

Le grand livre de ESP32forth

version 1.1 octobre 2023



Auteur(s)

- Marc PETREMANN petremann@arduino-forth.com

Collaborateur(s)

- XXX

Table des matières

Auteur(s).....	1
Collaborateur(s).....	1
Introduction.....	4
Découverte de la carte ESP32.....	5
Présentation.....	5
Les points forts.....	5
Les entrées/sorties GPIO sur ESP32.....	6
Périphériques de l'ESP32.....	8
Pourquoi programmer en langage FORTH sur ESP32?.....	9
Préambule.....	9
Limites entre langage et application.....	9
C'est quoi un mot FORTH?.....	10
Un mot c'est une fonction?.....	10
Le langage FORTH comparé au langage C.....	11
Ce que FORTH permet de faire par rapport au langage C.....	12
Mais pourquoi une pile plutôt que des variables?.....	13
Etes-vous convaincus?.....	13
Existe-t-il des applications professionnelles écrites en FORTH?.....	14
Un vrai FORTH 32 bits avec ESP32Forth.....	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
Dictionnaire / Pile / Variables / Constantes.....	20
Étendre le dictionnaire.....	20
Gestion du dictionnaire.....	20
Piles et notation polonaise inversée.....	21
Manipulation de la pile de paramètres.....	22
La pile de retour et ses utilisations.....	22
Utilisation de la mémoire.....	23
Variables.....	23
Constantes.....	24
Valeurs pseudo-constantes.....	24
Outils de base pour l'allocation de mémoire.....	24
Les variables locales avec ESP32Forth.....	26
Introduction.....	26
Le faux commentaire de pile.....	26
Action sur les variables locales.....	27
Les vocabulaires avec ESP32forth.....	30
Liste des vocabulaires.....	30
Liste du contenu d'un vocabulaire.....	31
Utilisation des mots d'un vocabulaire.....	31

Chainage des vocabulaires.....	32
Adapter les plaques d'essai à la carte ESP32.....	34
Les plaques d'essai pour ESP32.....	34
Construire une plaque d'essai adaptée à la carte ESP32.....	34
Gestion des fichiers sources par blocs.....	36
Les blocs.....	36
Ouvrir un fichier de blocs.....	36
Editer le contenu d'un bloc.....	37
Compilation du contenu des blocs.....	38
Exemple pratique pas à pas.....	38
Conclusion.....	39
Edition des fichiers sources avec VISUAL Editor.....	40
Editer un fichier source FORTH.....	40
Edition du code FORTH.....	40
Compilation du contenu des fichiers.....	41
Le système de fichiers SPIFFS.....	42
Accès au système de fichiers SPIFFS.....	42
Manipulation des fichiers.....	43
Organiser et compiler ses fichiers sur la carte ESP32.....	44
Edition et transmission des fichiers source.....	44
Organiser ses fichiers.....	45
Transférer un gros fichier vers ESP32forth.....	46
Conclusion.....	46

Introduction

Ceci est l'introduction,

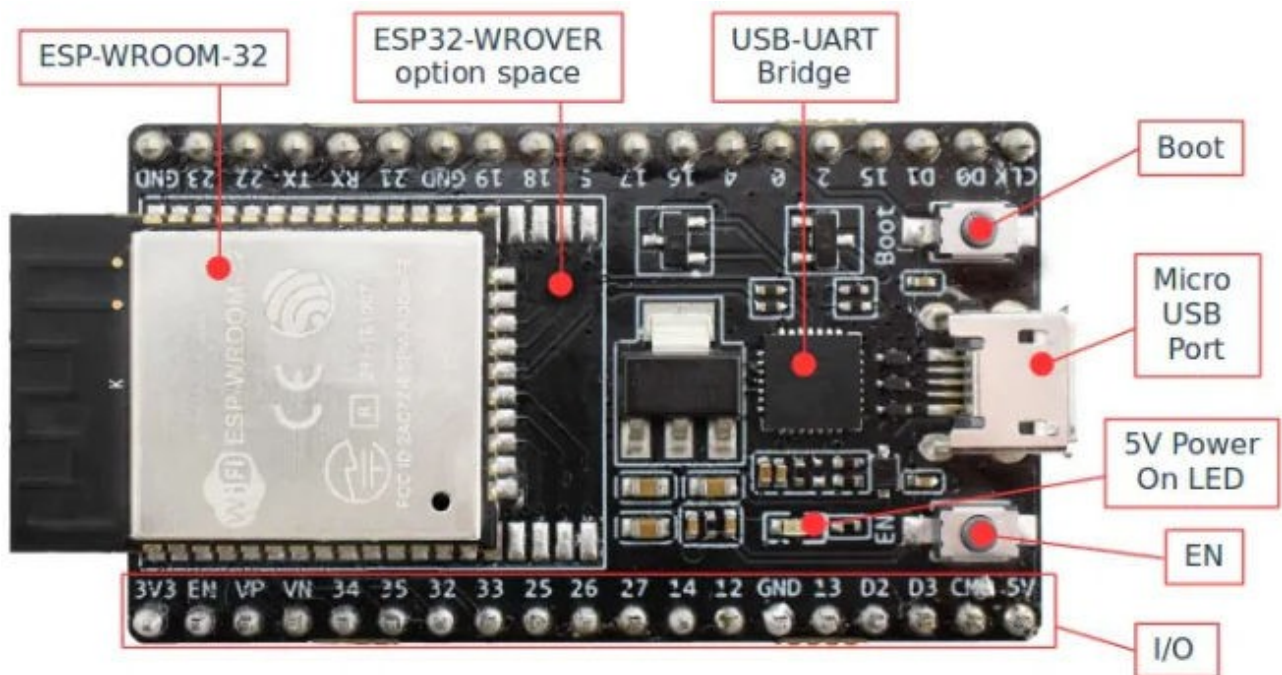
Découverte de la carte ESP32

Présentation

La carte ESP32 n'est pas une carte ARDUINO. Cependant, les outils de développement exploitent certains éléments de l'éco-système ARDUINO, comme l'IDE ARDUINO.

Les points forts

Coté nombre de ports disponibles, la carte ESP32 se situe entre un ARDUINO NANO et



ARDUINO UNO. Le modèle de base a 38 connecteurs:

Les périphériques ESP32 incluent :

- 18 canaux du convertisseur analogique-numérique (ADC)
- 3 interfaces SPI
- 3 interfaces UART
- 2 interfaces I2C
- 16 canaux de sortie PWM
- 2 convertisseurs numérique-analogique (DAC)
- 2 interfaces I2S

- 10 GPIO à détection capacitive

L'ADC (convertisseur analogique-numérique) et le DAC (convertisseur numérique-analogique) les fonctionnalités sont attribuées à des broches statiques spécifiques. Cependant, vous pouvez décider quel les broches sont UART, I2C, SPI, PWM, etc. Il vous suffit de les attribuer dans le code. Ceci est possible grâce à la fonction de multiplexage de la puce ESP32.

La plupart des connecteurs ont plusieurs utilisations.

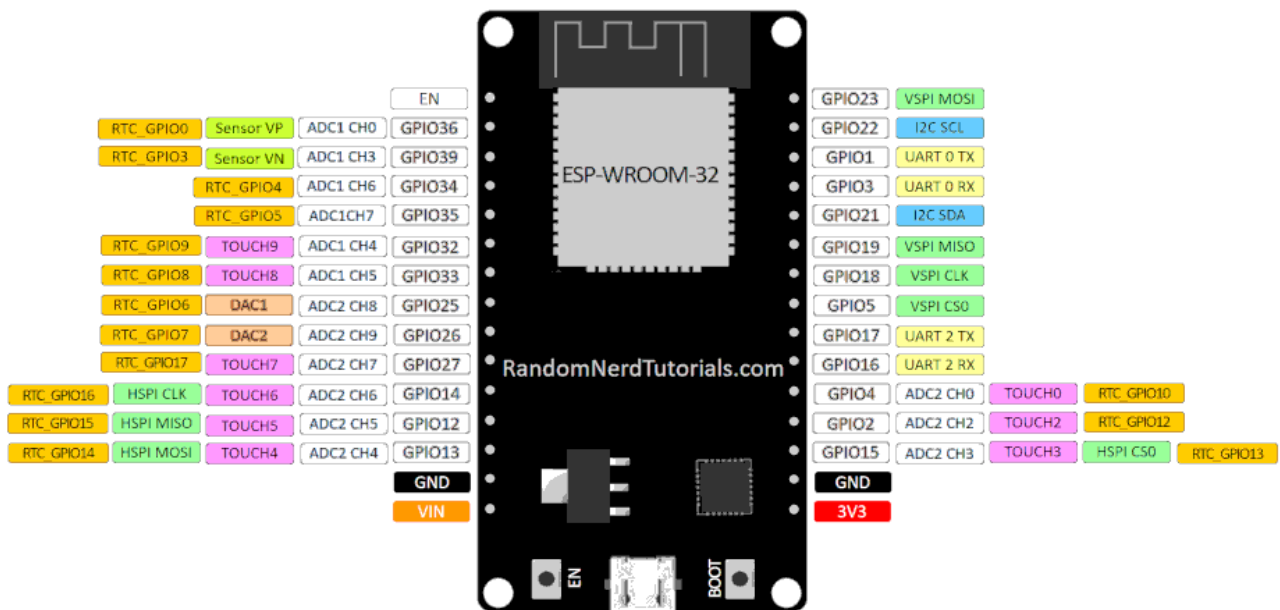
Mais ce qui distingue la carte ESP32, c'est qu'elle est équipée en série d'un support WiFi et Bluetooth, ce que ne proposent les cartes ARDUINO que sous forme d'extensions.

Les entrées/sorties GPIO sur ESP32

Voici, en photo, la carte ESP32 à partir de laquelle nous allons expliquer le rôle des différentes entrées/sorties GPIO:



La position et le nombre des E/S GPIO peut changer en fonction des marques de cartes. Si c'est le cas, seules les indications figurant sur la carte physique font foi. Sur la photo, rangée du bas, de gauche à droite: CLK, SD0, SD1, G15, G2, G0, G4, G16.....G22, G23, GND.



Sur ce schéma, on voit que la rangée basse commence par 3V3 alors que sur la photo, cette E/S est à la fin de la rangée supérieure. Il est donc très important de ne pas se fier au schéma et de contrôler plutôt deux fois le bon branchement des périphériques et composants sur la carte ESP32 physique.

Les cartes de développement basées sur un ESP32 possèdent en général 33 broches hormis celles pour l'alimentation. Certains pins GPIO ont des fonctionnements un peu particuliers:

GPIO	Noms possibles
6	SCK/CLK
7	SCK/CLK
8	SDO/SD0
9	SDI/SD1
10	SHD/SD2
11	CSC/CMD

Si votre carte ESP32 possède les E/S GPIO6, GPIO7, GPIO8, GPIO9, GPIO10, GPIO11, il ne faut surtout pas les utiliser car ils sont reliés à la mémoire flash de l'ESP32. Si vous les utilisez l'ESP32 ne fonctionnera pas.

Les E/S GPIO1(TX0) et GPIO3(RX0) sont utilisés pour communiquer avec l'ordinateur en UART via le port USB. Si vous les utilisez, vous ne pourrez plus communiquer avec la carte.

Les E/S GPIO36(VP), GPIO39(VN), GPIO34, GPIO35 peuvent être utilisés uniquement en entrée. Ils n'ont pas non plus de résistances pullup et pulldown internes intégrées.

La borne EN permet de contrôler l'état d'allumage de l'ESP32 via un fil extérieur. Il est relié au bouton EN de la carte. Lorsque l'ESP32 est allumé, il est à 3.3V. Si on relie ce pin à la masse, l'ESP32 est éteint. On peut l'utiliser lorsque l'ESP32 est dans un boîtier et que l'on veut pouvoir l'allumer/l'éteindre avec un interrupteur.

Périphériques de l'ESP32

Pour interagir avec les modules, capteurs ou circuits électroniques, l'ESP32 comme tout micro-contrôleur possède une multitude de périphériques. Ils sont plus nombreux que sur une carte Arduino classique.

ESP32 dispose des périphériques suivants:

- 3 interfaces UART
- 2 interfaces I2C
- 3 interfaces SPI
- 16 sorties PWM
- 10 capteurs capacitifs
- 18 entrées analogiques (ADC)
- 2 sorties DAC

Certains périphériques sont déjà utilisés par ESP32 lors de son fonctionnement basique. Il y a donc moins d'interfaces possibles pour chaque périphérique.

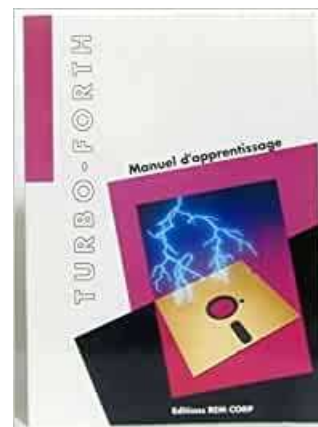
Pourquoi programmer en langage FORTH sur ESP32?

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 Loistech - 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 !      \ disable display echo from terminal
  ['] serial2-type is type
;
: typeToTerm ( -- )
  ['] default-type is type
  -1 echo !    \ enable display echo from terminal
;
```

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

Un mot c'est une fonction?

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

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

aurait son équivalent en langage C:

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

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

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

Oui, mais alors pourquoi FORTH plutôt que C?

Je m'attendais à cette question.

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

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

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

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

```
: >gray ( n -- n' )  
    dup 2/ xor      \ n' = n xor ( 1 time right shift logic )  
    ;
```

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){                // If all bits are received  
    rc5_ok = 1;            // Decoding process is OK  
    detachInterrupt(0);    // Disable external interrupt (INT0)  
    return;  
}
```

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

```
var-j @ 13 >              \ If all bits are received  
    if
```

```

1 rc5_ok ! \ Decoding process is OK
di          \ Disable external interrupt (INT0)
exit
then

```

Voici l'initialisation de registres en langage C:

```

void setup() {
  // Timer1 module configuration
  TCCR1A = 0;
  TCCR1B = 0;          // Disable Timer1 module
  TCNT1  = 0;          // Set Timer1 preload value to 0 (reset)
  TIMSK1 = 1;          // enable Timer1 overflow interrupt
}

```

La même définition en langage FORTH:

```

: setup {
  \ Timer1 module configuration
  0 TCCR1A !
  0 TCCR1B ! \ Disable Timer1 module
  0 TCNT1  ! \ Set Timer1 preload value to 0 (reset)
  1 TIMSK1 ! \ enable Timer1 overflow interrupt
}

```

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 aussi accès à toute la mémoire de la carte ESP32. Connectez-vous à la carte ARDUINO sur laquelle est installé FlashForth, puis tapez simplement:

```
hex here 100 dump
```

Vous devez retrouver 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.

Etes-vous convaincus?

Personnellement, je doute que ce seul chapitre vous convertisse irrémédiablement à la programmation en langage FORTH. En cherchant à maîtriser les cartes ESP32, vous avez deux possibilités:

- programmer en langage C et exploiter les nombreuses bibliothèques disponibles. Mais vous resterez enfermés dans les capacités de ces bibliothèques. L'adaptation des codes en langage C requiert une réelle connaissance en programmation en langage C et maîtriser l'architecture des cartes ESP32. La mise au point de programmes complexes sera toujours un souci.
- tenter l'aventure FORTH et explorer un monde nouveau et passionnant. Certes, ce ne sera pas facile. Il faudra comprendre l'architecture des cartes ESP32, les registres, les flags de registres de manière poussée. 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:

```
\ Turn a port pin on, dont change the others.  
: __/ ( pinmask portadr -- )  
  mset  
  ;  
\ Turn a port pin off, dont change the others.  
: \__ ( pinmask portadr -- )  
  mclr  
  ;
```

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

```
_0_ __/ \ turn LED on
```

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

Avec ESP32forth, voici tous les caractères à votre disposition pouvant composer un mot FORTH:

```
~}|{zyxwvutsrqponmlkjihgfedcba`_  
^]\[ZYXWVUTSRQPONMLKJIHGFEDCBA@?  
>=<;:9876543210/.-,*>('&%$#"!
```

Bonne programmation.

Un vrai FORTH 32 bits avec ESP32Forth

ESP32Forth est un vrai FORTH 32 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 ESP32Forth, l'interpréteur FORTH est disponible. Si vous entrez n'importe quel nombre, il sera déposé sur la pile sous sa forme d'entier 32 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 32 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 ESP32Forth, un caractère ASCII sera désigné par un entier 32 bits, mais dont la valeur sera bornée [32..256[. Exemple:

```
67 emit \ display C
```

Les valeurs en mémoire

ESP32Forth permet de définir des constantes, des variables. Leur contenu sera toujours au format 32 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 32 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'exemple 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 32 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:

```
.i :##  
# 6 base !  
# decimal  
[char] : hold  
.i  
.hms ( n -- )  
<# :## :## # # #> type  
.i  
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 32 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:

```
i morse: ( comp: c -- | exec -- )
  create
    c,
  does>
    dup 1+ swap c@ 0 do
      dup i + c@ emit
    loop
    drop space
i
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....

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, ESP32forth construit le nombre dans 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. ESP32forth 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
```

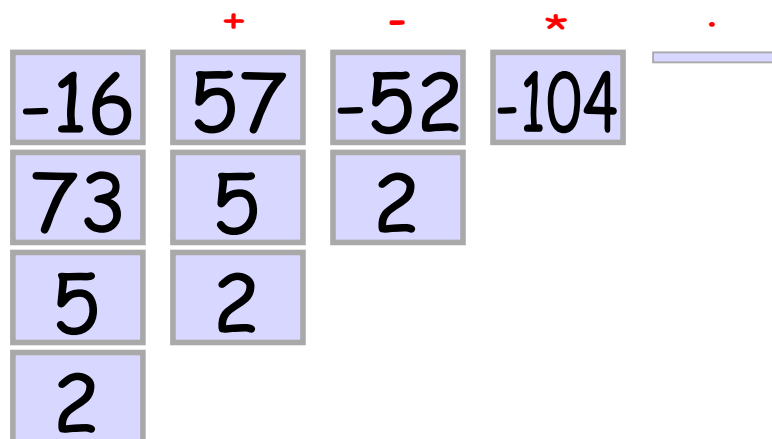
laisse la pile de paramètres dans l'état

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 ESP32forth 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 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, ESP32forth 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, ESP32forth 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 ESP32forth. 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 ESP32forth, les nombres 32 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 @ . \ display: 3
```

Constantes

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

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

\ définit la fréquence du port SPI
4000000 constant SPI_FREQ

\ sélectionne le vocabulaire SPI
only FORTH SPI also

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

Valeurs pseudo-constantes

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

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

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

Outils de base pour l'allocation de mémoire

The words `create` and `allot` are the basic tools for setting aside memory and attaching a convenient label to it. For example, the following transcript shows a new dictionary entry `x`

being created and an extra 16 bytes of memory being allotted to it.

Les mots **create** et **allot** sont les outils de base pour mettre de côté la mémoire et y attacher une étiquette pratique. Par exemple, la transcription suivante montre une nouvelle entrée de dictionnaire x en cours de création et 16 octets de mémoire supplémentaires qui lui sont alloués.

```
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 dans sa mémoire allouée place sur la pile.

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

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

Les variables locales avec ESP32Forth

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 }  
.....
```

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 **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
        @START_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
        @START_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)
  @BRANCH rdrop rdrop space 16 0 (do) -28 local@ j 16 * R@ + + CA@ DUP 32 < OVER 127 > OR
  @BRANCH DROP 46 emit BRANCH emit 1 (+loop) @BRANCH rdrop rdrop 1 (+loop)
  @BRANCH rdrop rdrop -4 local@ base ! cr cr rdrop rdrop rdrop rdrop ;

```

Les vocabulaires avec ESP32forth

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

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

Liste des vocabulaires

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

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

```
--> internals voclist      \ affiche
registers
ansi
editor
streams
tasks
rtos
sockets
Serial
ledc
SPIFFS
SD_MMC
SD
WiFi
Wire
ESP
structures
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**.

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:

```
sockets vlist
```

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

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

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: le mot **l** est défini dans les deux vocabulaires **asm** et **editor**.

C'est encore plus flagrant avec le mot **server**, défini dans les vocabulaires **httpd**, **telnetd** et **web-interface**.

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 **serial2-type** qui utilise le mot **Serial2.write** défini dans le vocabulaire **serial**:

```
serial \ Selection vocabulaire Serial
: serial2-type ( a n -- )
  Serial2.write drop
;
```

La seconde solution permet d'intégrer un seul mot d'un vocabulaire spécifique:

```
: serial2-type ( a n -- )
  [ serial ] Serial2.write [ FORTH ] \ compile mot depuis
vocabulaire serial
drop
```

```
;
```

La sélection d'un vocabulaire peut être effectuée implicitement depuis un autre mot du vocabulaire FORTH.

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**:

```
asm xtensa
order \ affiche:      xtensa >> asm >> FORTH
only
order \ affiche:      FORTH
```

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

```
only
order \ affiche:      FORTH
asm also
order \ affiche:      asm >> FORTH
xtensa
order \ affiche:      xtensa >> asm >> FORTH
```

Voici une séquence de chaînage compacte:

```
only asm also xtensa
```

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

```
only
order \ affiche:      FORTH
also ledc also serial also SPIFFS
order \ affiche:      SPIFFS >> FORTH
      \               Serial >> FORTH
      \               ledc >> FORTH
      \               FORTH
```

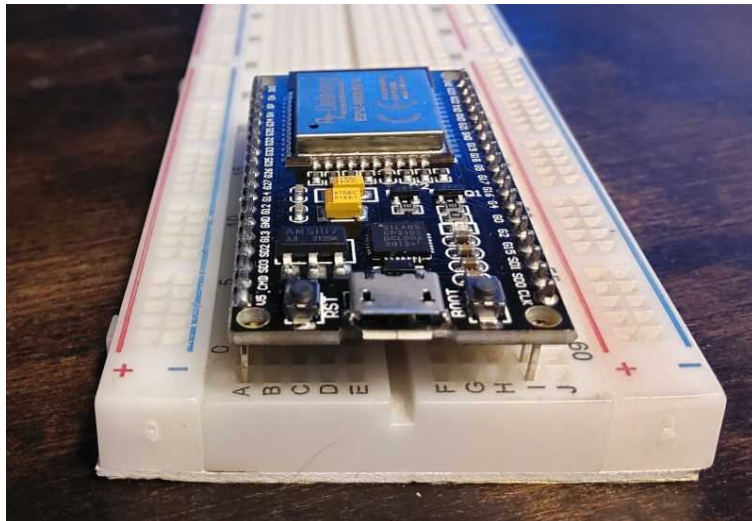
L'ordre de recherche, ici, commencera par le vocabulaire **SPIFFS**, puis **Serial**, puis **ledc** et pour finir le vocabulaire **FORTH**:

- 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;

Adapter les plaques d'essai à la carte ESP32

Les plaques d'essai pour ESP32

Vous venez de recevoir vos cartes ESP32. Et première mauvaise surprise, cette carte s'intègre très mal à la plaque d'essai:

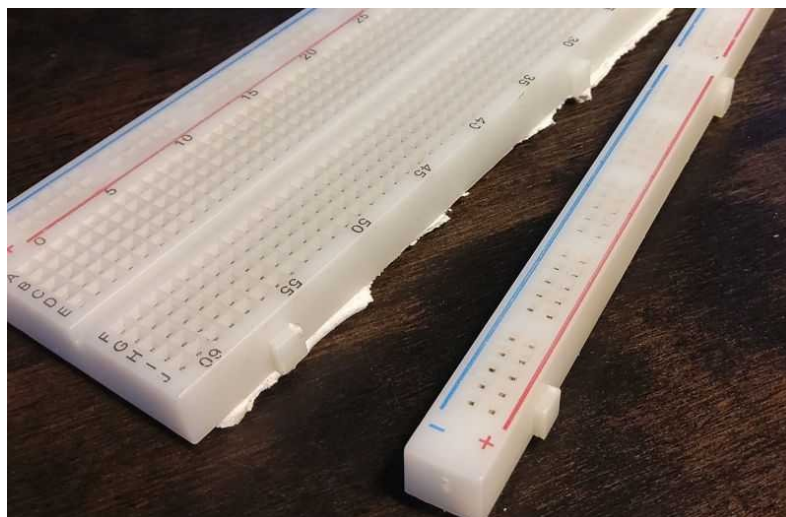


Il n'existe pas de plaque d'essai spécialement adaptée aux cartes ESP32.

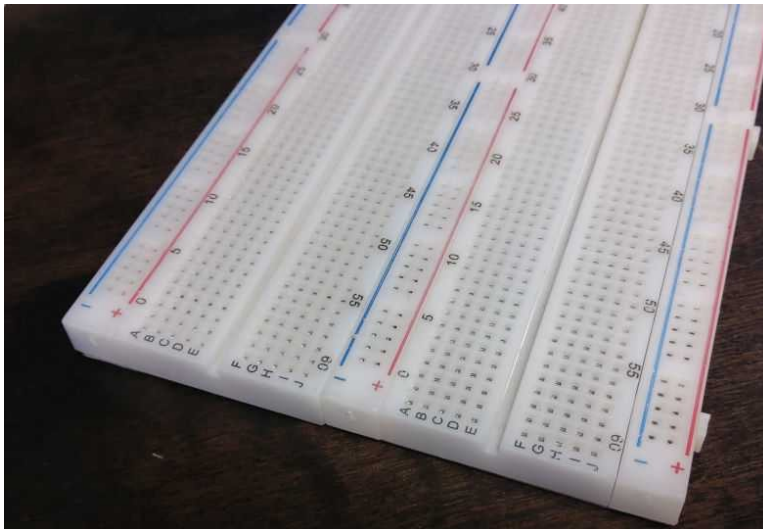
Construire une plaque d'essai adaptée à la carte ESP32

On va construire notre propre plaque d'essai. Pour celà, il faut disposer de deux plaques d'essai identiques.

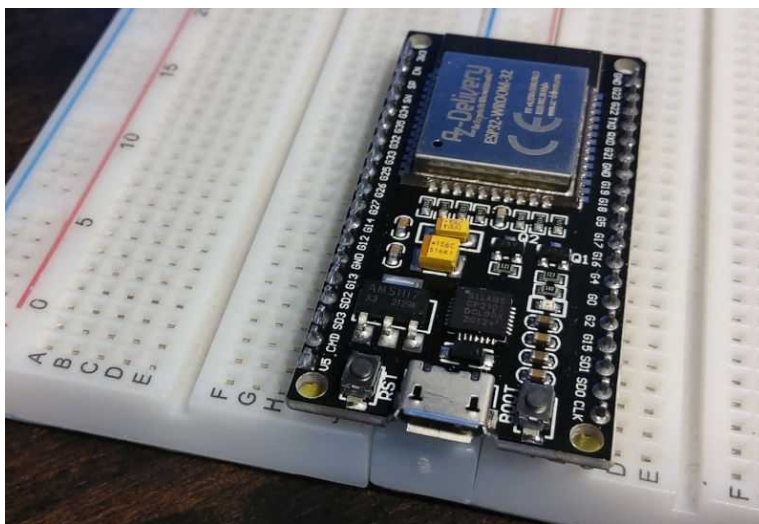
Sur l'une des plaques, on va retirer une ligne d'alimentation. Pour ce faire, utilisez un cutter et découpez par dessous. Vous devez pouvoir séparer cette ligne d'alimentation comme ceci:



On peut ensuite réassembler la carte entière avec cette carte. Vous avez des chevrons sur les cotés des plaques d'essai pour les relier ensemble:



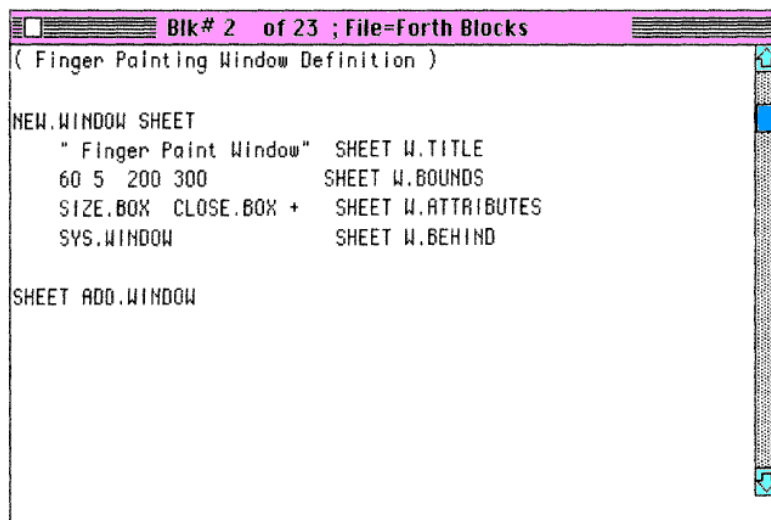
Et voilà! On peut maintenant placer notre carte ESP32:



Les ports E/S peuvent être étendus sans difficulté.

Gestion des fichiers sources par blocs

Les blocs



Ici un bloc sur un ancien ordinateur:

Un bloc est un espace de stockage dont l'unité a comme dimensions 16 lignes de 64 caractères. La taille d'un bloc est donc de $16 \times 64 = 1024$ octets. C'est très exactement la taille d'un Kilo-octet!

Ouvrir un fichier de blocs

Un fichier est déjà ouvert par défaut au démarrage de ESP32forth.

C'est le fichier **blocks.fb**.

En cas de doute, exécutez **default-use**.

Pour savoir ce qu'il y a dans ce fichier, utilisez les commandes de l'aditeur en tapant d'abord **editor**.

Voici nos premières commandes à connaître pour gérer le contenu des blocs:

- **l** liste le contenu du bloc courant
- **n** sélectionne le bloc suivant
- **p** sélectionne le bloc précédent

ATTENTION: un bloc a toujours un numéro compris entre 0 et n. Si vous vous retrouvez avec un numéro de bloc négatif, cela génère une erreur.

Editer le contenu d'un bloc

Maintenant que nous savons sélectionner un bloc en particulier, voyons comment y insérer du code source en langage FORTH...

Une stratégie consiste à créer un fichier source sur votre ordinateur à l'aide d'un éditeur de texte. Il suffira ensuite d'effectuer un copié/collé par ligne de votre code source vers les fichiers de blocs.

Voici les commandes essentielles pour gérer le contenu d'un bloc:

- **wipe** vide le contenu du bloc courant
- **d** efface la ligne n. Le numéro de ligne doit être dans l'intervalle 0..14. Les lignes qui suivent remontent vers le haut. Exemple: 3 D efface le contenu de la ligne 3 et fait remonter le contenu des lignes 4 à 15.
- **e** efface le contenu de la ligne n. Le numéro de ligne doit être dans l'intervalle 0..15. Les autres lignes ne remontent pas.
- **a** insère une ligne n. Le numéro de ligne doit être dans l'intervalle 0..14. Les lignes situées après la ligne insérées redescendent. Exemple: 3 A test insère test à la ligne 3 et fait descendre le contenu des lignes 4 à 15.
- **r** remplace le contenu de la ligne n. Exemple: 3 R test remplace le contenu de la ligne 3 par test

```
Block 0
| 0
create sintab \ 0...90 Grad, Index in Grad          | 1
0000 , 0175 , 0349 , 0523 , 0698 ,                  | 2
0872 , 1045 , 1219 , 1392 , 1564 ,                  | 3
1736 , 1908 , 2079 , 2250 , 2419 ,                  | 4
2588 , 2756 , 2924 , 3090 , 3256 ,                  | 5
3420 , 3584 , 3746 , 3907 , 4067 ,                  | 6
4226 , 4384 , 4540 , 4695 , 4848 ,                  | 7
5000 , 5150 , 5299 , 5446 , 5592 ,                  | 8
5736 , 5878 , 6018 , 6157 , 6293 ,                  | 9
| 10
| 11
| 12
| 13
| 14
| 15
ok
--> 10 R 6428 , 6561 , 6691 , 6820 , 6947 ,
CTRL-A Z for help | 115200 8N1 | NOR | Minicom 2.7.1 | VT102 | Déconr
```

Voici notre bloc 0 en cours d'édition:

En bas d'écran, la ligne **10 R 6428 , 6561 ,** est en cours d'intégration dans notre bloc à la ligne 10.

Vous remarquez que la ligne 0 n'a pas de contenu. Ceci génère une erreur à la compilation du code FORTH. Pour y remédier, on tape simplement **0 R** suivi de deux espaces.

Avec un peu d'entraînement, en quelques minutes, vous aurez inséré votre code FORTH dans ce bloc.

Procédez de même pour les blocs suivants si nécessaires. Au moment de passer au bloc suivant, vous forcez la sauvegarde du contenu des blocs en tapant **flush**.

Compilation du contenu des blocs

Avant de compiler le contenu d'un fichier de blocs, nous allons vérifier que leur contenu est bien sauvegardé. Pour cela:

- tapez **flush**, puis débranchez la carte ESP32;
- attendez quelques secondes et rebranchez la carte ESP32;
- tapez **editor** et **1**. Vous devez retrouver votre bloc 0 avec le contenu que vous avez édité.

Pour compiler le contenu de vos blocs, vous disposez de deux mots:

- **load** précédé du numéro du bloc dont on veut exécuter et/ou compiler le contenu. Pour compiler le contenu de notre bloc 0, on exécutera **0 load**;
- **thru** précédé de deux numéros de blocs va exécuter et/ou compiler le contenu des blocs comme si on exécute une succession de mots **load**. Exemple: **0 2 thru** exécute et/ou compile le contenu des blocs 0 à 2.

La vitesse d'exécution et/ou de compilation du contenu des blocs est quasi instantanée.

Exemple pratique pas à pas

Nous allons voir, avec un exemple pratique, comment insérer un code source dans le bloc 1. On reprend un code prêt à être intégré dans notre bloc:

```
1 list
editor
0 r \ tools for REGISTERS definitions and manipulations
1 r : mclr { mask addr -- }    addr @ mask invert and addr ! ;
2 r : mset { mask addr -- }    addr @ mask or addr ! ;
3 r : mtst { mask addr -- x }  addr @ mask and ;
4 r : defREG: \ define a register, similar as constant
5 r      create ( addr1 -- <name> ) ,
6 r      does> ( -- regAddr )    @ ;
7 r : .reg ( reg -- ) \ display reg content
8 r      base @ >r binary @ <#
9 r      4 for aft 8 for aft # then next
10 r      bl hold then next #>
11 r      cr space ." 33222222 22221111 11111100 00000000"
12 r      cr space ." 10987654 32109876 54321098 76543210"
```

```
13 r      cr type  r> base !  ;
14 r : defMASK:  create ( mask0 position -- )    lshift ,
15 r      does> ( -- mask1 )                @  ;
save-buffers
```

Procédez simplement par des copié/collé partiels du code ci-dessus et exécutez ce code par ESP32 Forth:

- **1 list** pour sélectionner et voir ce que contient le bloc 1
- **editor** pour sélectionner le vocabulaire **editor**
- copiez les lignes **n r....** par paquets de trois et faites-les exécuter
- **save-buffers** sauvegarde en dur le code dans le fichier de bloc

Eteignez la carte ESP32. Redémarrez-là. Si vous tapez **1 list** vous devez voir le code édité et sauvegardé.

Pour compiler ce code, tapez simplement **1 load**.

Conclusion

L'espace de fichiers disponible pour ESP32forth est proche de 1,8Mo. Vous pouvez donc gérer sans souci des centaines de blocs pour les fichiers sources en langage FORTH. Il est conseillé d'installer des codes sources de parties de codes stables. Ainsi, lors de la phase de mise au point de programmes, il sera bien plus aisé d'intégrer à votre code en phase de mise au point:

```
2 5 thru \ integrate pwm commands for motors
```

au lieu de recharger systématiquement ce code par ligne série ou WiFi.

L'autre intérêt des blocs est de permettre l'embarquement in situ de paramètres, tables de données, etc... utilisables ensuite par vos programmes.

Edition des fichiers sources avec VISUAL Editor

Editer un fichier source FORTH

Pour éditer un fichier source FORTH avec ESP32forth, on va utiliser l'éditeur visual.

Pour éditer un fichier **dump.fs**, procéder comme ceci depuis le terminal connecté à une carte ESP32 contenant ESP32forth:

```
visual edit /spiffs/dump.fs
```

Le code complet de **DUMP** est disponible ici:

<https://github.com/MPETREMANN11/ESP32forth/blob/main/tools/dumpTool.txt>

Le mot **edit** est suivi du répertoire de stockage des fichiers source:

- si le fichier n'existe pas, il est créé;
- si le fichier existe, il est récupéré dans l'éditeur.

Notez bien le nom du fichier que vous avez créé.

Choisissez comme extension de fichier **fs**, pour **F**orth **S**ource.

Edition du code FORTH

Dans l'éditeur, déplacez le curseur avec les flèches gauche-droite-haut-bas disponible au clavier.



Le terminal rafraîchit l'affichage à chaque déplacement du curseur ou modification du code source.

Pour quitter l'éditeur:

- CTRL-S: enregistre le contenu du fichier en cours d'édition
- CTRL-X: quitte l'édition:
 - N: sans enregistrement des modifications du fichier
 - Y: avec enregistrement des modifications

Compilation du contenu des fichiers

La compilation du contenu de notre fichier **dump.fs** s'exécute ainsi:

```
include /spiffs/dump.fs
```

La compilation est beaucoup plus rapide que par l'intermédiaire du terminal.

Les fichiers sources embarqués dans la carte ESP32 avec ESP32forth sont persistants.

Après mise hors tension et rebranchement de la carte ESP32, le fichier sauvegardé reste disponible immédiatement.

On peut définir autant de fichiers que nécessaire.

Il est donc facile d'intégrer dans la carte ESP32 une collection d'outils et de routines dans lesquelles on viendra piocher selon besoins.

Le système de fichiers SPIFFS

ESP32Forth contient un système rudimentaire de fichiers sur mémoire Flash interne. Les fichiers sont accessibles via une interface série dénommée SPIFFS pour Serial Peripheral Interface Flash File System.

Même si le système de fichiers SPIFFS est simple, il permet d'accroître considérablement la souplesse de vos développements avec ESP32Forth:

- gérer des fichiers de configuration
- intégrer des extensions logicielles accessibles sur demande
- modulariser les développements en modules fonctionnels réutilisables

Et bien d'autres usages que nous vous laisserons découvrir...

Accès au système de fichiers SPIFFS

Pour compiler le contenu d'un fichier source édité par visual edit, taper:

```
include /spiffs/dumpTool.fs
```

Le mot **include** doit toujours être utilisé depuis le terminal.

Pour voir la liste des fichiers SPIFFS, utilisez le mot **ls**:

```
ls /spiffs/  
\ affiche:  
\ dumpTool.fs
```

Ici, c'est le fichier **dumpTool.fs** qui a été sauvegardé. Pour SPIFFS, les extensions de fichier n'ont aucune importance. Les noms de fichiers ne doivent pas contenir de caractère espace, ni le caractère /.

Editons et sauvegardons un nouveau fichier **myApp.fs** avec **visual editor**.

Réexécutons **ls**:

```
ls /spiffs/  
\ display:  
\ dumpTool.fs  
\ myApp.fs
```

Le système de fichiers SPIFFS ne gère pas les sous-dossiers comme sur un ordinateur sous Linux. Pour créer un pseudo répertoire, il suffit de l'indiquer au moment de créer un nouveau fichier. Par exemple, éditons le fichier **other/myTest.fs**. Une fois édité et sauvegardé, exécutons **ls**:

```
ls /spiffs/
```

```
\ affiche:
\ dumpTool.fs
\ myApp.fs
\ other/myTest.fs
```

Si on veut visualiser seulement les fichiers de ce pseudo répertoire **other**, il faut faire suivre **/spiffs/** du nom de ce pseudo répertoire:

```
ls /spiffs/other
\ affiche:
\ myTest.fs
```

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

Manipulation des fichiers

Pour effacer intégralement un fichier, utiliser le mot **rm** suivi du nom de fichier à supprimer:

```
rm /spiffs/other/myTest.fs
ls /spiffs/
\ affiche:
\ dumpTool.fs
\ myApp.fs
```

Pour renommer un fichier, utilisez le mot **mv**:

```
mv /spiffs/myApp.fs /spiffs/main.fs
ls /spiffs/
\ affiche:
\ dumpTool.fs
\ main.fs
```

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

```
cp /spiffs/main.fs /spiffs/mainTest.fs
ls /spiffs/
\ affiche:
\ dumpTool.fs
\ main.fs
\ mainTest.fs
```

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

```
cat /spiffs/dumpTool.fs
\ affiche contenu de dumpTool.fs
```

Pour enregistrer le contenu d'une chaîne dans un fichier, agir en deux phases:

- créer un nouveau fichier avec **touch**
- enregistrer le contenu de la chaîne avec **dump-file**

```
touch /spiffs/mTest,fs \ crée nouveau fichier mtest,fs
ls /spiffs/           \ affiche:
\ dumpTool.fs
\ main.fs
\ mainTest.fs
\ mTests

\ enregistre chaîne "Insère mon texte dans mTest" dans mTest
r| ." Insère mon texte dans mTest" | s" /spiffs/mTest" dump-file

include /spiffs/mTest \ affiche: Insert my text in mTest
```

Organiser et compiler ses fichiers sur la carte ESP32

Nous allons voir comment gérer des fichiers pour une application en cours de mise au point sur une carte ESP32 avec ESP32forth installé dessus.

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

Les explications qui suivent ne sont données qu'à titre de conseils. Ils sont issus d'une certaine expérience et ont pour but de faciliter le développement de grosses applications avec ESP32forth.

Edition et transmission des fichiers source

Tous les fichiers sources de votre projet sont sur votre ordinateur. Il est conseillé d'avoir un sous-dossier dédié à ce projet. Par exemple, vous travaillez sur un afficheur OLED SSD1306. Vous créez donc un répertoire nommé SSD1306.

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

L'édition des fichiers sur ordinateur est réalisée avec n'importe quel éditeur de fichiers texte.

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

Ces fichiers sources seront ensuite copiés ou transférés sur la carte ESP32 via la liaison série et un programme de type terminal:

- par copié/collé en utilisant visual sur ESP32forth, à réserver pour les fichiers de petite taille;

- avec une procédure particulière qui sera détaillée plus loin pour les fichiers importants.

Organiser ses fichiers

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

Partons de notre répertoire SSD1306 sur notre ordinateur.

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

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

```
\ OLED SSD1306 128x32 dev et tests affichage
s" /SPIFFS/config.fs" included
```

En phase de développement, le contenu de ce fichier **main.fs** sera chargé manuellement en exécutant **include** comme ceci:

```
include /spiffs/main.fs
```

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

```
\
*****
*
\ Configuration pour affichage OLED SSD1306 128x32
\
*****
*

\ pour SSD1306_128_32
  128 constant SSD1306_LCDWIDTH
  32 constant SSD1306_LCDHEIGHT
```

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

Notre prochain fichier sera **SSD10306commands.fs**. Voici comment charger son contenu depuis main.fs:

```
\ OLED SSD1306 128x32 dev et tests affichage
s" /spiffs/config.fs" included
s" /spiffs/SSD10306commands.fs" included
```

Le contenu du fichier **SSD10306commands.fs** fait près de 230 lignes de code. Il est exclu de copier ligne à ligne dans l'éditeur visual de ESP32forth le contenu de ce fichier. voici une méthode pour copier et enregistrer sur ESP32forth en une seule fois le contenu de ce gros fichier.

Transférer un gros fichier vers ESP32forth

Pour permettre cette transmission de gros fichier, compilez ce code dans ESP32forth:

```
: noType
  2drop ;

visual
: xEdit
  ['] noType is type
  edit
  ['] default-type is type
;
```

Ce code Forth très court permet de transférer un programme FORTH très long depuis l'éditeur sur PC vers un fichier dans la carte ESP32 :

- compiler avec ESP32forth, le code FORTH, ici les mots noType et xEdit
- ouvrir sur votre PC le programme à transférer dans un fichier sur la carte ESP32
- ajoutez en haut du programme la ligne permettant ce transfert rapide **xEdit** **/spiffs/SSD10306commands.fs**
- copier tout le code de votre programme édité sur votre PC,
- passer dans le terminal connecté à ESP32forth
- copiez votre code

Si tout se passe bien, rien ne devrait apparaître sur l'écran du terminal. Attendez quelques secondes.

Ensuite, tapez : CTRL-X, puis appuyez sur "Y"

Vous devriez reprendre le contrôle.

Vérifiez la présence de votre nouveau fichier : **ls /spiffs/**

Vous pouvez maintenant compiler le contenu de votre nouveau fichier.

Conclusion

Les fichiers enregistrés dans le système de fichiers SPIFFS de ESP32forth sont disponibles de manière permanente.

Si vous mettez la carte ESP32 hors service, puis la rebranchez, les fichiers seront disponibles immédiatement.

Le contenu des fichiers est modifiable in situ avec **visual edit**.

Cette commodité rendra les développements beaucoup plus rapides et faciles.

Index lexical

allot.....	25	effacer fichier.....	43	rm.....	43
c!.....	23	forget.....	20	SPIFFS.....	42
c@.....	23	include.....	42	value.....	24
constant.....	24	liste fichiers.....	42	variable.....	23
create.....	25	ls.....	42	20
drop.....	22	mémoire.....	23	20
dup.....	22	pile de retour.....	22	@.....	23

