pour être certain de sélectionner une couleur valable, vous pouvez utiliser le mot **RGB** :

```
windows
$FF $00 $00 RGB \ push 32 bits color code on stack
```

Chaque valeur traitée par **RGB** doit être comprise dans l'intervalle [0..255] en décimal, ou [\$00..\$FF] en hexadécimal. La première valeur correspond à la composante rouge, la seconde à la composante verte, la troisième à la composante bleue.

```
: DRAWtext ( -- )
  LPRECT 10 10 200 120 SetRect drop
  hdc $FF $00 $00 RGB SetTextColor drop
  hdc STR01 LPRECT FORMATTING DrawTextA drop
```

```
;
```

Résultat :

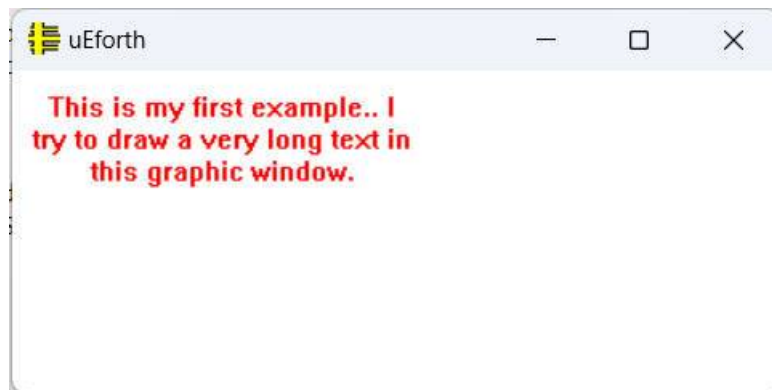


Figure 32: Affichage du texte en couleur

TextOutA

Ce mot est défini comme ceci :

```
\ write text
z" TextOutA"      5 Gdi32 TextOutA ( hdc x y lpString c -- fl )
```

Liste des paramètres :

- **hdc** handle de contexte;
- **x** coordonnée x de la position d'affichage du texte;
- **y** coordonnée y de la position d'affichage du texte;
- **lpString** adresse de la chaîne texte à dessiner. La chaîne n'a pas besoin d'être terminée par zéro, car c spécifie la longueur de la chaîne;
- **c** longueur de la chaîne.

Utilisation :

```
: String01 s" This is a first test string..." ;
: String02 s" This is a second test string..." ;

: TXTout ( -- )
  hdc 10 10 String01 TextOutA drop
  hdc 10 30 String02 TextOutA drop ;
```

Résultat :

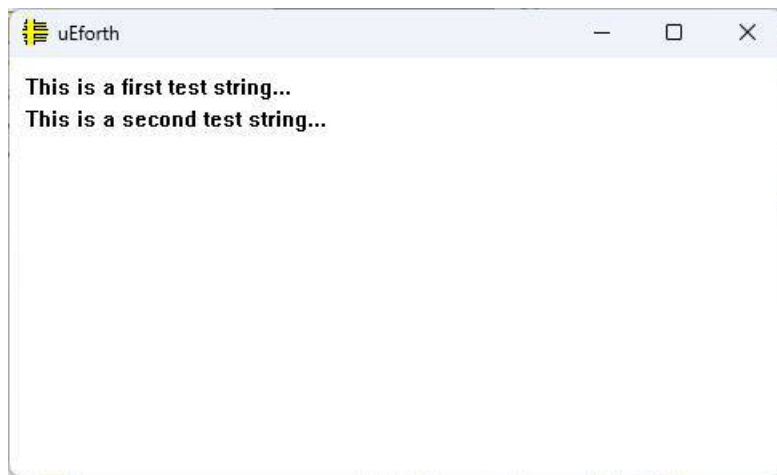


Figure 33: Affichage avec TextOutA

Avec eForth Windows, il semble que le mot **SetTextColor** n'aie pas d'effet sur la couleur du texte.

Les fontes de caractères

Une fonte de caractères est un ensemble de glyphes, c'est-à-dire de représentations visuelles de caractères, d'une même police d'écriture, de même style, corps et graisse.

Fonte et police

La fonte se distingue du caractère (*typeface* ou *font family* en anglais) qui regroupe tous les corps et graisses d'une même famille. Ainsi, **Helvetica** est un caractère. L'Helvetica romain gras 10 points est une fonte, et l'Helvetica romain gras 12 points est une autre fonte.

Quelques familles de caractères et les fontes associées :

- **Arial** Arial, Arial Italic, Arial Bold, Arial Bold Italic
- **Courier New** Courier New, Courier New Italic, Courier New Bold, Courier New Bold Italic
- **Verdana** Verdana, Verdana Italic, Verdana Bold, Verdana Bold Italic

Ici, nous n'avons cité que trois familles de fontes. Il en existe beaucoup d'autres. Certaines ne sont pas utilisables via **Gdi32.dll**.

CreateFontA

L'accès à une fonte s'effectue via le mot **CreateFontA** dont voici la définition :

```
\ Creates a logical font with the specified characteristics  
z" CreateFontA" 14 Gdi32 CreateFontA ( [14_params] -- hFont )
```

Le mot **CreateFontA** utilise 14 paramètres. Dans l'ordre :

- **cHeight** hauteur de la police en pixels
- **cWidth** largeur moyenne des caractères
- **cEscapement** angle d'échappement
- **cOrientation** orientation
- **cWeight** poids de la police (gras, normal)
- **bItalic** booléen style italique
- **bUnderline** booléen Souligné
- **bStrikeOut** booléen barré
- **iCharSet** Jeu de caractères

- **iOutPrecision** précision de sortie
- **iClipPrecision** précision du découpage
- **iQuality** qualité de sortie
- **iPitchAndFamily** Pitch et famille de la police
- **pszFaceName** nom de la police

cHeight

Ce paramètre détermine en hauteur, en unités logiques, de la cellule de caractère ou du caractère de la police.

Cwidth

Largeur moyenne, en unités logiques, des caractères dans la police demandée. Si cette valeur est égale à zéro, le mappeur de polices choisit une valeur de correspondance la plus proche.

Cescapement

Angle, en dixièmes de degrés, entre le vecteur d'échappement et l'axe X de l'appareil. Le vecteur d'échappement est parallèle à la ligne de base d'une ligne de texte.

Corientation

Angle, en dixièmes de degrés, entre la ligne de base de chaque caractère et l'axe X de l'appareil.

Cweight

Poids de la police comprise entre 0 et 1000. Par exemple, 400 est normal et 700 en gras. Si cette valeur est égale à zéro, une pondération par défaut est utilisée.

Les valeurs suivantes sont définies pour des raisons pratiques: **FW_DONTCARE FW_THIN FW_EXTRALIGHT FW_ULTRALIGHT FW_LIGHT FW_NORMAL FW_REGULAR FW_MEDIUM FW_SEMIBOLD FW_DEMIBOLD FW_BOLD FW_EXTRABOLD FW_ULTRABOLD FW_HEAVY FW_BLACK**

bltalic

Spécifie une police italique si elle est définie sur **TRUE**.

Bunderline

Spécifie une police soulignée si elle est définie sur **TRUE**.

BstrikeOut

Police barré si elle est définie sur **TRUE**.

lcharSet

Jeu de caractères. Les valeurs suivantes sont prédéfinies: **ANSI_CHARSET**
BALTIC_CHARSET **CHINESEBIG5_CHARSET** **DEFAULT_CHARSET** **EASTEUROPE_CHARSET**
GB2312_CHARSET **GREEK_CHARSET** **HANGUL_CHARSET** **MAC_CHARSET** **OEM_CHARSET**
RUSSIAN_CHARSET **SHIFTJIS_CHARSET** **SYMBOL_CHARSET** **TURKISH_CHARSET**
VIETNAMESE_CHARSET

iOutPrecision

Précision de sortie. La précision de sortie définit à quel point la sortie doit correspondre à la hauteur, à la largeur, à l'orientation des caractères, à l'échappement, au pitch et au type de police de la police demandée.

lclipPrecision

Précision du découpage. La précision du découpage définit comment découper des caractères qui se trouvent partiellement en dehors de la zone de découpage.

lquality

Qualité de sortie. La qualité de sortie définit la manière avec laquelle GDI doit tenter de faire correspondre les attributs de police logique à ceux d'une police physique réelle.

lpitchAndFamily

Pitch et famille de la police. Les deux bits d'ordre inférieur spécifient le pitch de la police et peuvent être l'une des valeurs suivantes :

```
DEFAULT_PITCH  
FIXED_PITCH  
VARIABLE_PITCH
```

Les quatre bits d'ordre élevé spécifient la famille de polices et peuvent être l'une des valeurs suivantes :

- **FF_DECORATIVE** Polices de nouveauté. Le gothique en est un exemple.
- **FF_DONTCARE** Utilisez la police par défaut.
- **FF_MODERN** Polices avec une largeur de trait constante, avec ou sans empattements. Pica, Elite et Courier New en sont des exemples.
- **FF_ROMAN** Polices avec une largeur de trait variable et avec empattements. MS Serif en est un exemple.
- **FF_SCRIPT** Polices conçues pour ressembler à de l'écriture manuscrite. Script et Cursive en sont des exemples.
- **FF_SWISS** Polices avec une largeur de trait variable et sans empattements. MS Sans Serif en est un exemple.

pszFaceName

Pointeur vers une chaîne terminée par null qui spécifie le nom de police de la police. La longueur de cette chaîne ne doit pas dépasser 32 caractères, y compris le caractère null de fin.

Exemple de sélection d'une fonte :

```
0 value hFontArial16    \ handle for selected font

: selectFontArial ( -- )
    16                \ Hauteur de la police en pixels
    0                 \ Largeur moyenne des caractères
    0                 \ Angle d'échappement
    0                 \ Orientation
    FW_NORMAL         \ Poids de la police (gras, normal)
    FALSE             \ Style italique
    FALSE             \ Souligné
    FALSE             \ Barré
    DEFAULT_CHARSET   \ Jeu de caractères
    OUT_DEFAULT_PRECIS
    CLIP_DEFAULT_PRECIS
    DEFAULT_QUALITY
    DEFAULT_PITCH FF_SWISS or
    z" Arial"         \ Nom de la police
    CreateFontA to hFontArial16
;

```

La création d'une fonte par **CreateFontA** est intégrée dans le mot **selectFontArial**.

La taille de la fonte est fixée à 16. A l'issue de l'exécution de **CreateFontA**, le handle résultant est stocké dans la valeur **hFontArial16**.

Voici comment utiliser ce handle **hFontArial16** :

```
: lpString s" This is a test string..." ;

: TXTout ( -- )
    selectFontArial
    hdc hFontArial16 SelectObject drop
    hdc 10 10 lpString TextOutA drop
;

```

Dans la définition de **TXTout**, on effectue ces opérations :

- **selectFontArial** sélectionne la fonte et stocke le handle
- **hdc hFontArial16 SelectObject drop** rattache le handle **hFontArial16** à l'environnement graphique **hdc**

- **hdc 10 10 lpString TextOutA drop** affiche la chaîne de caractères qui est définie dans **lpString**

Résultat :

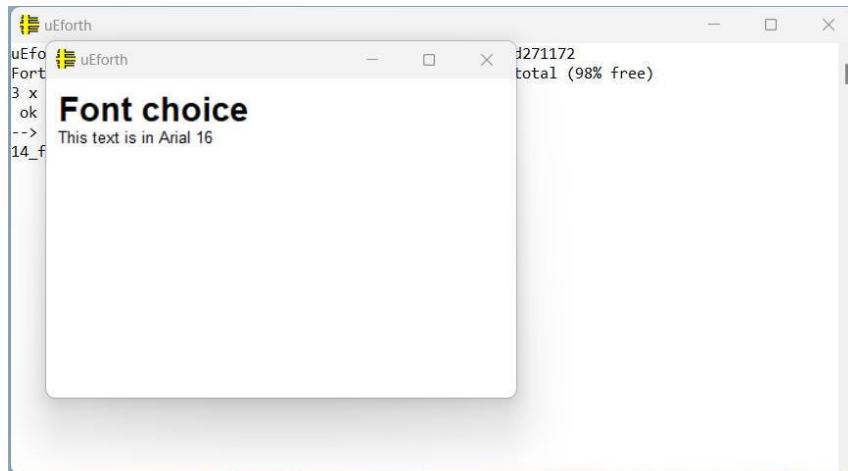


Figure 34: sélection fonte Arial

Ici, le code final qui exploite **TXTout** :

```
\ define window dimension
400 constant WINDOW_WIDTH
300 constant WINDOW_HEIGHT

: run14
  WINDOW_WIDTH WINDOW_HEIGHT window
  TXTout
  begin
    poll
    IDLE event = if

    then
      event FINISHED =
    until
  ;
```

En résumé, il faudra définir autant de *handle* pour chaque fonte que vous souhaitez utiliser. Ensuite, la sélection d'une fonte s'effectue au travers du *handle* de chaque fonte définie.

COMx: Ouvrir et gérer un port série

Ici, on va voir comment gérer le port série depuis l'API Windows pour communiquer entre eForth Windows et les appareils connectés à ce port série.

Pour savoir si un port série est actif, cliquer sur le champ de recherche Windows et tapez Gestionnaire de périphériques.

Ouvrez le gestionnaire de périphériques et recherchez Ports (COM) :

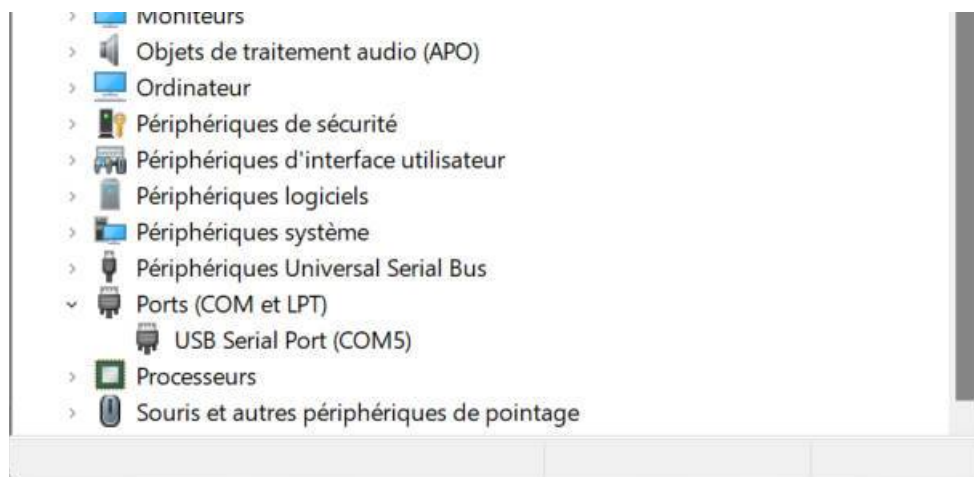


Figure 35: recherche port série ouvert

Ici, le port série **COM5** est disponible.

ATTENTION: s'il n'y a aucun appareil connecté en liaison série, le gestionnaire de périphériques n'indiquera aucun port série disponible.

Pour la suite, on suppose le programme eForth paramétré pour accéder à notre port série **COM5**. A charge pour vous d'adapter le programme en fonction du port série disponible quand vous connectez un appareil.

Le port série

Le port série est apparu sur les premiers ordinateurs comme le premier interface simple pour communiquer avec des accessoires, dont des imprimantes, modems, etc...

Le câblage était souvent un cauchemar, entre fiches DB9, DB25, fils croisés, signaux de handshake pris en compte ou ignorés. Avec l'apparition des ports USB, ces soucis ont disparu.

Aujourd'hui, de petites cartes d'interface convertissent les signaux USB en signaux compatibles RS322. De son côté, Windows permet de paramétrer la communication. C'est ce paramétrage que nous allons aborder.

Ouverture du port série

Le mot **CreateFileA** permet d'ouvrir un port série. Même si dans l'énoncé de ce mot on retrouve *File*.

Windows considère les ports série comme ressources au même titre que les vrais fichiers les accès réseau, etc...

Le mot **CreateFile** est déjà défini dans le vocabulaire **windows**. Utilisation :

```
only forth also
windows also structures

$10000000 constant GENERIC_ALL      \ All possible access rights
$20000000 constant GENERIC_EXECUTE \ Execute access
$40000000 constant GENERIC_WRITE   \ Write access
$80000000 constant GENERIC_READ    \ Read access

\ parameters used by CreateFileA
z" COM5" value CF_lpFileName
GENERIC_READ GENERIC_WRITE or value CF_dwDesiredAccess
0      value CF_dwShareMode
NULL   value CF_lpSecurityAttributes
OPEN_EXISTING value CF_dwCreationDisposition
0      value CF_dwFlagsAndAttributes
NULL   value CF_hTemplateFile

: .error ( -- )
  getLastError
  dup ." Error: " . space
  case
    2 of ." port indisponible "   endof
    5 of ." acces refuse "       endof
  endcase
;

-1 constant INVALID_HANDLE_VALUE

0 value hSerial

\ create handle for serial port
: create-serial ( -- hSerial )
  CF_lpFileName
  CF_dwDesiredAccess
  CF_dwShareMode
  CF_lpSecurityAttributes
  CF_dwCreationDisposition
  CF_dwFlagsAndAttributes
  CF_hTemplateFile
```

```

CreateFileA
dup INVALID_HANDLE_VALUE = if
    .error
then
;

```

Chaque paramètre utilisé par **CreateFileA** est défini dans une valeur, exemple :

```
z" COM5" value CF_lpFileName
```

Si vous voulez gérer un autre port série, il suffit d'ajuster ce paramètre.

Toute la création du port série est définie dans le mot **create-serial**. L'exécution de ce mot empile un handle qui est stocké dans la valeur **hSerial**.

Ce handle **hSerial** devient notre seul point d'accès à tous les autres mots de gestion du port série.

Récupération des paramètres du port série

Pour gérer tous les paramètres du port série, il est nécessaire de définir une structure DCB :

```

struct DCB
i32 field ->DCBlength
i32 field ->BaudRate
i32 field ->fBinary
i32 field ->fParity
i32 field ->fOutxCtsFlow
i32 field ->fOutxDsrFlow
i32 field ->fDtrControl
i32 field ->fDsrSensitivity
i32 field ->fTXContinueOnXoff
i32 field ->fOutX
i32 field ->fInX
i32 field ->fErrorChar
i32 field ->fNull
i32 field ->fRtsControl
i32 field ->fAbortOnError
i32 field ->fDummy2
i16 field ->wReserved
i16 field ->XonLim
i16 field ->XoffLim
i8 field ->ByteSize
i8 field ->Parity
i8 field ->StopBits
i8 field ->XonChar
i8 field ->XoffChar
i8 field ->ErrorChar
i8 field ->EofChar

```



```
i8 field ->EvtChar
i16 field ->wReserved1
```

Création du mot **dcbSerialParams** utilisant cette structure :

```
\ Sets the control parameter for a serial communication device
\ struct DCB \ transfered in Kernel32-definitions.fs

\ DCB structure for COM port
create dcbSerialParams
  DCB allot
```

Le fichier **Kernel32-definitions.fs** est disponible ici:

<https://github.com/MPETREMANNN11/eForth-Windows/tree/main/extensions>

Pour remplir **dcbSerialParams**, on définit le mot **get-serial-params** :

```
\ store serial parameters in DCB structure
: get-serial-params ( hSerial -- )
  dcbSerialParams GetCommState 0 = if
    abort" Error: GetCommState"
  then
  ;
```

Ce mot utilise **GetCommState**, dont voici la définition :

```
\ Retrieves the current control settings for a specified communications device
z" GetCommState" 2 Kernel32 GetCommState ( hSerial lpDCB -- f1 )
```

Paramètres :

- **hSerial** handle du port série ouvert
- **lpDCB** structure à remplir

GetCommState remplit la structure **dcbSerialParams** avec les paramètres du port série ouvert précédemment par **create-serial**.

Modification des paramètres du port série

Pour valider les nouveaux paramètres du port série, on utilise le mot **SetCommState** dont voici la définition :

```
\ configures a communications device according to the specifications in a DCB
z" SetCommState" 2 Kernel32 SetCommState ( hSerial lpDCB -- f1 )
```

Voici comment le mot **SetCommState** est utilisé dans **set-serial-params** :

```
2 constant EVENPARITY
3 constant MARKPARITY
0 constant NOPARITY
1 constant ODDPARITY
```

```

3 constant SPACEPARITY

0 constant ONESTOPBIT      \ 1 stop bit
1 constant ONE5STOPBITS    \ 1.5 stop bits
2 constant TWOSTOPBITS     \ 2 stop bits

\ set speedn byte size, parity and stop bit
: set-speed-8N1 ( dcbStruct -- )
  >r
  115200      r@ !field ->BaudRate
  8           r@ !field ->ByteSize
  NOPARITY    r@ !field ->Parity
  ONESTOPBIT  r> !field ->StopBits
;

\ set serial port with DCB structure
: set-serial-params ( hSerial -- )
  dcbSerialParams SetCommState 0 = if
    abort" Error: GetCommState"
  then
;

```

Initialisation du port série

L'initialisation du port série se déroule en plusieurs étapes :

- récupération du handle rattaché au port COM, ici le port COM5
- récupération des paramètres courants du port ouvert
- modification des paramètres de vitesse, parité...
- affectation des paramètres modifiés au port série ouvert

```

\ initialise serial port
: init-serial
  create-serial to hSerial
  hSerial get-serial-params
  dcbSerialParams set-speed-8N1
  hSerial set-serial-params
;

\ close serial port
: close-serial ( -- )
  hSerial CloseHandle 0 = if
    abort" Error: CloseHandle"
  then
;

```

Le code Forth complet en lien avec cet article est disponible ici :

Transmission et réception via le port série

La transmission via le port série utilise le mot **WriteFile** :

```
variable BytesWritten

: to-serial { addr len -- }
  hSerial addr len BytesWritten NULL WriteFile 0= if
    ." Error : WriteFile " .error
  then
    \ cr BytesWritten @ . ." octets transis"
  ;
```

Après exécution de **to-serial**, le nombre d'octets transmis est stocké dans la variable **BytesWritten**. Pour transmettre une chaîne de commande, il faut compléter la chaîne de caractères correspondant à l'équivalent de CR:

```
create CRLF
  $0D c, $0A c,

: CRLF-to-serial ( -- )
  CRLF 1 to-serial
;

: str-to-serial ( addr len -- )
  to-serial
  CRLF-to-serial
;
```

Dans **CRLF-to-serial**, on ne transmet que le code **\$0D**. Si c'est nécessaire, on peut ajuster la définition :

- **CRLF 1+ 1 to-serial** pour ne transmettre que \$0A
- **CRLF 2 to-serial** pour ne transmettre que \$0D \$0A

Pour recevoir des données depuis le port série, on utilise le mot **ReadFile** :

```
variable BytesRead

256 constant bufferSize
create BUFFER
  bufferSize allot

: from-serial ( -- )
  BUFFER bufferSize erase
  hSerial BUFFER bufferSize BytesRead NULL ReadFile 0= if
    ." Error : BytesRead " .error
```

```
    then
;

```

Le tampon de réception a été limité à 256 octets avec la constante **bufferSize**. Si nécessaire, on peut facilement augmenter la taille du tampon.

L'exécution du mot **from-serial** lit les données récupérées par le port série et les enregistre dans **BUFFER**. La variable **BytesRead** enregistre le nombre de caractères reçus. Pour afficher la chaîne de caractères reçus, voici le mot **.buffer** :

```
: .buffer ( -- )
    buffer BytesRead @ type
;

```

Le test de transmission est effectué entre un PC et une carte **Z79Forth**. La carte **Z79Forth** est connectée sur un port USB et reconnue par le PC sur le port **COM5**.

Voici le test du port série entre le PC et la carte Z79Forth :

```
init-serial
s" hex 3f 2f + . " str-to-serial
from-serial
.buffer \ display: hex 3f 2f + . 6E ok

```

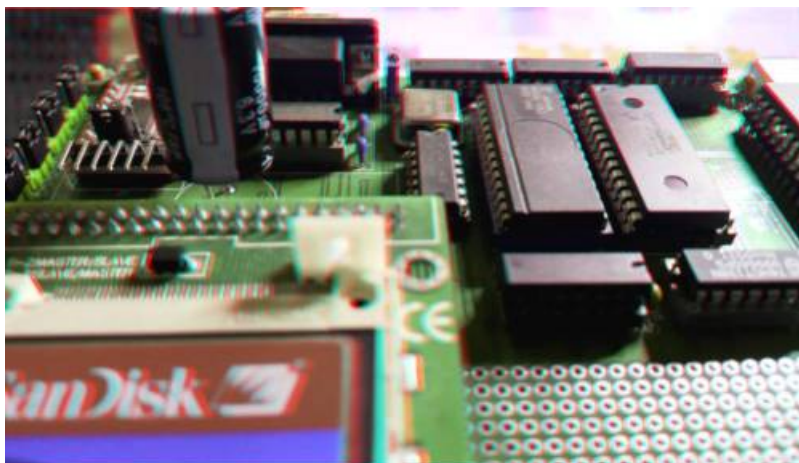


Figure 36: carte Z79Forth

Le code Forth **hex 3f 2f + .** a été transmis à la carte **Z79Forth**. La chaîne affichée par **.buffer** correspond au code Forth exécuté par la carte **Z79Forth**.

En conclusion, le code Forth gérant cette transmission série peut certainement être amélioré. En l'état, il permet déjà l'utilisation d'accessoires, comme un transmetteur LoRa.

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 7.0.7.21

FORTH

=	-rot	└	:	:	:noname
!	?	?do	?dup	┐	."
.s	!	(local)	[[']	[char]
[ELSE]	[IF]	[THEN]	l	f	}transfer
@	*	*/	*/MOD	/	/mod
#	#!	#>	#fs	#s	#tib
+	+	+loop	+to	<	<#
<=	<>	=	>	>=	>BODY
>flags	>flags&	>in	>link	>link&	>name
>params	>R	>size	0<	0<>	0=
1-	1/F	1+	2!	2@	2*
2/	2drop	2dup	3dup	4*	4/
abort	abort"	abs	accept	afliteral	aft
again	ahead	align	aligned	allocate	allot
also	AND	ansi	argc	argv	ARSHIFT
asm	assert	at-xy	base	begin	bg
BIN	binary	bl	blank	block	block-fid
block-id	buffer	bye	c,	C!	C@
CASE	cat	catch	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	fq
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>	rl
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

windows

```

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_CHANGEBCCHAIN 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_MDI TILE WM_MDIMAXIMIZE
WM_MDI NEXT WM_MDI RESTORE WM_MDI ACTIVATE WM_MDI DESTROY WM_MDI CREATE WM_DEVICECHANGE
WM_POWERBROADCAST WM_MOVING WM_CAPTURECHANGED WM_SIZING WM_NEXTMENU WM_EXITMENULOOP
WM_ENTERMENULOOP WM_PARENTNOTIFY WM_MOUSEHWHEEL WM_XBUTTONDOWNLCLK WM_XBUTTONUP
WM_XBUTTONDOWN WM_MOUSEWHEEL WM_MOUSELAST WM_MBUTTONDOWNLCLK WM_MBUTTONUP
WM_MBUTTONDOWN WM_RBUTTONDOWNLCLK WM_RBUTTONUP WM_RBUTTONDOWN WM_LBUTTONDOWNLCLK
WM_LBUTTONUP WM_LBUTTONDOWN WM_MOUSEMOVE WM_MOUSEFIRST CB_MSGMAX CB_GETCOMBOBOXINFO
CB_MULTIPLEADDSTRING CB_INITSTORAGE CB_SETDROPPEDWIDTH CB_GETDROPPEDWIDTH
CB_SETHORIZONTALTEXT CB_GETHORIZONTALTEXT 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_DELETESTRING CB_ADDSTRING CB_SETEDITSEL
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_MENURBUTTONUP WM_ENTERIDLE WM_MENUCHAR
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_KEYLAST WM_UNICHAR WM_SYSDRAWCHAR WM_SYSCHAR WM_SYSKEYUP
WM_SYSKEYDOWN WM_DEADCHAR WM_CHAR WM_KEYUP WM_KEYDOWN WM_KEYFIRST 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_SETRDRAW 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_SHOWNOINACTIVE 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 process-heap HeapReAlloc HeapFree HeapAlloc GetProcessHeap
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 d0<ior 0=ior ior FILE_END FILE_CURRENT FILE_BEGIN FILE_ATTRIBUTE_NORMAL
OPEN_EXISTING CREATE_ALWAYS FILE_SHARE_WRITE FILE_SHARE_READ GetFileSize
SetEndOfFile SetFilePointer MoveFileA DeleteFileA FlushFileBuffers CloseHandle
WriteFile ReadFile CreateFileA NULL wargs-convert wz>sz wargv wargc
CommandLineToArgvW
Shell32 GetModuleHandleA GetCommandLineW GetLastError WaitForSingleObject
GetTickCount Sleep ExitProcess Kernel32 contains? dll sofunc GetProcAddress
LoadLibraryA WindowProcShim SetupCtrlBreakHandler boot_extra windows-builtins
calls

```


Liste des fonctions graphiques de la librairie Gdi32

Ce chapitre résume les fonctions graphiques de la librairie **Gdi32.dll**. Certaines sont installées d'origine dans eForth Windows ou dans les mots graphiques disponibles dans ce fichier :

<https://github.com/MPETREMANN11/eForth-Windows/blob/main/graphics/Gdi32-definitions.fs>

Les mots en lien avec la librairie **Gdi32** et définis dans eForth Windows sont mis en évidence.

Bitmaps

- AlphaBlend
- BitBlt
- CreateBitmap
- CreateBitmapIndirect
- CreateCompatibleBitmap
- CreateDIBitmap
- CreateDIBSection
- CreateDiscardableBitmap
- ExtFloodFill
- FloodFill
- GdiAlphaBlend
- GdiGradientFill
- GdiTransparentBlt
- GetBitmapBits
- GetBitmapDimensionEx
- GetDIBColorTable
- GetDIBits
- GetPixel
- GetStretchBltMode

- GradientFill
- LoadBitmap
- MaskBlt
- PlgBlt
- SetBitmapBits
- SetBitmapDimensionEx
- SetDIBColorTable
- SetDIBits
- SetDIBitsToDevice
- SetPixel
- SetPixelV
- SetStretchBltMode
- StretchBlt
- StretchDIBits
- TransparentBlt

Clip

- ExcludeClipRect
- ExtSelectClipRgn
- GetClipBox
- GetClipRgn
- GetMetaRgn
- GetRandomRgn
- IntersectClipRect
- OffsetClipRgn
- PtVisible
- RectVisible
- SelectClipPath
- SelectClipRgn
- SetMetaRgn

Coordonnées et transformation

- ClientToScreen
- CombineTransform
- DPtoLP
- GetCurrentPositionEx
- GetDisplayAutoRotationPreferences
- GetGraphicsMode
- GetMapMode
- GetViewportExtEx
- GetViewportOrgEx
- GetWindowExtEx
- GetWindowOrgEx
- GetWorldTransform
- LPtoDP
- MapWindowPoints
- ModifyWorldTransform
- OffsetViewportOrgEx
- OffsetWindowOrgEx
- ScaleViewportExtEx
- ScaleWindowExtEx
- ScreenToClient
- SetDisplayAutoRotationPreferences
- SetGraphicsMode
- SetMapMode
- SetViewportExtEx
- **SetViewportOrgEx (hdc x y lppt - fl)**
spécifie quel point de fenêtre correspond à l'origine de la fenêtre d'affichage (0,0).
- SetWindowExtEx
- SetWindowOrgEx

- SetWorldTransform

Couleurs

- AnimatePalette
- CreateHalftonePalette
- CreatePalette
- GetColorAdjustment
- GetNearestColor
- GetNearestPaletteIndex
- GetPaletteEntries
- GetSystemPaletteEntries
- GetSystemPaletteUse
- RealizePalette
- ResizePalette
- SelectPalette
- SetColorAdjustment
- SetPaletteEntries
- SetSystemPaletteUse
- UnrealizeObject
- UpdateColors

Pinceaux

- CreateBrushIndirect
Creates a brush with a specified style, color, and pattern
- CreateDIBPatternBrushPt
Creates a brush with the pattern from a DIB
- CreateHatchBrush
Creates a brush with a hatch pattern and color
- CreatePatternBrush
Creates a brush with a bitmap pattern
- **CreateSolidBrush (param - null|brush)**
crée un pinceau qui sera sélectionné par **SelectObject**

- `GetBrushOrgEx`
Gets the brush origin for a device context
- `GetSysColorBrush`
Gets a handle to a brush that corresponds to a color index
- `PatBlt`
Paints a rectangle
- `SetBrushOrgEx`
Sets the brush origin for a device context
- `SetDCBrushColor`
Sets the current device context brush color.

Stylos

- `CreatePen (iStyle cWidth color - hPen)`
crée un stylo qui sera sélectionné par `SelectObject`
- `CreatePenIndirect`
- `ExtCreatePen`
- `SetDCPenColor`

Lignes et courbes

- `LineTo (hdc x y - fl)`
Tracé de ligne.
- `MoveToEx (hdc x y LPPPOINT - fl)`
sélectionne position de départ de tracé, pour `LineTo` par exemple.

Formes remplies

- `Chord`
- `Ellipse (hdc left top right bottom -- fl)`
tracé d'ellipse.
- `FillRect`
- `FrameRect`
- `InvertRect`
- `Pie`
- `Polygon (hdc *apt cpt -- fl)`
Tracé de polygone fermé,

- PolyPolygon
- **Rectangle (hdc left top right bottom - fl)**
tracé de rectangle.
- RoundRect

Fontes et textes

- AddFontMemResourceEx
- AddFontResource
- AddFontResourceEx
- CreateFont
- CreateFontIndirect
- CreateFontIndirectEx
- CreateScalableFontResource
- DrawText
- DrawTextEx
- EnableEUDC
- FONTENUMPROC
- EnumFontFamilies
- EnumFontFamiliesEx
- FONTENUMPROC
- EnumFonts
- FONTENUMPROC
- ExtTextOut
- GetAspectRatioFilterEx
- GetCharABCWidths
- GetCharABCWidthsFloat
- GetCharABCWidthsI
- GetCharacterPlacement
- GetCharWidth
- GetCharWidth32

- GetCharWidthFloat
- GetCharWidthI
- GetFontData
- GetFontLanguageInfo
- GetFontUnicodeRanges
- GetGlyphIndices
- GetGlyphOutline
- GetKerningPairs
- GetOutlineTextMetrics
- GetRasterizerCaps
- GetTabbedTextExtent
- GetTextAlign
- GetTextCharacterExtra
- GetTextColor
- GetTextExtentExPoint
- GetTextExtentExPointI
- GetTextExtentPoint
- GetTextExtentPoint32
- GetTextExtentPointI
- GetTextFace
- GetTextMetrics
- PolyTextOut
- RemoveFontMemResourceEx
- RemoveFontResource
- RemoveFontResourceEx
- SetMapperFlags
- SetTextAlign
- SetTextCharacterExtra
- **SetTextColor (hdc color -- fl)**
coloration de texte. Utilisable avec **TextOutA**

- SetTextJustification
- TabbedTextOut
- **TextOutA (hdc x y lpstring len - fl)**
tracé de texte.

Contexte appareil

- CancelDC
- ChangeDisplaySettings
- ChangeDisplaySettingsEx
- CreateCompatibleDC
- CreateDC
- CreateIC
- DeleteDC
- DeleteObject
- DrawEscape
- EnumDisplayDevices
- EnumDisplaySettings
- EnumDisplaySettingsEx
- EnumObjects
- GOBJENUMPROC
- GetCurrentObject
- GetDC
- GetDCBrushColor
- GetDCEx
- GetDCOrgEx
- GetDCPenColor
- GetDeviceCaps
- GetLayout
- GetObject
- GetObjectType

- GetStockObject
- ReleaseDC
- ResetDC
- RestoreDC
- SaveDC
- **SelectObject (hdc h - fl)**
sélectionne un objet dans le contexte **hdc**. Peut sélectionner un objet défini par **CreatePen, CreateSolidBrush**.
- SetDCBrushColor
- SetDCPenColor
- SetLayout
- WindowFromDC

Régions

- CombineRgn Combines two regions and stores the result in a third region.
- CreateEllipticRgn Creates an elliptical region.
- CreateEllipticRgnIndirect Creates an elliptical region.
- CreatePolygonRgn Creates a polygonal region.
- CreatePolyPolygonRgn Creates a region consisting of a series of polygons.
- CreateRectRgn Creates a rectangular region.
- CreateRectRgnIndirect Creates a rectangular region.
- CreateRoundRectRgn Creates a rectangular region with rounded corners.
- EqualRgn Checks the two specified regions to determine whether they are identical.
- ExtCreateRegion Creates a region from the specified region and transformation data.
- FillRgn Fills a region by using the specified brush.
- FrameRgn Draws a border around the specified region by using the specified brush.
- GetPolyFillMode Retrieves the current polygon fill mode.
- GetRegionData Fills the specified buffer with data describing a region.
- GetRgnBox Retrieves the bounding rectangle of the specified region.

- **InvertRgn** Inverts the colors in the specified region.
- **OffsetRgn** Moves a region by the specified offsets.
- **PaintRgn** Paints the specified region by using the brush currently selected into the device context.
- **PtInRegion** Determines whether the specified point is inside the specified region.
- **RectInRegion** Determines whether any part of the specified rectangle is within the boundaries of a region.
- **SetPolyFillMode** Sets the polygon fill mode for functions that fill polygons.
- **SetRectRgn** Converts a region into a rectangular region with the specified coordinates.

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.....	64	fconstant.....	64	set-precision.....	63
ansi.....	84	FCOS.....	65	SetCommState.....	137
Arc.....	120	field.....	48	SetRect.....	123
BASE.....	67	forget.....	35	SetTextColor.....	126
BEGIN.....	79	FORTH.....	141	SF!.....	64
BINARY.....	67	fsqrt.....	64	SF@.....	64
c!.....	38	fvariable.....	64	SPACE.....	72
c@.....	38	GetCommState.....	137	struct.....	48
cell.....	44	GIT.....	21	structures.....	48, 84
constant.....	39	graphics.....	107	TextOutA.....	127
create.....	91	HEX.....	67	THEN.....	77
CreateFileA.....	135	HOLD.....	68	to.....	41
CreateFontA.....	129	i8.....	49	UNTIL.....	79
CreatePen.....	113	IF.....	77	value.....	39
CreateSolidBrush.....	116	is.....	88	variable.....	38
DECIMAL.....	67	LineTo.....	112	variables locales.....	40
defer.....	88	LOOP.....	80	voclist.....	84
dll.....	106	mémoire.....	38	WHILE.....	79
DO.....	80	MessageBoxA.....	99	WriteFile.....	139
DOES>.....	91	MoveToEx.....	112	Z79Forth.....	140
DrawTextA.....	122	Netbeans.....	21	35
drop.....	37	order.....	86	35
dump.....	33	pile de retour.....	37	:noname.....	89
dup.....	37	Polygon.....	118	71
editor.....	84	ReadFile.....	139	.s.....	33
Ellipse.....	119	Rectangle.....	116	{.....	40
ELSE.....	77	RECURSE.....	82	}.....	40
EMIT.....	70	REPEAT.....	79	@.....	38
EXECUTE.....	87	RGB.....	126	#.....	68
f.....	63	S".....	71	#>.....	68
F**.....	64	S>F.....	66	#S.....	68
F>S.....	65	see.....	33	+to.....	41
FATAN2.....	64	SelectObject.....	114, 116	<#.....	68