

Le grand livre de ESP32forth

version 1.5 - 22 octobre 2023



Auteur(s)

- Marc PETREMANN petremann@arduino-forth.com

Collaborateur(s)

- Vaclav POSELT
- Thomas SCHREIN

Table des matières

Auteur(s).....	1
Collaborateur(s).....	1
Introduction.....	7
Aide à la traduction.....	8
Découverte de la carte ESP32.....	9
Présentation.....	9
Les points forts.....	9
Les entrées/sorties GPIO sur ESP32.....	10
Périphériques de l'ESP32.....	11
Pourquoi programmer en langage FORTH sur ESP32?.....	13
Préambule.....	13
Limites entre langage et application.....	13
C'est quoi un mot FORTH?.....	14
Un mot c'est une fonction?.....	14
Le langage FORTH comparé au langage C.....	15
Ce que FORTH permet de faire par rapport au langage C.....	16
Mais pourquoi une pile plutôt que des variables?.....	17
Êtes-vous convaincus?.....	17
Existe-t-il des applications professionnelles écrites en FORTH?.....	18
Un vrai FORTH 32 bits avec ESP32Forth.....	20
Les valeurs sur la pile de données.....	20
Les valeurs en mémoire.....	20
Traitement par mots selon taille ou type des données.....	21
Conclusion.....	22
Dictionnaire / Pile / Variables / Constantes.....	24
Étendre le dictionnaire.....	24
Gestion du dictionnaire.....	24
Piles et notation polonaise inversée.....	25
Manipulation de la pile de paramètres.....	26
La pile de retour et ses utilisations.....	26
Utilisation de la mémoire.....	27
Variables.....	27
Constantes.....	28
Valeurs pseudo-constantes.....	28
Outils de base pour l'allocation de mémoire.....	28
Couleurs de texte et position de l'affichage sur terminal.....	30
Codage ANSI des terminaux.....	30
Coloration du texte.....	31
Position de l'affichage.....	32
Les variables locales avec ESP32Forth.....	34
Introduction.....	34
Le faux commentaire de pile.....	34

Action sur les variables locales.....	35
Structures de données pour ESP32forth.....	38
Préambule.....	38
Les tableaux en FORTH.....	38
Tableau de données 32 bits à une dimension.....	38
Mots de définition de tableaux.....	39
Lire et écrire dans un tableau.....	39
Exemple pratique de gestion d'un écran virtuel.....	40
Gestion de structures complexes.....	43
Définition de sprites.....	45
Affichage des nombres et chaînes de caractères.....	48
Changement de base numérique.....	48
Définition de nouveaux formats d'affichage.....	49
Affichage des caractères et chaînes de caractères.....	51
Variables chaînes de caractères.....	53
Code des mots de gestion de variables texte.....	53
Ajout de caractère à une variable alphanumérique.....	56
Les vocabulaires avec ESP32forth.....	57
Liste des vocabulaires.....	57
Les vocabulaires essentiels.....	58
Liste du contenu d'un vocabulaire.....	58
Utilisation des mots d'un vocabulaire.....	58
Chainage des vocabulaires.....	59
Adapter les plaques d'essai à la carte ESP32.....	61
Les plaques d'essai pour ESP32.....	61
Construire une plaque d'essai adaptée à la carte ESP32.....	61
Alimenter la carte ESP32.....	63
Choix de la source d'alimentation.....	63
Alimentation par le connecteur mini-USB.....	63
Alimentation par le pin 5V.....	63
Démarrage automatique d'un programme.....	66
Installer et utiliser le terminal Tera Term sous Windows.....	68
Installer Tera Term.....	68
Paramétrage de Tera Term.....	68
Utilisation de Tera Term.....	70
Compiler du code source en langage Forth.....	71
Gestion des fichiers sources par blocs.....	73
Les blocs.....	73
Ouvrir un fichier de blocs.....	73
Editer le contenu d'un bloc.....	74
Compilation du contenu des blocs.....	75
Exemple pratique pas à pas.....	75
Conclusion.....	76
Edition des fichiers sources avec VISUAL Editor.....	77
Editer un fichier source FORTH.....	77

Edition du code FORTH.....	77
Compilation du contenu des fichiers.....	78
Le système de fichiers SPIFFS.....	79
Accès au système de fichiers SPIFFS.....	79
Manipulation des fichiers.....	80
Organiser et compiler ses fichiers sur la carte ESP32.....	81
Edition et transmission des fichiers source.....	81
Organiser ses fichiers.....	82
Transférer un gros fichier vers ESP32forth.....	83
Conclusion.....	83
Gérer un feu tricolore avec ESP32.....	85
Les ports GPIO sur la carte ESP32.....	85
Montage des LEDs.....	85
Gestion des feux tricolores.....	87
Conclusion.....	88
Les interruptions matérielles avec ESP32forth.....	89
Les interruptions.....	89
Montage d'un bouton poussoir.....	90
Consolidation logicielle de l'interruption.....	91
Informations complémentaires.....	92
Utilisation de l'encodeur rotatif KY-040.....	95
Présentation de l'encodeur.....	95
Montage de l'encodeur sur la plaque d'essai.....	96
Analyse des signaux de l'encodeur.....	97
Programmation de l'encodeur.....	98
Test de l'encodage.....	99
Incrémenter et décrémenter une variable avec l'encodeur.....	99
Clignotement d'une LED par timer.....	102
Débuter en programmation FORTH.....	102
Clignotement par TIMER.....	104
Les interruptions matérielles et logicielles.....	104
Utiliser les mots interval et rerun.....	105
Minuterie pour femme de ménage.....	107
Préambule.....	107
Une solution.....	107
Une minuterie en FORTH pour ESP32Forth.....	108
Gestion du bouton d'allumage lumière.....	109
Conclusion.....	111
Horloge temps réel logicielle.....	112
Le mot MS-TICKS.....	112
Gestion d'une horloge logicielle.....	112
Mesurer le temps d'exécution d'un mot FORTH.....	114
Mesurer la performance des définitions FORTH.....	114
Test de quelques boucles.....	115

Programmer un analyseur d'ensoleillement.....	117
Préambule.....	117
Le panneau solaire miniature.....	117
Récupération d'un panneau solaire miniature.....	117
Mesure de la tension du panneau solaire.....	118
Mesure du courant du panneau solaire.....	119
Abaissement de la tension du panneau solaire.....	119
Programmation de l'analyseur solaire.....	119
Gestion activation et désactivation d'un appareil.....	121
Déclenchement par interruption timer.....	123
Appareils commandés par le capteur d'ensoleillement.....	124
Gestion des sorties N/A (Numériques/Analogiques).....	125
La conversion numérique / analogique.....	125
La conversion N/A avec circuit R2R.....	125
La conversion N/A avec ESP32.....	125
Possibilités de la conversion N/A.....	127
Installation de la librairie OLED pour SSD1306.....	128
L'interface I2C sur ESP32.....	130
Introduction.....	130
Échange maître esclave.....	131
Adressage.....	132
Définition des ports GPIO pour I2C.....	133
Protocoles du bus I2C.....	133
Détection d'un périphérique I2C.....	133
Programmer en assembleur XTENSA.....	135
Préambule.....	135
Compiler l'assembleur XTENSA.....	136
Programmer en assembleur.....	136
Résumé des instructions de base.....	137
Un dés-assembleur en prime.....	138
Premiers pas en assembleur XTENSA.....	140
Préambule.....	140
Invocation de l'assembleur Xtensa.....	140
Xtensa et la pile FORTH.....	140
Ecriture d'une macro instruction Xtensa.....	141
Gestion de la pile FORTH en assembleur Xtensa.....	143
Efficacité des mots écrits en assembleur XTENSA.....	145
Boucles et branchements en assembleur XTENSA.....	147
L'instruction LOOP en assembleur XTENSA.....	147
Gérer une boucle en assembleur XTENSA avec ESP32forth.....	148
Définition de macro-instructions de gestion de boucle.....	148
Utilisation des macros For, et Next,.....	148
Les instructions de branchement en assembleur XTENSA.....	149
Définition de macros de branchement.....	149
Syntaxe des macro instructions de branchement.....	150

Ressources.....	152
En anglais.....	152
En français.....	152
GitHub.....	152

Introduction

Je gère depuis 2019 plusieurs sites web consacrés aux développements en langage FORTH pour les cartes ARDUINO et ESP32, ainsi que la version eForth web :

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

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

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

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

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

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

Aide à la traduction

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

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

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

Pour me contacter : petremann@arduino-forth.com

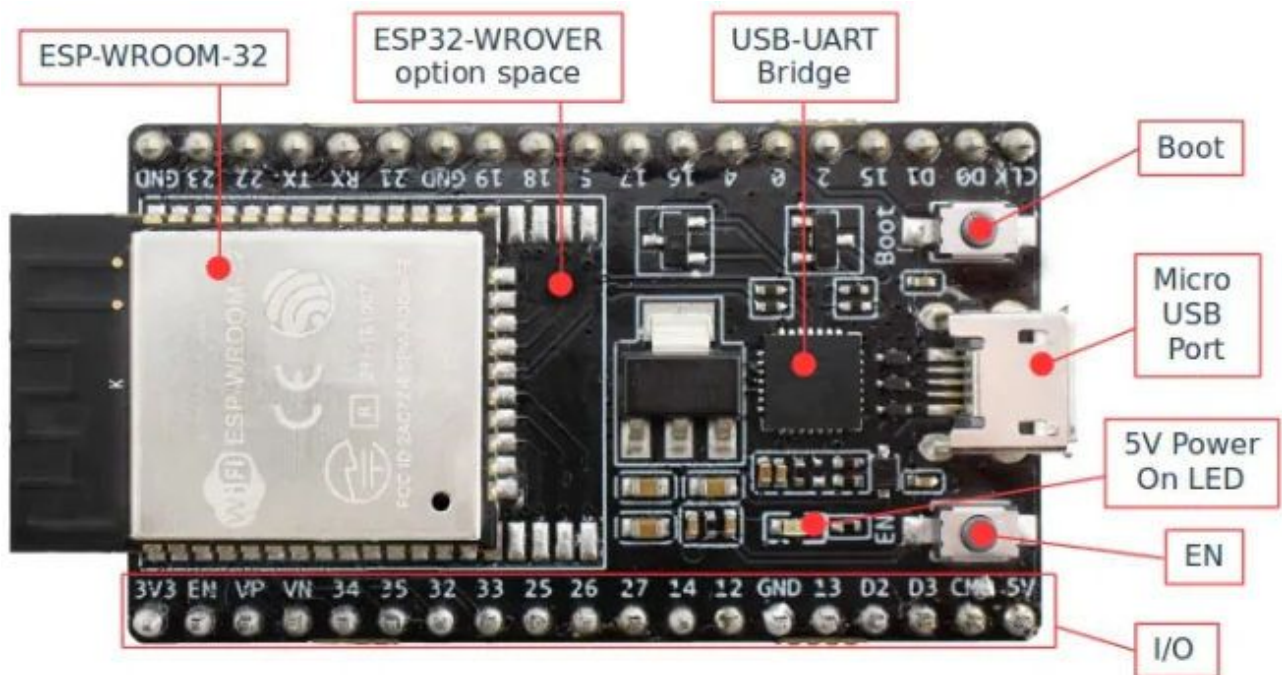
Découverte de la carte ESP32

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.

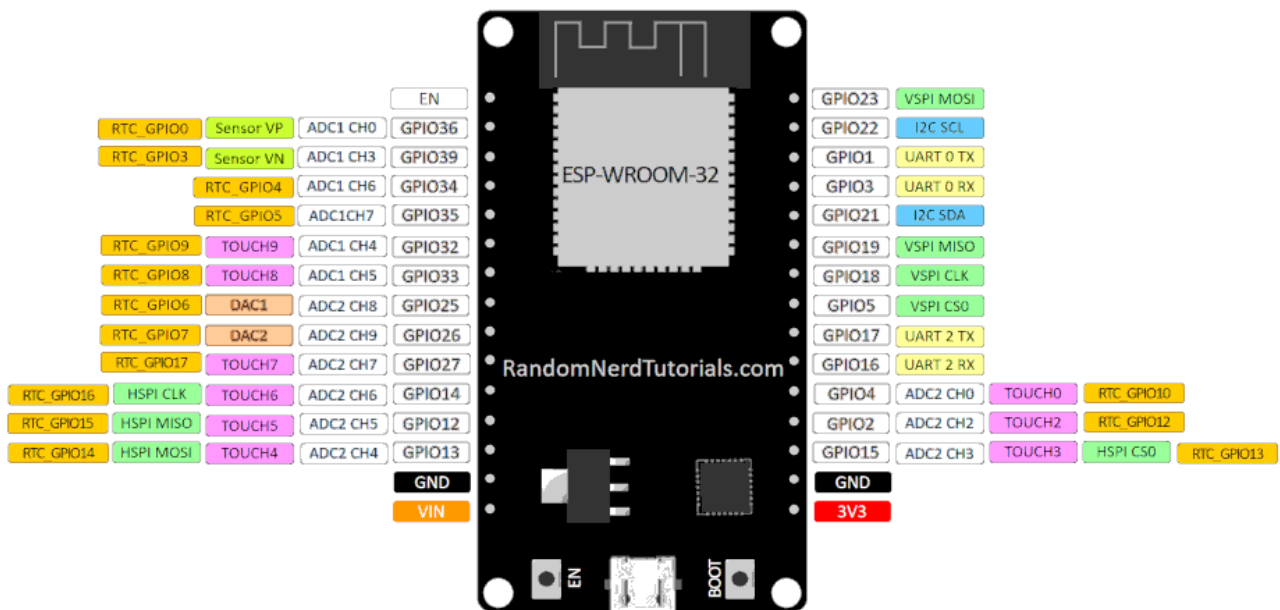
Mais ce qui distingue la carte ESP32, c'est qu'elle est équipée en série d'un support WiFi et Bluetooth, ce que ne proposent les cartes ARDUINO que sous forme d'extensions.

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.

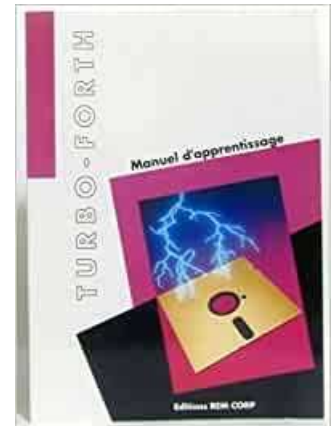
Pourquoi programmer en langage FORTH sur ESP32?

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 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.

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

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

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

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

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

Le langage FORTH comparé au langage C

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

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

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

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

```
var-j @ 13 >              \ Si tous les bits sont recus  
    if
```

```

1 rc5_ok ! \ Le processus de decodage est OK
di        \ Desactiver l'interruption externe (INT0)
exit
then

```

Voici l'initialisation de registres en langage C:

```

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

```

La même définition en langage FORTH:

```

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

```

Ce que FORTH permet de faire par rapport au langage C

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

Êtes-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**:

```
\ Active broche de port, ne changez pas les autres.  
: __/ ( pinmask portadr -- )  
  mset  
  ;  
\ Désactivez 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 !

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'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 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:

```
: :##
  # 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 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 :

```
: 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....

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
```

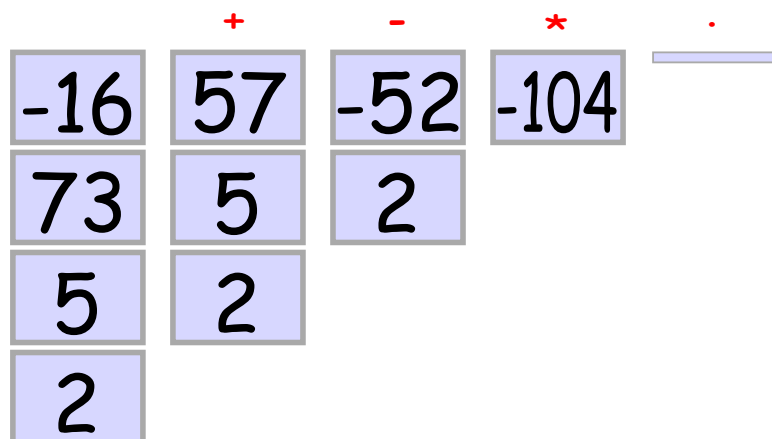
laisse la pile de paramètres dans l'état

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
```

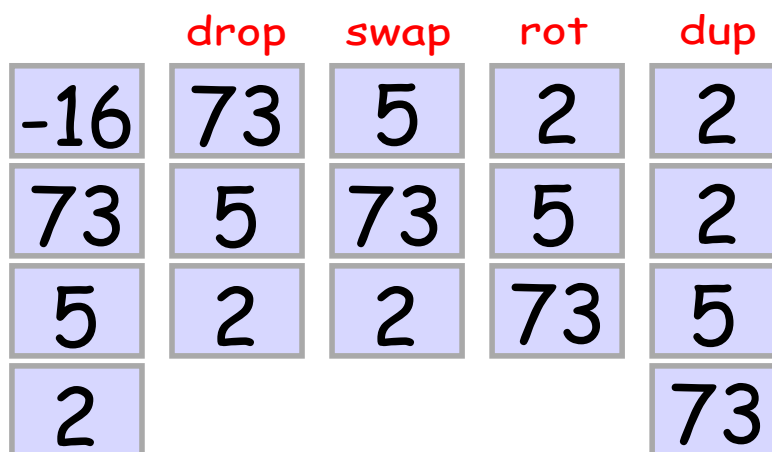

Notez que ESP32forth affiche commodément les éléments de la pile lors de l'interprétation de chaque ligne et que la valeur de -16 est affichée sous la forme d'entier non signé 32 bits. En outre, le mot `.` consomme la valeur de données -104, laissant la pile vide. Si nous exécutons `.` sur la pile maintenant 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 @ . \ affiche: 3
```

Constantes

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

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

\ définit la fréquence du port SPI
4000000 constant SPI_FREQ

\ sélectionne le vocabulaire SPI
only FORTH SPI also

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

Valeurs pseudo-constantes

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

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

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

Outils de base pour l'allocation de mémoire

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

```
create graphic-array ( --- addr )
```

```
%00000000 c,  
%00000010 c,  
%00000100 c,  
%00001000 c,  
%00010000 c,  
%00100000 c,  
%01000000 c,  
%10000000 c,
```

Lorsqu'il est exécuté, le mot **graphic-array** poussera l'adresse de la première entrée.

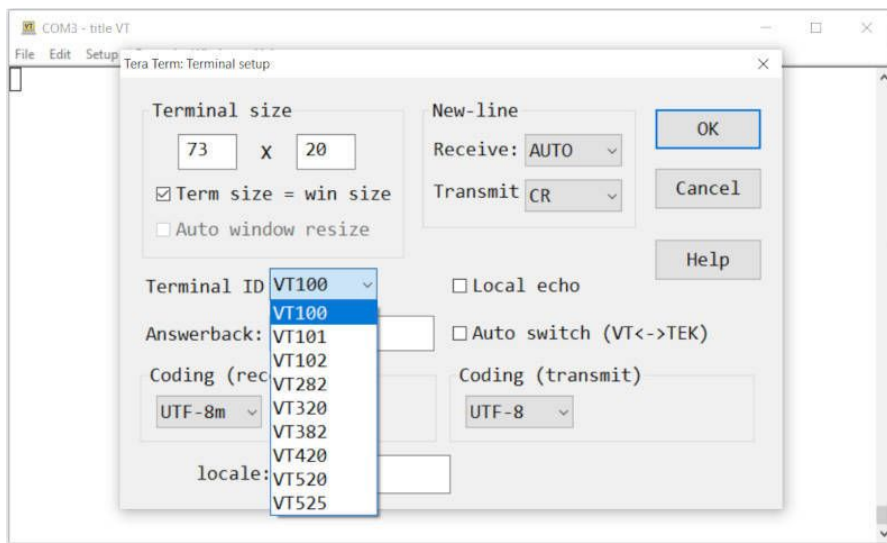
Nous pouvons maintenant accéder à la mémoire allouée à **graphic-array** en utilisant les mots de récupération et de stockage expliqués plus tôt. Pour calculer l'adresse du troisième octet attribué à **graphic-array** on peut écrire **graphic-array 2 +**, en se rappelant que les indices commencent à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

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

Couleurs de texte et position de l'affichage sur terminal

Codage ANSI des terminaux

Si vous utilisez un logiciel de terminal pour communiquer avec ESP32forth, il y a de grandes chances pour que ce terminal émule un terminal de type VT ou équivalent. Ici, TeraTerm paramétré pour émuler un terminal VT100:



Ces terminaux ont deux caractéristiques intéressantes:

- colorer le fond de page et le texte à afficher
- positionner le curseur d'affichage

Ces deux caractéristiques sont contrôlées par des séquences ESC (échappement). Voici comment sont définis les mots **bg** et **fg** dans ESP32forth:

```
forth definitions ansi
: fg ( n -- ) esc ." [38;5;" n. ." m" ;
: bg ( n -- ) esc ." [48;5;" n. ." m" ;
: normal esc ." [0m" ;
: at-xy ( x y -- ) esc ." [" 1+ n. ." ;" 1+ n. ." H" ;
: page esc ." [2J" esc ." [H" ;
```

Le mot **normal** annule les séquences de coloration définies par **bg** et **fg**.

Le mot **page** vide l'écran du terminal et positionne le curseur au coin supérieur gauche de l'écran.

Coloration du texte

Voyons comment colorer d'abord le texte:

```
: testFG ( -- )
  page
  16 0 do
    16 0 do
      j 16 * i + fg
      ." X"
    loop
  cr
loop
normal
;
```

L'exécution de **testFG** donne ceci à l'affichage:



Pour tester les couleurs de fond, on procédera de cette manière:

```
: testBG ( -- )
  page
  16 0 do
    16 0 do
      j 16 * i + bg
      space space
    loop
  cr
loop
normal
```

;

L'exécution de **testBG** donne ceci à l'affichage:



Position de l'affichage

Le terminal est la solution la plus simple pour communiquer avec ESP32forth. Avec les séquences d'échappement ANSI il est facile d'améliorer la présentation des données.

```
09 constant red
11 constant yellow
14 constant cyan
15 constant whyte
: box { x0 y0 xn yn color -- }
  color bg
  yn y0 - 1+ \ determine height
  0 do
    x0 y0 i + at-xy
    xn x0 - spaces
  loop
  normal
;

: 3boxes ( -- )
  page
  2 4 20 6 cyan box
```

```
8 6 28 8 red box
14 8 36 10 yellow box
0 0 at-xy
;
```

L'exécution de **3boxes** affiche ceci:



Vous voici maintenant équipés pour réaliser des interfaces simples et efficaces permettant une interaction avec les définitions FORTH compilées par ESP32forth.

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 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
        @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 ;

```

Structures de données pour ESP32forth

Préambule

ESP32forth est une version 32 bits du langage FORTH. Ceux qui ont pratiqué FORTH depuis ses débuts ont programmé avec des versions 16 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 ESP32forth:

```
cell . \ affiche 4
```

La valeur 4 signifie que la taille des éléments déposés sur la pile de données est de 4 octets, soit $4 \times 8 \text{ bits} = 32 \text{ bits}$.

Avec une version FORTH 16 bits, `cell` empilera la valeur 2. De même, si vous utilisez une version 64 bits, `cell` empilera la valeur 8.

Les tableaux en FORTH

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

Tableau de données 32 bits à une dimension

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

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

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

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

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

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

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

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

Mots de définition de tableaux

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

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

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

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

Avec cette variante, on stocke les mêmes valeurs dans quatre fois moins d'espace mémoire.

Lire et écrire dans un tableau

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

```
arrayC myCTemps
```

```

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

```

Dans notre exemple, le tableau contient 6 éléments. Avec ESP32forth, 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          \ réserve 63 * 16 octets
    mySCREEN SCR_WIDTH SCR_HEIGHT * bl fill \ remplis cet espace
avec 'space'

```

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

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

```

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

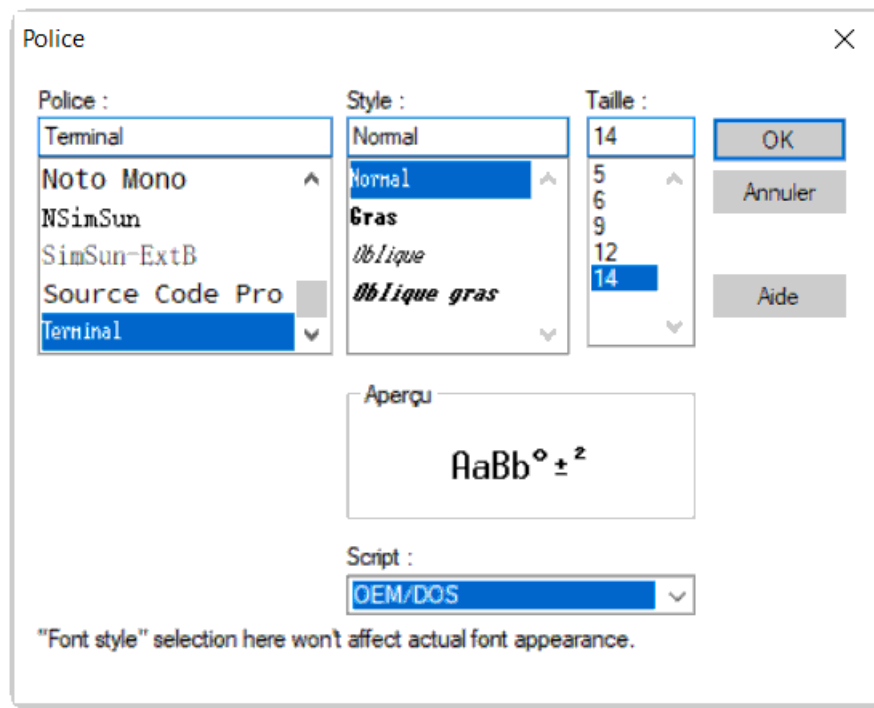
```

Exemple pratique de gestion d'un écran virtuel

Avant d'aller plus loin dans notre exemple de gestion d'un écran virtuel, voyons comment modifier le jeu de caractères du terminal TERA TERM et l'afficher.

Lancer TERA TERM:

- dans la barre de menu, cliquer sur Setup
- sélectionner Font et Font...
- paramétrer la fonte ci-après:



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:

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:

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:

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

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:

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



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 la fenêtre du terminal. Cet affichage est loin d'être performant. Mais il peut être bien plus rapide sur un vrai écran OLED.

Gestion de structures complexes

ESP32forth dispose du vocabulaire structures. Le contenu de ce vocabulaire permet de définir des structures de données complexes.

Voici un exemple trivial de structure:

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

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

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

```
create DateTime
  YMDHMS allot

2022 DateTime >year  !
  03 DateTime >month !
  21 DateTime >day   !
```

```

22 DateTime >hour    !
36 DateTime >min     !
15 DateTime >sec     !

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

DateTime .date

```

On a défini le mot **DateTime** qui est un tableau simple de 6 cellules 32 bits consécutives. L'accès à chacune des cellules est réalisée par l'intermédiaire du pointeur correspondant. On peut redéfinir l'espace alloué de notre structure **YMDHMS** en utilisant le mot **i8** pour pointer des octets:

```

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

create cDateTime
  cYMDHMS allot

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

: .cDate ( date -- )
  >r
  ." YEAR: " r@ >year    @ . cr
  ." MONTH: " r@ >month  c@ . cr

```

```

."    DAY: " r@ >day      c@ . cr
."    HH: " r@ >hour      c@ . cr
."    MM: " r@ >min       c@ . cr
."    SS: " r@ >sec       c@ . cr
r> drop
;
cDateTime .cDate      \ affiche:
\  YEAR: 2022
\  MONTH: 3
\  DAY: 21
\  HH: 22
\  MM: 36
\  SS: 15

```

Dans cette structure cYMDHMS, on a gardé l'année au format 32 bits et réduit toutes les autres valeurs à des entiers 8 bits. On constate, dans le code de .cDate, que l'utilisation des pointeurs permet un accès aisé à chaque élément de notre structure complexe....

Définition de sprites

On a 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
    mySCREEN SCR_WIDTH SCR_HEIGHT * bl fill

```

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

```

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

create myVscreen      \ definit un ecran 8x32 octets
    32 c,              \ compile width
    08 c,              \ compile height
    myVscreen >width  c@
    myVscreen >height c@ * allot

```

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

```

: sprite: ( width height -- )
  create
    swap c, c, \ compile width et height
  does>
;
2 1 sprite: blackChars
  $db c, $db c,
2 1 sprite: greyChars
  $b2 c, $b2 c,
blackChars >content 2 type \ affiche contenu du sprite blackChars

```

Voici comment définir un sprite 5 x 7 octets:

```

5 7 sprite: char3
  $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 sprAddr -- }
  sprAddr >height c@ 0 do
    xpos ypos at-xy
    sprAddr >width c@ i * \ calculate offset in sprite datas
    sprAddr >content + \ calculate real address for line n
in sprite datas
    sprAddr >width c@ 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

```

Résultat de l'affichage de notre sprite:

```
COM3 - Tera Term VT
File Edit Setup Control Window Help
ok
-->
ok
-->
ok
-->
ok
-->
ok
-->
ok
--> blackColor fg
ok
```

A 3x3 grid of colored squares. The first column contains black squares, the second column contains red squares, and the third column contains blue squares. The squares are arranged in a 3x3 grid, with the first row having black, red, and blue squares, the second row having black, red, and blue squares, and the third row having black, red, and blue squares.

J'espère que le contenu de ce chapitre vous aura donné quelques idées intéressantes que vous aimeriez partager...

Affichage des nombres et chaînes de caractères

Changement de base numérique

FORTH ne traite pas n'importe quels nombres. Ceux que vous avez utilisés en essayant les précédents exemples sont des entiers signés simple précision. Le domaine de définition des entiers 32 bits est compris -2147483648 à 2147483647. Exemple :

2147483647 .	\ affiche	2147483647
2147483647 1+ .	\ affiche	-2147483648
-1 u.	\ affiche	4294967295

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.

ESP32forth 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 ESP32forth ;
- **HEX** pour sélectionner la base numérique hexadécimale.

Dès sélection d'une de ces bases numériques, les nombres littéraux seront interprétés, affichés ou traités dans cette base. Tout nombre entré précédemment dans une base numérique différente de la base numérique courante est automatiquement converti dans la base numérique actuelle. Exemple :

DECIMAL	\ base en décimal
255	\ empile 255
HEX	\ sélectionne base hexadécimale
1+	\ incrémente 255 devient 256
.	\ affiche 100

On peut définir sa propre base numérique en définissant le mot approprié ou en stockant cette base dans **BASE**. Exemple :

```
: BINARY ( ---)      \ sélectionne la base numérique binaire
  2 BASE ! ;
DECIMAL 255 BINARY .  \ affiche      11111111
```

Le contenu de **BASE** peut être empilé comme le contenu de n'importe quelle autre variable :

```
VARIABLE RANGE_BASE    \ définition de variable RANGE-BASE
BASE @ RANGE_BASE !     \ stockage contenu BASE dans RANGE-BASE
HEX FF 10 + .           \ affiche 10F
RANGE_BASE @ BASE !     \ restaure BASE avec contenu de RANGE-BASE
```

Dans une définition **:** , le contenu de **BASE** peut transiter par la pile de retour:

```
: OPERATION ( ---)
  BASE @ >R             \ stocke BASE sur pile de retour
  HEX FF 10 + .         \ opération du précédent exemple
  R> BASE ! ;           \ restaure valeur initiale de BASE
```

ATTENTION: les mots **>R** et **R>** ne sont pas exploitables en mode interprété. Vous ne pouvez utiliser ces mots que dans une définition qui sera compilée.

Définition de nouveaux formats d'affichage

Forth dispose de primitives permettant d'adapter l'affichage d'un nombre à un format quelconque. Ces primitives ne traitent que des nombres double précision :

- **<#** 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
35.75 .EUROS	\ affiche	35,75 EUR
1015 3575 + .EUROS	\ affiche	45,90 EUR

Dans la définition de EUROS, le mot **<#** débute la séquence de définition de format d'affichage. Les deux mots **#** placent les chiffres des unités et des dizaines dans la chaîne de caractère. Le mot **HOLD** place le caractère **,** (virgule) à la suite des deux chiffres de droite, le mot **#S** complète le format d'affichage avec les chiffres non nuls à la suite de **,** . Le mot **#>** ferme la définition de format et dépose sur la pile l'adresse et la longueur de la chaîne contenant les digits du nombre à afficher. Le mot **TYPE** affiche cette chaîne de caractères.

En exécution, une séquence de format d'affichage traite exclusivement des nombres entiers 32 bits signés ou non signés. La concaténation des différents éléments de la chaîne se fait de droite à gauche, c'est à dire en commençant par les chiffres les moins significatifs.

Le traitement d'un nombre par une séquence de format d'affichage est exécutée en fonction de la base numérique courante. La base numérique peut être modifiée entre deux digits.

Voici un exemple plus complexe démontrant la compacité du FORTH. Il s'agit d'écrire un programme convertissant un nombre quelconque de secondes au format HH:MM:SS:

```
: :00 ( ---)
  DECIMAL #          \ insertion digit unité en décimal
  6 BASE !           \ sélection base 6
  #                  \ insertion digit dizaine
  [char] : HOLD      \ insertion caractère :
  DECIMAL ;          \ retour base décimale
: HMS ( n ---)       \ affiche nombre secondes format HH:MM:SS
  <# :00 :00 #S #> TYPE SPACE ;
```

Exemples d'exécution:

59 HMS	\ affiche	0:00:59
60 HMS	\ affiche	0:01:00
4500 HMS	\ affiche	1:15:00

Explication : le système d'affichage des secondes et des minutes est appelé système sexagésimal. Les **unités** sont exprimées dans la base numérique décimale, les **dizaines** sont exprimées dans la base six. Le mot **:00** gère la conversion des unités et des dizaines dans ces deux bases pour la mise au format des chiffres correspondants aux secondes et aux minutes. Pour les heures, les chiffres sont tous décimaux.

Autre exemple, soit à définir un programme convertissant un nombre entier simple précision décimal en binaire et l'affichant au format bbbb bbbb bbbb bbbb:

```

: FOUR-DIGITS ( ---)
  # # # # 32 HOLD ;
: AFB ( d ---)          \ format 4 digits and a space
  BASE @ >R              \ Current database backup
  2 BASE !                \ Binary digital base selection
  <#
  4 0 DO                  \ Format Loop
    FOUR-DIGITS
  LOOP
  #> TYPE SPACE          \ Binary display
  R> BASE ! ;            \ Initial digital base restoration

```

Exemple d'exécution :

```

DECIMAL 12 AFB    \ affiche    0000 0000 0000 0110
HEX 3FC5 AFB      \ affiche    0011 1111 1100 0101

```

Encore un exemple, soit à créer un agenda téléphonique où l'on associe à un patronyme un ou plusieurs numéros de téléphone. On définit un mot par patronyme :

```

: .## ( ---)
  # # [char] . HOLD ;
: .TEL ( d ---)
  CR <# .## .## .## .## # # #> TYPE CR ;
: DUGENOU ( ---)
  0618051254 .TEL ;
dugenou \ display : 06.18.05.12.54

```

Cet agenda, qui peut être compilé depuis un fichier source, est facilement modifiable, et bien que les noms ne soient pas classés, la recherche y est extrêmement rapide.

Affichage des caractères et chaînes de caractères

L'affichage d'un caractère est réalisé par le mot **EMIT**:

```

65 EMIT          \ affiche A

```

Les caractères affichables sont compris dans l'intervalle 32..255. Les codes compris entre 0 et 31 seront également affichés, sous réserve de certains caractères exécutés comme des codes de contrôle. Voici une définition affichant tout le jeu de caractères de la table ASCII :

```

variable #out
: #out+! ( n -- )
  #out +!          \ incrémente #out
;
: (.) ( n -- a l )
  DUP ABS <# #S ROT SIGN #>

```

```

;
: .R ( n l -- )
  >R (.) R> OVER - SPACES TYPE
;
: JEU-ASCII ( ---)
  cr 0 #out !
  128 32
  DO
    I 3 .R SPACE      \ affiche code du caractère
    4 #out+!
    I EMIT 2 SPACES    \ affiche caractère
    3 #out+!
    #out @ 77 =
    IF
      CR 0 #out !
    THEN
  LOOP ;

```

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

```

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

```

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

```

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

```

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

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

```

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

```

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

```
LIGNE1 TYPE \ affiche E..Enregistrement de données
```

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

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

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

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

Variables chaînes de caractères

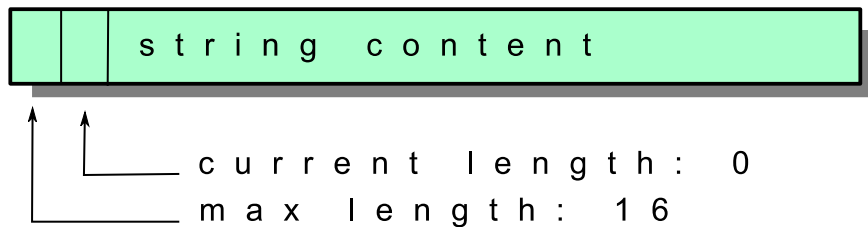
Les variables alpha-numérique texte n'existent pas nativement dans ESP32forth. Voici le premier essai de définition du mot **string** :

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

Une variable chaîne de caractères se définit comme ceci :

```
16 string strState
```

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



Code des mots de gestion de variables texte

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

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

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

\ define a strvar
: string ( n --- names_strvar )
    create
        dup
        ,                \ n is maxlength
        0 ,              \ 0 is real length
        allot
    does>
        cell+ cell+
        dup cell - @
    ;

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

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

\ Example:
```

```

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

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

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

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

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

\ append char c to string
: c+$! ( c str1 -- )
  over >r
  + c!
  r> cell - dup @ 1+ swap !
;

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

```

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

```
64 string myNewString
```

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

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

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

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

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

Ajout de caractère à une variable alphanumérique

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

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

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

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

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

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

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

Les vocabulaires avec 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**.

Les vocabulaires essentiels

Voici la liste des principaux vocabulaires disponibles dans ESP32forth :

- **ansi** gestion de l'affichage dans un terminal ANSI ;
- **editor** donne accès aux commandes d'édition des fichiers de type bloc ;
- **oled** gestion d'afficheurs OLED 128 x 32 ou 128 x 64 pixels. Le contenu de ce vocabulaire n'est disponible qu'après compilation de l'extension **oled.h** ;
- **structures** gestion de structures complexes ;
- @TODO : à compléter

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
```

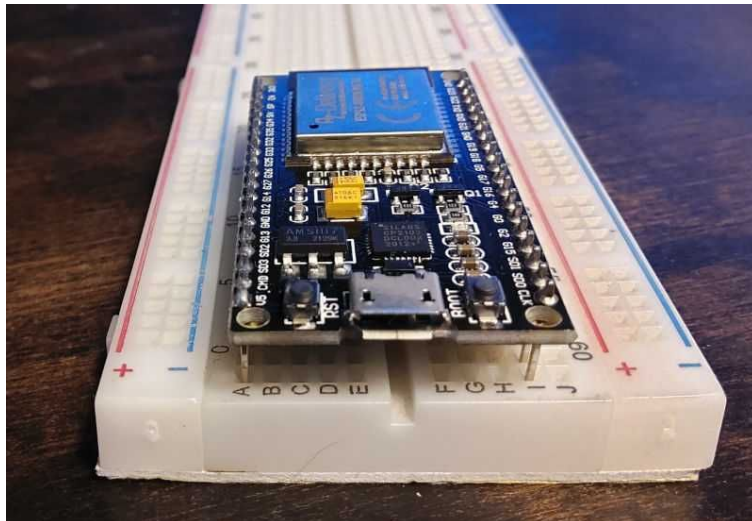
L'ordre de recherche, ici, commencera par le vocabulaire **SPIFFS**, puis **Serial**, puis **ledc** et pour finir le vocabulaire **FORTH**:

- 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:

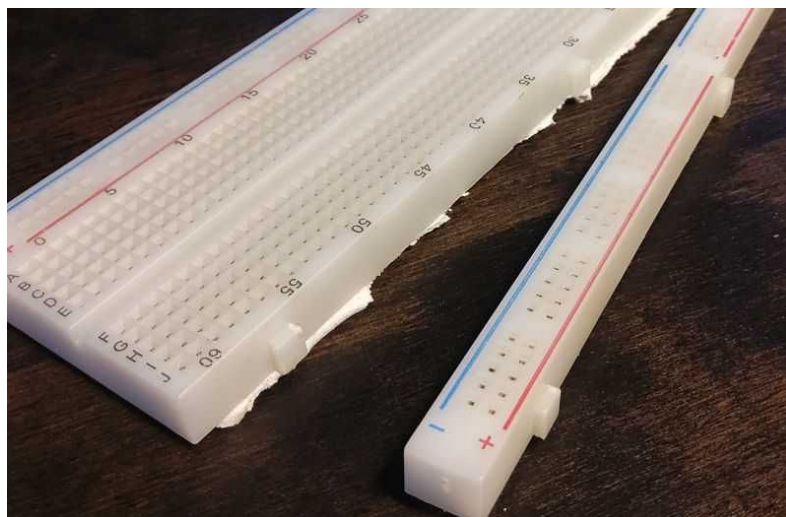


Il n'existe pas de plaque d'essai spécialement adaptée aux cartes ESP32.

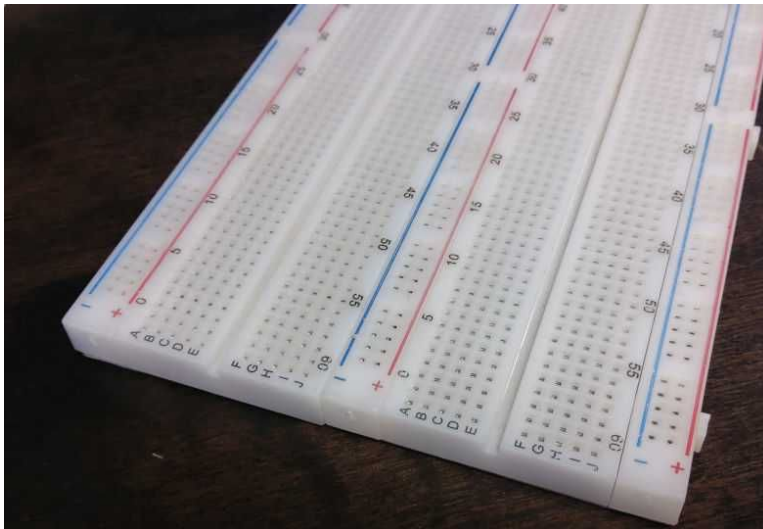
Construire une plaque d'essai adaptée à la carte ESP32

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

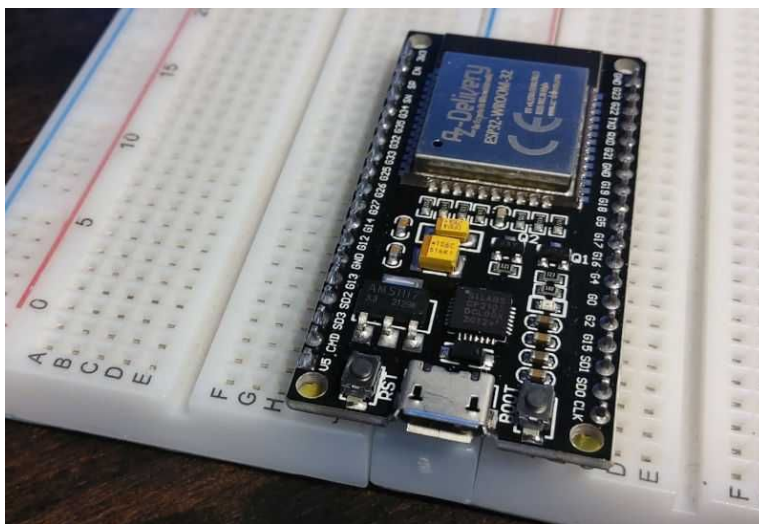
Sur l'une des plaques, on va retirer une ligne d'alimentation. Pour ce faire, utilisez un cutter et découpez par dessous. Vous devez pouvoir séparer cette ligne d'alimentation comme ceci:



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.

Alimenter la carte ESP32

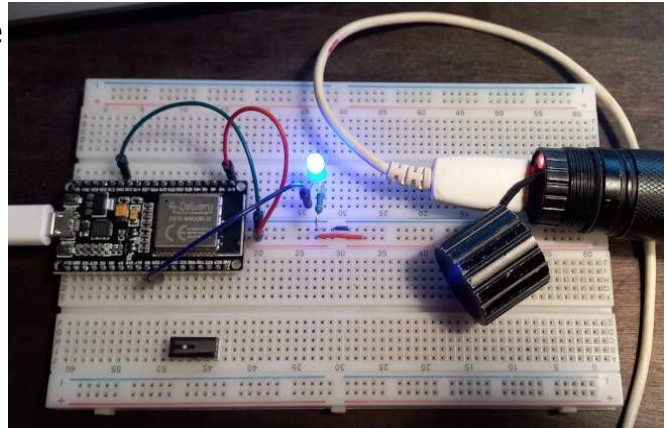
Choix de la source d'alimentation

Nous allons voir ici comment alimenter une carte ESP32. Le but est de donner des solutions pour exécuter les programmes FORTH compilés par ESP32forth.

Alimentation par le connecteur mini-USB

C'est la solution la plus simple. On remplace l'alimentation provenant du PC par une source différente:

- un bloc d'alimentation secteur comme ceux utilisés pour recharger un téléphone mobile;
- une batterie de secours pour téléphone mobile (power bank).



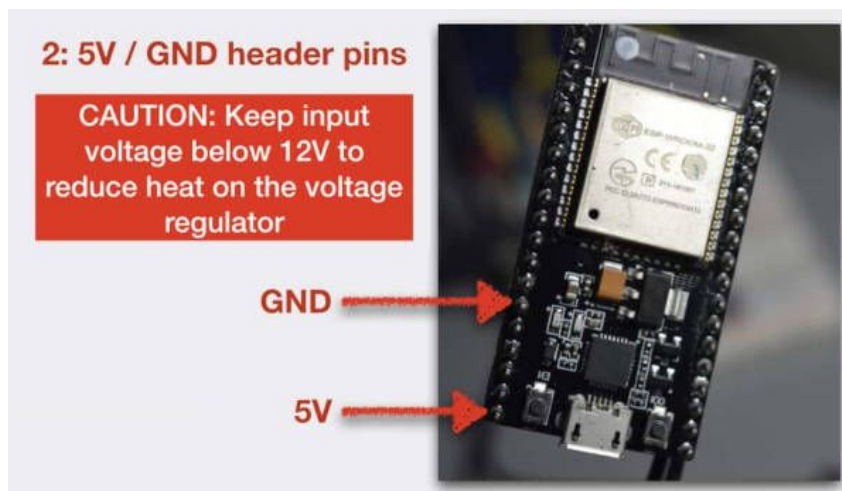
Ici, on alimente notre carte ESP32 avec une batterie de secours pour appareils mobiles.

Alimentation par le pin 5V

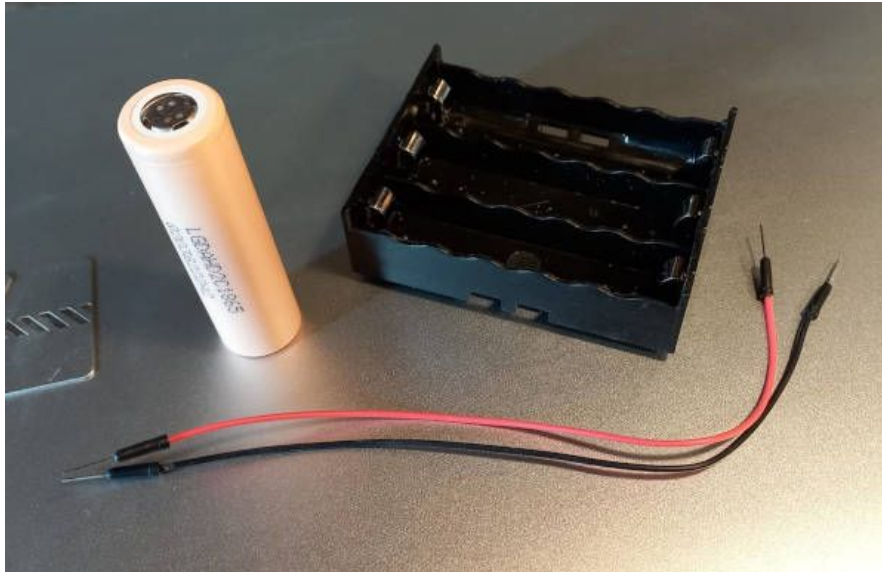
La deuxième option consiste à connecter une alimentation externe non régulée à la broche 5 V et à la masse. Tout ce qui se situe entre 5 et 12 volts devrait fonctionner.

Mais il est préférable de maintenir la tension d'entrée à environ 6 ou 7 Volts pour éviter de perdre trop de puissance sous forme de chaleur sur le régulateur de tension.

Voici les bornes permettant une alimentation externe 5-12V:



Pour exploiter l'alimentation 5V, il faut ce matériel:



- deux batteries lithium 3,7V
- un support batterie
- deux fils dupont

On soude une extrémité de chaque fil dupont aux bornes du support batteries. Ici, notre support accepte trois batteries. Nous n'exploiterons qu'un seul logement à batterie. Les batteries sont montées en série.

Une fois les fils dupont soudés, on installe la batterie et on vérifie que la polarité de sortie est bien respectée:



Maintenant, on peut alimenter notre carte ESP32 par le pin 5V.

ATTENTION: la tension batterie doit être entre 5 à 12 Volts.

Démarrage automatique d'un programme

Comment être certain que la carte ESP32 fonctionne bien une fois alimentée par nos batteries?

La solution la plus simple est d'installer un programme et de paramétrer ce programme pour qu'il démarre automatiquement à la mise sous tension de la carte ESP32. Compilez ce programme:

```
18 constant myLED

0 value LED_STATE

: led.on ( -- )
    HIGH dup myLED pin
    to LED_STATE
;

: led.off ( -- )
    LOW dup myLED pin
    to LED_STATE
;
timers also \ select timers vocabulary

: led.toggle ( -- )
    LED_STATE if
        led.off
    else
        led.on
    then
    0 rerun
;

: led.blink ( -- )
    myLED output pinMode
    ['] led.toggle 500000 0 interval
    led.toggle
;

startup: led.blink
bye
```

Installez une LED sur le pin G18.

Coupez l'alimentation et rebranchez la carte ESP32. Si tout s'est bien passé, la LED doit clignoter au bout de quelques secondes. C'est le signe que le programme s'exécute au démarrage de la carte ESP32.

Débranchez le port USB et branchez la batterie. La carte ESP32 doit démarrer et la LED clignoter.

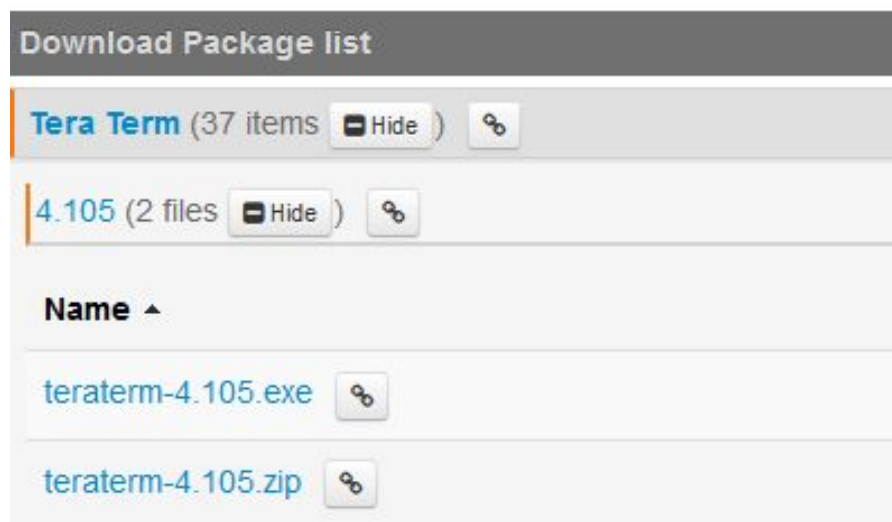
Tout le secret tient dans la séquence **startup: led.blink**. Cette séquence fige le code FORTH compilé par ESP32forth et désigne le mot **led.blink** comme mot à exécuter au démarrage de ESP32forth une fois la carte ESP32 sous tension.

Installer et utiliser le terminal Tera Term sous Windows

Installer Tera Term

La page en anglais pour Tera Term, c'est ici:

<https://ttssh2.osdn.jp/index.html.en>



Allez sur la page téléchargement, récupérez le fichier exe ou zip:

Installez Tera Term. L'installation est simple et rapide.

Paramétrage de Tera Term

Pour communiquer avec la carte ESP32, il faut régler certains paramètres:

- cliquez sur Configuration -> port série

Tera Term: Serial port setup and connection

Port: COM3 New setting

Speed: 115200

Data: 8 bit Cancel

Parity: none Help

Stop bits: 1 bit

Flow control: none

Transmit delay

0 msec/char 0 msec/line

Device Friendly Name: Silicon Labs CP210x USB to UART Bridge
 Device Instance ID: USB\VID_10C4&PID_EA60\0001
 Device Manufacturer: Silicon Labs
 Provider Name: Silicon Laboratories Inc.
 Driver Date: 9-19-2016
 Driver Version: 6.7.4.261

Pour un affichage confortable:

Tera Term: Window setup

Title: Tera Term OK

Cursor shape

☒ Block ☐ Vertical line ☐ Horizontal line

☐ Hide title bar ☐ Hide menu bar

☒ 16 Colors (PC style) ☒ 16 Colors (aixterm style) ☒ 256 Colors (xterm style)

☒ Enable bold font ☒ Scroll buffer: 10000 lines

Color

☒ Text Attribute Normal

☐ Background Reverse

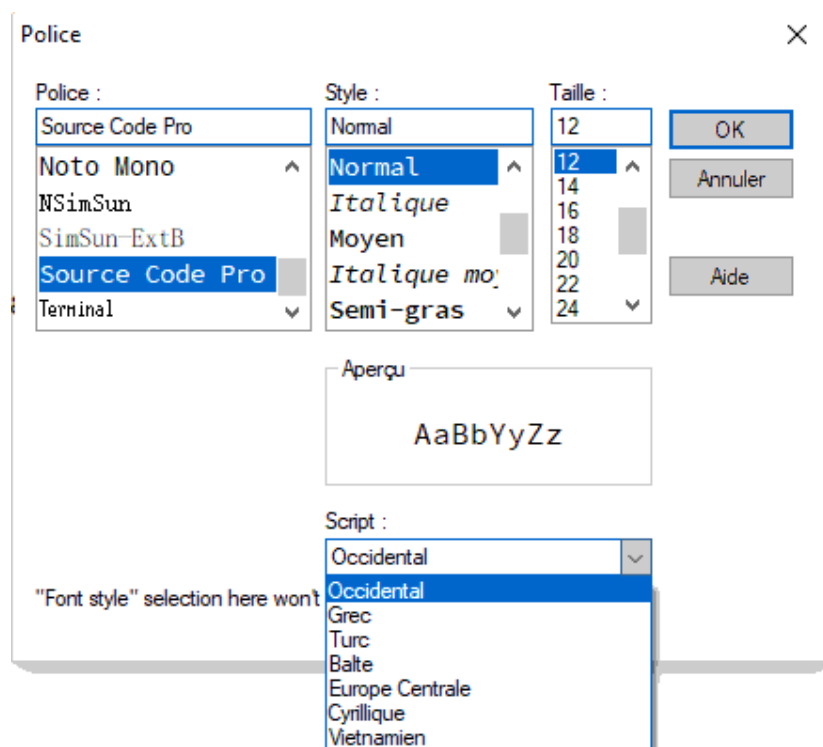
R: 0 < > ABC

G: 0 < >

B: 0 < >

☒ Always use Normal text's BG

Pour des caractères lisibles:



- cliquez sur Configuration -> police

Pour retrouver tous ces réglages au prochain lancement du terminal Tera Term, sauvegardez la configuration:

- cliquez sur *Setup* -> *Save setup*
- acceptez le nom **TERATERM.INI**.

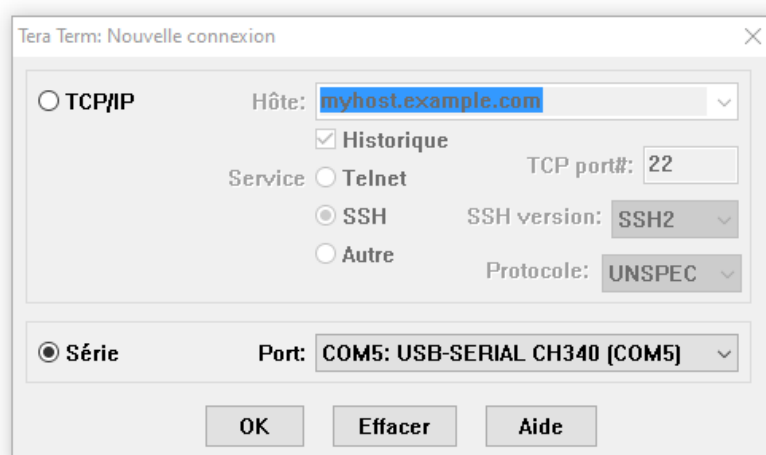
Utilisation de Tera Term

Une fois paramétré, fermez Tera Term.

Connectez votre carte ESP32 à un port USB disponible de votre PC.

Relancez Tera Term, puis cliquez sur *fichier* -> *nouvelle connexion*

Sélectionnez le port série:



Si tout s'est bien passé, vous devez voir ceci:



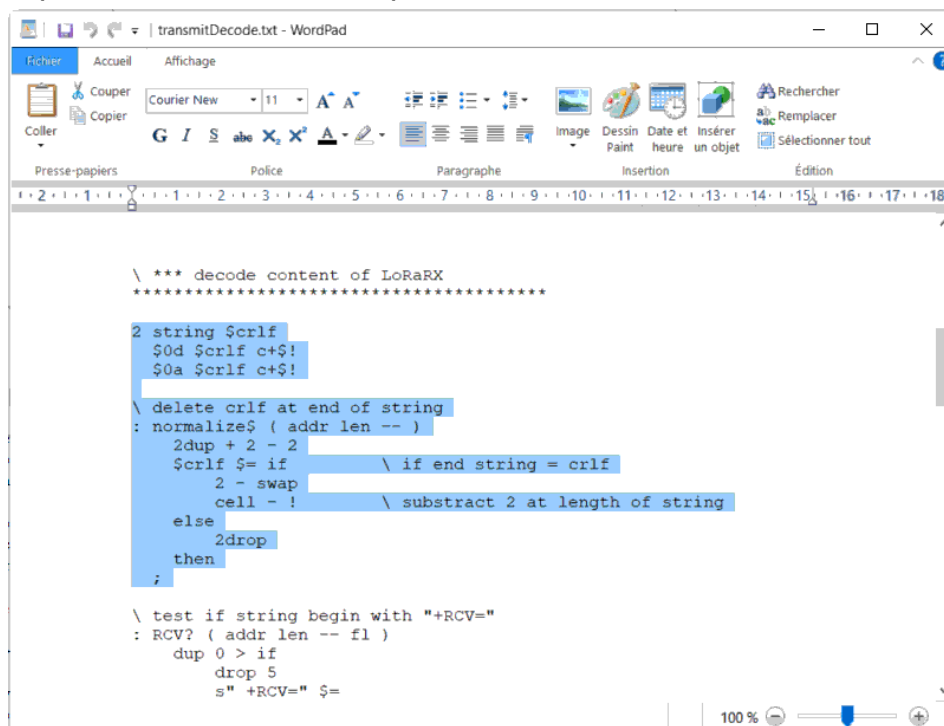
```
COM3 - Tera Term VT
File Edit Setup Control Window Help
ESP32forth v7.0.6.10 - rev 17c8b34289028a5c731d
ok
-->
```

Compiler du code source en langage Forth

Tout d'abord, rappelons que le langage FORTH est sur la carte ESP32! FORTH n'est pas sur votre PC. Donc, on ne peut pas compiler le code source d'un programme en langage FORTH sur le PC.

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

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



```
transmitDecode.txt - WordPad
Fichier Accueil Affichage
Couper Copier
Coller
Police Paragraphe Insertion Édition
Rechercher Remplacer Sélectionner tout

\ *** decode content of LoRaRX
*****

2 string $CrLf
  $0d $CrLf c+$!
  $0a $CrLf c+$!
\ delete crlf at end of string
: normalize$ ( addr len -- )
  2dup + 2 - 2
  $CrLf $= if \ if end string = crlf
    2 - swap
    cell - ! \ subtract 2 at length of string
  else
    2drop
  then
;

\ test if string begin with "+RCV="
: RCV? ( addr len -- f1 )
  dup 0 > if
    drop 5
    s" +RCV=" $=
```

Le code source en langage FORTH peut être composé et édité avec n'importe quel éditeur de texte: bloc notes, PSpad, Wordpad..

Personnellement j'utilise l'IDE Netbeans. Cet IDE permet d'éditer et gérer des codes sources dans de nombreux langages de programmation..

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

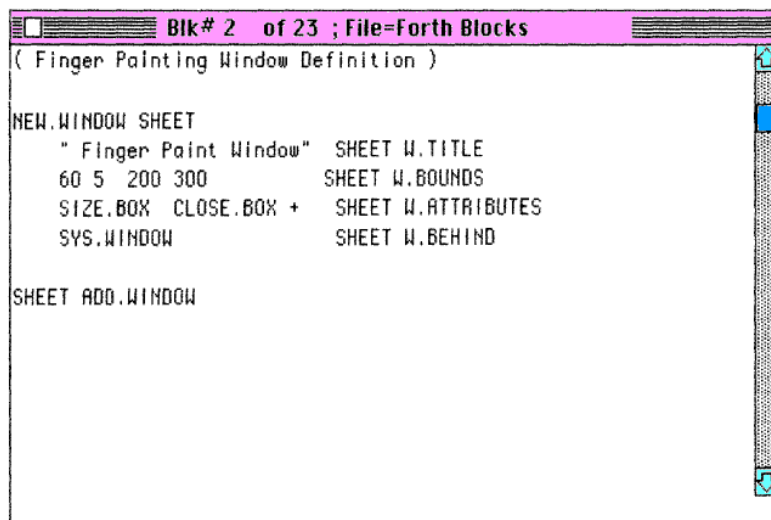
Cliquez sur la fenêtre du terminal Tera Term. Faites Coller:

Il suffit de valider en cliquant sur OK et le code sera interprété et/ou compilé.

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

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'éditeur 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/MPETREMAN11/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.

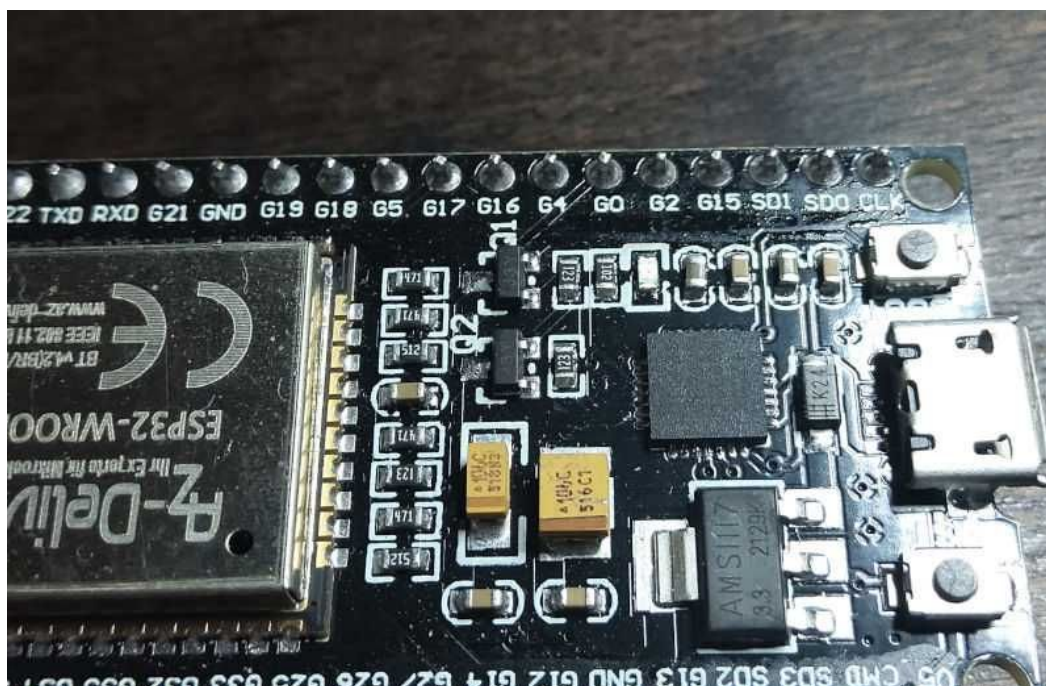
Gérer un feu tricolore avec ESP32

Les ports GPIO sur la carte ESP32

Les ports GPIO (anglais : General Purpose Input/Output, littéralement Entrée-sortie à usage général) sont des ports d'entrées-sorties très utilisés dans le monde des microcontrôleurs

La puce ESP32 est livrée avec 48 broches ayant multiples fonctions. Toutes les broches ne sont pas exploitées sur les cartes de développement ESP32, et certaines broches ne peuvent pas être utilisées.

Il y a de nombreuses questions sur la façon d'utiliser les GPIO ESP32. Quels connecteurs devriez-vous utiliser? Quels connecteurs devriez-vous éviter d'utiliser dans vos projets?



Si on regarde à la loupe une carte ESP32, on voit ceci:

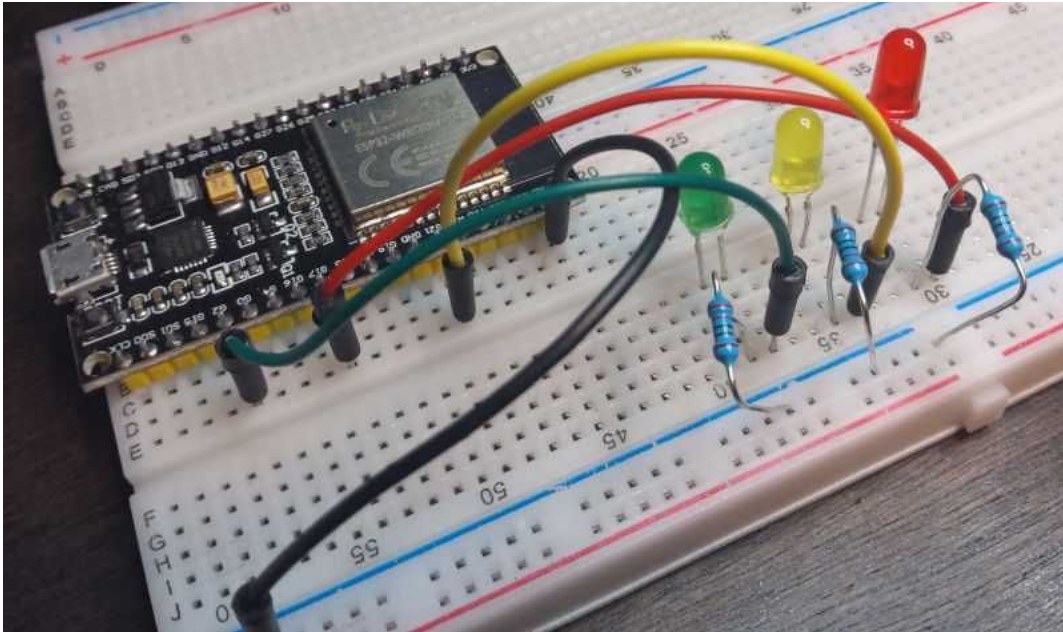
Chaque connecteur est identifié par une série de lettres et chiffres, ici de gauche à droite sur notre photo: G22 TXD RXD G21 GND G19 G18, etc...

Les connecteurs qui nous intéressent pour cette prise en main sont préfixés par la lettre G suivis de un ou deux chiffres. Par exemple, G2 correspond à GPIO 2.

La définition et l'exploitation en mode sortie d'un connecteur GPIO est assez simple.

Montage des LEDs

Le montage est assez simple et une seule photo suffit:



- LED verte connectée à G2 - fil vert
- LED jaune connectée à G21 - fil jaune
- LED rouge connectée à G17 - fil rouge
- fil noir connecté à GND

Notre code utilise le mot **include** suivi du fichier à charger.

On définit nos LEDs avec **defPin:**

```
DEFINED? defPIN: invert
  [if] include /spiffs/defpin.txt [then]
\ Use:
\ numGPIO defPIN: PD7  ( define portD pin #7)

2 defPIN: ledGREEN
21 defPIN: ledYELLOW
17 defPIN: ledRED

: LEDinit
  ledGREEN    output pinMode
  ledYELLOW   output pinMode
  ledRED      output pinMode
;
```

Beaucoup de programmeurs ont la mauvaise habitude de nommer les connecteurs par leur numéro. Exemple :

```
17 defPin: pin17
```

ou

```
17 defPin: GPIO17.
```

Pour être efficace, il faut nommer les connecteurs par leur fonction. Ici on définit les connecteurs **ledRED** ou **ledGREEN**.

Pourquoi? Parce que le jour où vous avez besoin de rajouter des accessoires et libérer par exemple le connecteur G21, il suffit de redéfinir **21 defPIN: ledYELLOW** avec le nouveau numéro de connecteur. Le reste du code sera inchangé et exploitable.

Gestion des feux tricolores

Voici la partie de code qui contrôle nos LEDs dans notre simulation de feu tricolore :

```
\ trafficLights execute one light cycle
: trafficLights ( ---)
    high ledGREEN    pin      3000 ms      low ledGREEN    pin
    high ledYELLOW   pin      800 ms       low ledYELLOW   pin
    high ledRED      pin      3000 ms      low ledRED      pin
    ;

\ classic traffic lights loop
: lightsLoop ( ---)
    LEDinit
    begin
        trafficLights
    key? until
    ;

\ german traffic light style
: Dtraffic ( ---)
    high ledGREEN    pin      3000 ms      low ledGREEN    pin
    high ledYELLOW   pin      800 ms       low ledYELLOW   pin
    high ledRED      pin      3000 ms
    ledYELLOW high    800 ms
    \ simultaneous red and yellow ON
    high ledRED      pin \ simultaneous red and yellow OFF
    high ledYELLOW   pin
    ;

\ german traffic lights loop
: DlightsLoop ( ---)
```

```
LEDinit
begin
    Dtraffic
    key? until
;
```

Conclusion

Ce programme de gestion de feux tricolores aurait parfaitement pu être écrit en langage C. Mais l'avantage du langage FORTH, c'est qu'il donne la main, via le terminal, pour analyser, déboguer et modifier très rapidement des fonctions (en FORTH on dit des mots).

La gestion de feux tricolores est un exercice facile en langage C. Mais quand les programmes deviennent un peu plus complexes, le processus de compilation et téléversement s'avère rapidement fastidieux.

Il suffit d'agir via le terminal et de faire un simple copié/collé de n'importe quel fragment de code en langage FORTH pour qu'il soit compilé et/ou exécuté.

Si vous utilisez un programme terminal pour communiquer avec la carte ESP32, tapez simplement **DlightsLoop** ou **lightsLoop** pour tester le fonctionnement du programme. Ces mots utilisent une boucle conditionnelle. Il suffit de taper sur une touche du clavier pour que le mot cesse de s'exécuter en fin de boucle.

Les interruptions matérielles avec ESP32forth

Les interruptions

Quand on souhaite gérer des événements extérieurs, un bouton poussoir par exemple, nous disposons de deux solutions :

- tester aussi régulièrement que possible l'état du bouton, au travers d'une boucle. On agira en fonction de l'état de ce bouton.
- utiliser une interruption. On affecte le code d'exécution à une interruption rattachée à un pin. Le bouton est connecté à ce pin et la modification d'état va exécuter ce mot.

La solution par interruption est la plus élégante. Elle permet de soulager le programme principal en évitant la surveillance du bouton dans une boucle.

Dans sa documentation ESP32forth, Brad NELSON donne un exemple simple de gestion d'interruption :

```
17 input pinMode
: test ." pinvalue: " 17 digitalRead . cr ;
' test 17 pinchange
```

Sauf que cet exemple, tel qu'il est rédigé, a de fortes chances de ne pas fonctionner. Nous allons voir pourquoi et donner les éléments pour qu'il fonctionne.

Montage d'un bouton poussoir

Le bouton est connecté en entrée à l'alimentation 3,3V de la carte ESP32.

La sortie du bouton poussoir est raccordée au pin GPIO17. C'est tout.

Pour que l'exemple de Brad NELSON soit fonctionnel, il faut sélectionner le vocabulaire **interrupts** avant de paramétrer l'interruption par **pinchange**. Au passage, on définira la constante **button**:

```
17 constant button
button input pinMode
: test ." pinvalue: "
    button digitalRead . cr
;
interrupts
' test button pinchange
forth
```

Ca fonctionne, mais il y a effet inattendu qui provoque le déclenchement intempestif de l'interruption:

```
pinvalue: 0
pinvalue: 1
pinvalue: 1
pinvalue: 1
pinvalue: 0
pinvalue: 0
pinvalue: 0
pinvalue: 1
pinvalue: 1
pinvalue: 1
pinvalue: 0
pinvalue: 0
pinvalue: 1
```

La solution matérielle consisterait à mettre une résistance de forte valeur en sortie du bouton et reliée à GND.

Consolidation logicielle de l'interruption

Dans la carte ESP32, on peut activer une résistance sur n'importe quel pin GPIO. Cette activation est réalisée par le mot `gpio_pulldown_en`. Ce mot accepte comme paramètre le numéro de pin GPIO dont il faut activer la résistance. En retour ce mot renvoie 0 si l'action s'est bien déroulée, un code d'erreur dans le cas contraire :

```
17 constant button
button input pinMode
: test ." pinvalue: "
    button digitalRead . cr
;
interrupts
button gpio_pulldown_en drop
' test button pinchange
forth
```

Le résultat de l'exécution de l'interruption est nettement meilleur :

```
ok    button digitalRead . cr
ok    ;
ok interrupts
ok button gpio_pulldown_en drop
ok ' test button pinchange
ok forth
--> pinvalue: 1
pinvalue: 0
pinvalue: 1
pinvalue: 0
pinvalue: 1
pinvalue: 0
```


A chaque changement d'état, on a une interruption. Sur la copie écran ci-dessus, chaque changement d'état affiche **pinvalue: 1** puis **pinvalue: 0**.

Il est possible de prendre en compte une interruption sur le front montant seul. C'est possible en indiquant :

```
button GPIO_INTR_POSEDGE gpio_set_intr_type drop
```

Le mot **gpio_set_intr_type** accepte ces paramètres :

- **GPIO_INTR_ANYEDGE** pour gérer les interruptions en front montant ou descendant
- **GPIO_INTR_NEGEDGE** pour gérer les interruptions en front descendant seul
- **GPIO_INTR_POSEDGE** pour gérer les interruptions en front montant seul
- **GPIO_INTR_DISABLE** pour désactiver les interruptions

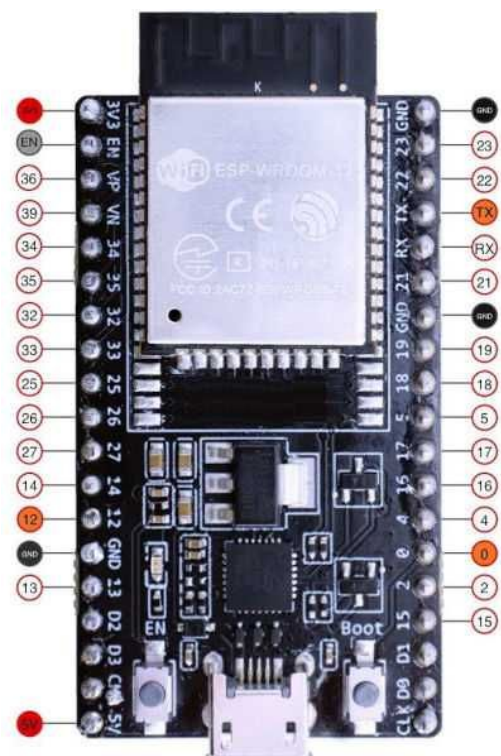
Code FORTH complet avec détection du front montant :

```
17 constant button
0 constant GPIO_PULLUP_ONLY
button input pinMode
: test ." pinvalue: "
    button digitalRead . cr
;
interrupts
button gpio_pulldown_en drop
button GPIO_INTR_POSEDGE gpio_set_intr_type drop
' test button pinchange
forth
```

Informations complémentaires

Pour ESP32, toutes les broches du GPIO peuvent être utilisées en interruption, à l'exception de GPIO6 à GPIO11.

N'utilisez pas les broches colorées en orange ou en rouge. Votre programme pourrait avoir un comportement inattendu en utilisant celles-ci.



Utilisation de l'encodeur rotatif KY-040

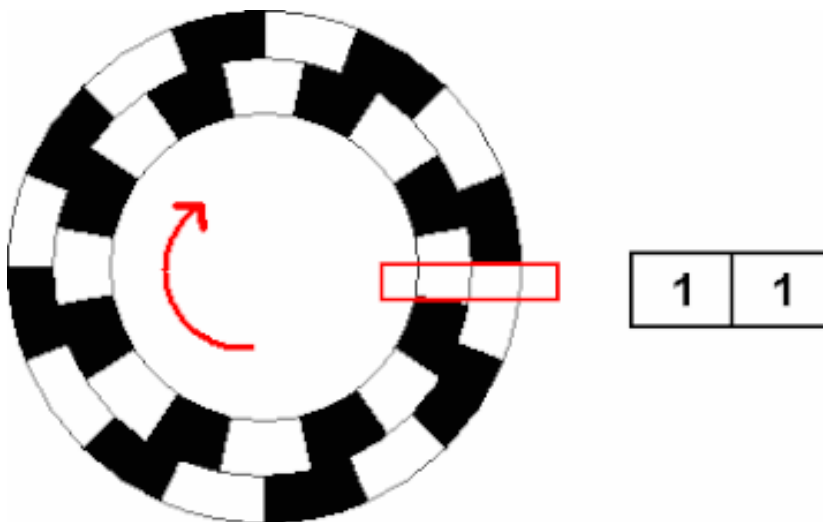
Présentation de l'encodeur

Pour faire varier un signal, nous disposons de plusieurs solutions:

- une résistance variable dans un potentiomètre
- deux boutons gérant par logiciel la variation
- un encodeur rotatif

L'encodeur rotatif est une solution intéressante. On peut le faire agir comme un potentiomètre, avec l'avantage de ne pas avoir de butée de début et fin de course.

Son principe est très simple. Voici les signaux émis par notre encodeur rotatif:



Voici notre encodeur:

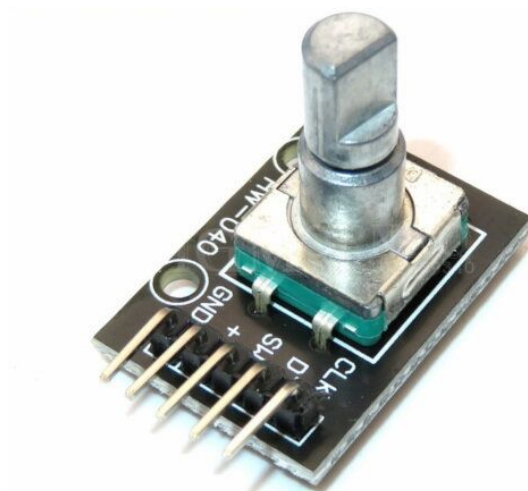
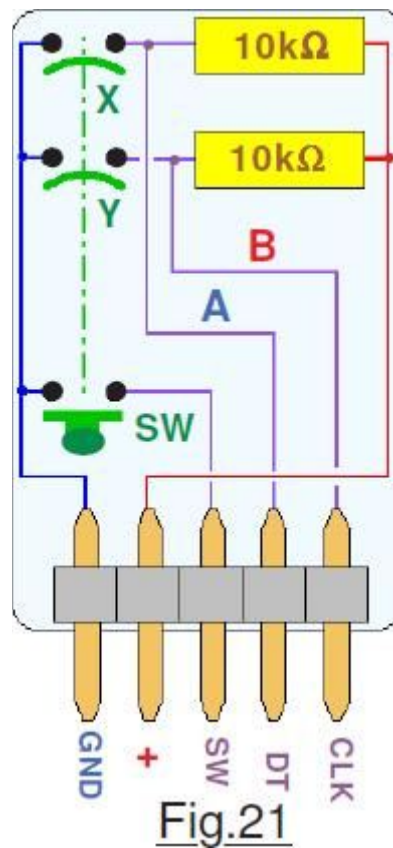


Schéma du fonctionnement interne:



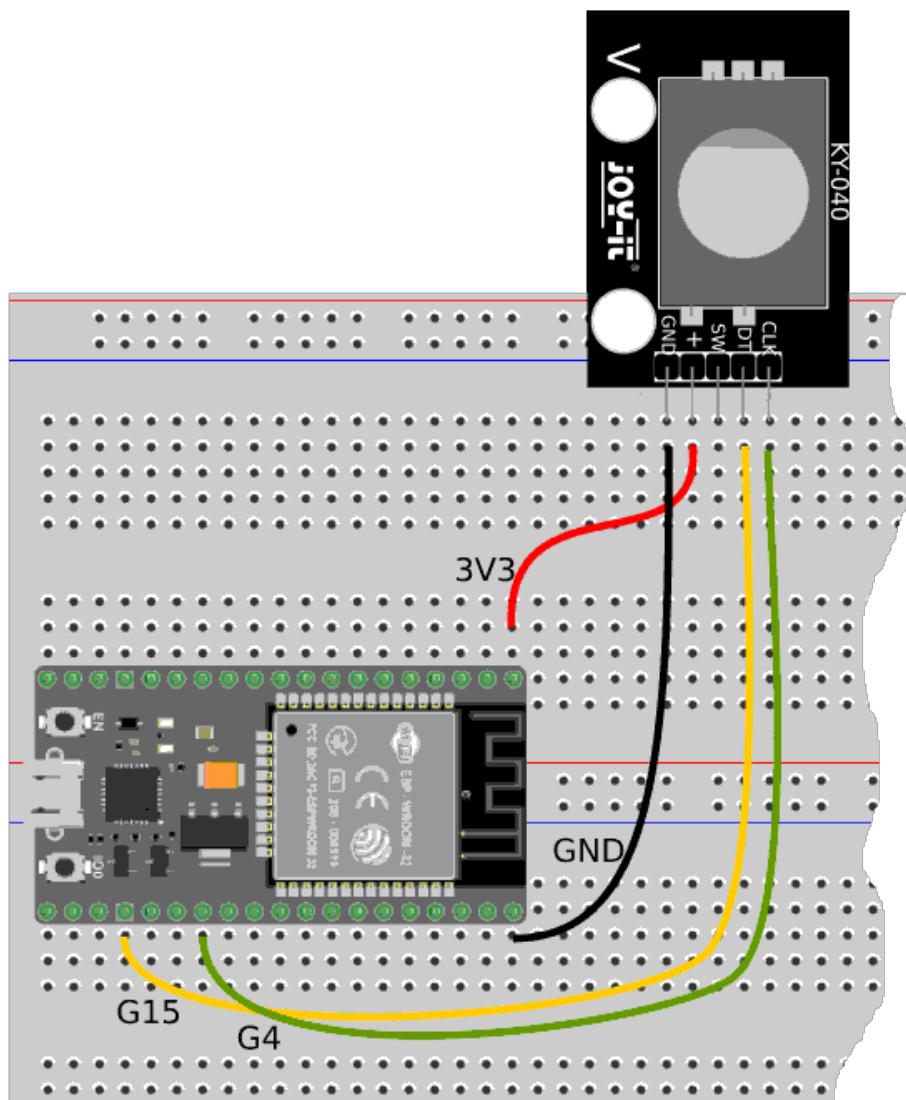
Selon ce schéma, deux bornes nous intéressent:

- A (DT) -> switch X
- B (CLK) -> switch Y

Cet encodeur peut être alimenté en 5V ou 3,3V. Ça nous arrange, car la carte ESP32 dispose d'une sortie 3,3V.

Montage de l'encodeur sur la plaque d'essai

Le câblage de notre encodeur à la carte ESP32 ne nécessite que 4 fils:

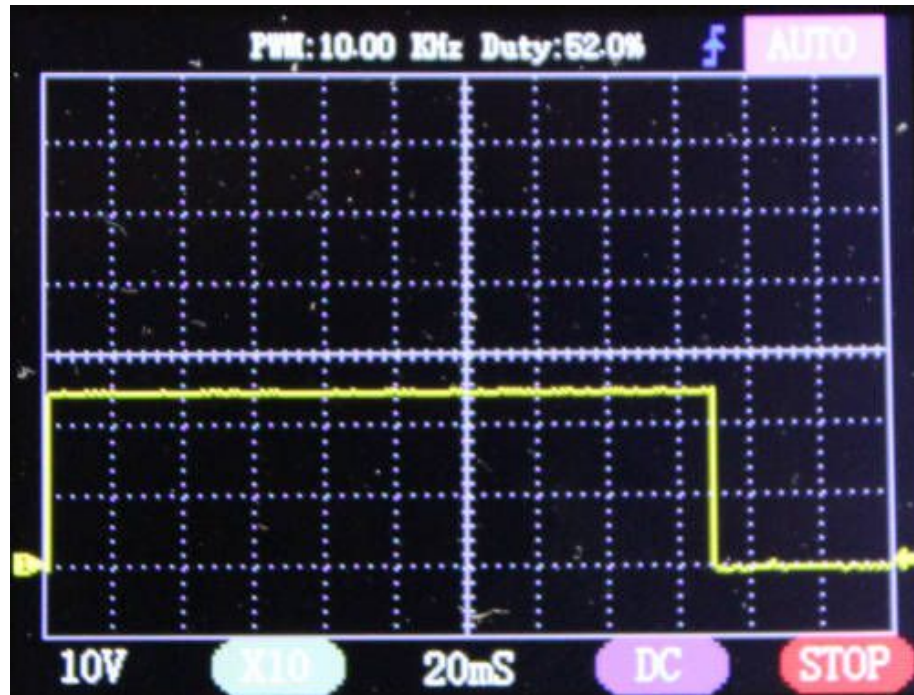


ATTENTION: la position des pins G4 et G15 peut varier selon la version de votre carte ESP32.

Analyse des signaux de l'encodeur

Tel que notre encodeur est connecté, chaque borne A ou B reçoit une tension, ici 3,3V, dont l'intensité est limitée par une résistance de 10Kohms.

L'analyse du signal sur la borne G15 montre bien une présence de la tension 3,3V:



Sur cette capture de signal, le niveau bas sur la borne G15 apparaît quand on manoeuvre la tige de commande de l'encodeur. Au repos, le signal sur la borne G15 est au niveau haut.

Ceci change tout, car, au niveau programmation, on doit traiter l'interruption de G15 en front descendant.

Programmation de l'encodeur

L'encodeur sera géré par interruption. Les interruptions déclenchent le programme seulement si un signal particulier atteint un niveau bien défini.

Nous allons gérer une seule interruption sur la borne GPIO G15:

```
interrupts

\ enable interrupt on GPIO G15
: intG15enable ( -- )
    15 GPIO_INTR_POSEDGE gpio_set_intr_type drop
    ;

\ disable interrupt on GPIO G15
: intG15disable ( -- )
    15 GPIO_INTR_DISABLE gpio_set_intr_type drop
    ;
```

```

: pinsInit ( -- )
  04 input pinmode          \ G04 as an input
  04 gpio_pulldown_en drop  \ Enable pull-down on GPIO 04
  15 input pinmode          \ G15 as an input
  15 gpio_pulldown_en drop  \ Enable pull-down on GPIO 15
  intG15enable
;

```

Dans le mot **pinsInit**, on initialise les pins GPIO G4 et G15 en entrée. Puis on détermine le mode d'interruption de G15 sur front descendant avec **15 GPIO_INTR_POSEDGE gpio_set_intr_type drop**.

Test de l'encodage

Cette partie de code n'est pas à exploiter dans un montage définitif. Elle sert seulement à vérifier que l'encodeur est correctement branché et fonctionne bien:

```

: test ( -- )
  cr ." PIN: "
  cr ." - G15: " 15 digitalRead .
  cr ." - G04: " 04 digitalRead .
;

pinsInit    \ initialise G4 and G15
' test 15 pinchange

```

C'est la séquence **' test 15 pinchange** qui indique à ESP32Forth d'exécuter le code de test si une interruption se déclenche sur action de la borne G15.

Résultat de l'action sur notre encodeur. Nous n'avons conservé que les résultats d'actions arrivant en butée, une fois dans le sens horaire inverse, puis dans le sens horaire:

```

PIN:
- G15: 1 \ reverse clockwise turn
- G04: 1
PIN:
- G15: 0 \ clockwise turn
- G04: 1

```

Incrémenter et décrémenter une variable avec l'encodeur

Maintenant que nous avons testé l'encodeur par interruption matérielle, on va pouvoir gérer le contenu d'une variable. Pour ce faire, on définit notre variable **KYvar** et les mots permettant d'en modifier son contenu:

```

0 value KYvar    \ content is incremented or decremented

\ increment content of KYvar

```

```

: incKYvar ( n -- )
  1 +to KYvar
;

\ decrement content of KYvar
: decKYvar ( n -- )
  -1 +to KYvar
;

```

Le mot **incKYvar** incrémente le contenu de **KYvar**. Le mot **decKYvar** décrémente le contenu de **KYvar**.

On teste la modification du contenu de la variable **KYvar** via ce mot **testIncDec** défini ainsi:

```

\ used by interruption when G15 activated
: testIncDec ( -- )
  intG15disable
  15 digitalRead if
    04 digitalRead if
      decKYvar
    else
      incKYvar
    then
      cr ." KYvar: " KYvar .
  then
  1000 0 do loop \ small wait loop
  intG15enable
;

pinsInit
' testIncDec 15 pinchange

```

Tournez la commande de l'encodeur vers la droite (sens horaire) va incrémenter le contenu de la variable KYvar. Une rotation vers la gauche décrémente le contenu de la variable KYvar:

```

pinsInit
' testIncDec 15 pinchange
-->
KYvar: 1    \ rotate Clockwise
KYvar: 2
KYvar: 3
KYvar: 4
KYvar: 3    \ rotate Contra Clockwise

```

```
KYvar: 2  
KYvar: 1  
KYvar: 0  
KYvar: -1  
KYvar: -2
```

Listing complet: using the KY-040 Rotary Encoder @TODO : mettre listing complet en section listing

Clignotement d'une LED par timer

Débuter en programmation FORTH

Tout débutant en programmation connaît très bien cet exemple plus que classique : le clignotement d'une LED. Voici le code source, en langage C pour ESP32:

```
/*
 * This ESP32 code is created by esp32io.com
 * This ESP32 code is released in the public domain
 * For more detail (instruction and wiring diagram),
 * visit https://esp32io.com/tutorials/esp32-led-blink
 */

// the code in setup function runs only one time when ESP32 starts
void setup() {
  // initialize digital pin GPIO18 as an output.
  pinMode(18, OUTPUT);
}

// the code in loop function is executed repeatedly infinitely
void loop() {
  digitalWrite(18, HIGH); // turn the LED on
  delay(500);             // wait for 500 milliseconds
  digitalWrite(18, LOW);  // turn the LED off
  delay(500);             // wait for 500 milliseconds
}
```

En langage FORTH, ce n'est guère différent :

```
18 constant myLED

: led.blink ( -- )
  myLED output pinMode
  begin
    HIGH myLED pin
    500 ms
    LOW myLED pin
    500 ms
  key? until
;
```

Si vous compilez ce code FORTH avec ESP32forth installé sur votre carte ESP32 et que vous tapez **led.blink** depuis le terminal, la LED connectée au port GPIO18 va clignoter.

Pour injecter un code écrit en langage C, il faudra le compiler sur le PC, puis le téléverser sur la carte ESP32, opérations qui prennent un certain temps. Alors qu'avec le langage FORTH, le compilateur est déjà opérationnel sur notre carte ESP32. Le compilateur va compiler le programme écrit en langage FORTH en deux à trois secondes et permettre son exécution immédiate en tapant simplement le mot contenant ce code, ici **led.blink** pour notre exemple.

En langage FORTH, on peut compiler des centaines de mots et les tester immédiatement, tous de manière individuelle, ce que ne permet pas du tout le langage C.

On factorise notre code FORTH comme ceci :

```
18 constant myLED

: led.on ( -- )
  HIGH myLED pin
;

: led.off ( -- )
  LOW myLED pin
;

: waiting ( -- )
  500 ms
;

: led.blink ( -- )
  myLED output pinMode
  begin
    led.on      waiting
    led.off     waiting
  key? until
;
```

Depuis le terminal, on peut simplement allumer la LED en tapant **led.on** et l'éteindre en tapant **led.off**. L'exécution de **led.blink** reste possible.

La factorisation a pour but de découper une fonction complexe et peu lisible en un ensemble de fonctions plus simples et lisibles. Avec FORTH, la factorisation est conseillée, d'une part pour permettre une mise au point plus facile, d'autre part pour permettre la réutilisation des mots factorisés.

Ces explications peuvent paraître triviales pour ceux qui connaissent et maîtrisent le langage FORTH. C'est loin d'être évident pour les personnes programmant en langage C, obligés de regrouper les appels de fonctions dans la fonction générale **loop()**.

Maintenant que ceci est expliqué, on va tout oublier ! Parce que...

Clignotement par TIMER

On va oublier tout ce qui a été expliqué précédemment. Parce que cet exemple de clignotement de LED a un énorme inconvénient. Notre programme ne fait que ça et rien d'autre. En clair, c'est un vrai gâchis matériel et logiciel de faire clignoter une LED à notre carte ESP32. Nous allons voir une manière très différente de produire ce clignotement, en langage FORTH exclusivement.

ESP32forth dispose de deux mots qui vont être très utiles pour gérer ce clignotement de LED: **interval** et **rerun**.

Mais avant d'aborder le fonctionnement de ces deux mots, intéressons-nous à la notion d'interruption...

Les interruptions matérielles et logicielles

Si vous pensez gérer des micro-contrôleurs sans vous intéresser aux interruptions matérielles ou logicielles, alors abandonnez le développement informatique pour les cartes ESP32 !

Vous avez le droit de débiter et ne pas connaître les interruptions. Et on va vous expliquer les interruptions et la manière d'utiliser les interruptions par timers.

Voici un exemple pas du tout informatique de ce qu'est une interruption :

- vous attendez un colis important ;
- vous descendez toutes les minutes au portail de votre domicile, voir si le facteur est arrivé.

Dans ce scénario, vous passez en fait votre temps à descendre, regarder, remonter. En fait, vous n'avez quasiment plus le temps de faire autre chose...

Dans la réalité, voici ce qui doit se passer :

- vous restez dans votre domicile;
- le facteur arrive et sonne à la porte;
- vous descendez et récupérez votre colis...

Un micro-contrôleur, ce qui inclue la carte ESP32, dispose de deux types d'interruptions :

- **les interruptions matérielles** : elles se déclenchent sur une action physique sur une des entrées GPIO de la carte ESP32 ;
- **les interruptions logicielles** : elles se déclenchent si certains registres atteignent des valeurs pré-définies.

C'est le cas des interruption par timer, que nous allons définir comme interruptions logicielles.

Utiliser les mots **interval** et **rerun**

Le mot **interval** est défini dans le vocabulaire **timers**. Il accepte trois paramètres :

- **xt** qui est le code d'exécution du mot à lancer quand l'interruption se déclenche;
- **usec** est le délai d'attente, en micro-secondes, avant déclenchement de l'interruption;
- **t** est le numéro du timer à déclencher. Ce paramètre doit être dans l'intervalle [0..3]

Reprenons partiellement le code factorisé de notre clignotement LED :

```
18 constant myLED

0 value LED_STATE

: led.on ( -- )
  HIGH dup myLED pin
  to LED_STATE
;

: led.off ( -- )
  LOW dup myLED pin
  to LED_STATE
;

timers \ select timers vocabulary
: led.toggle ( -- )
  LED_STATE if
    led.off
  else
    led.on
  then
    0 rerun
;

' led.toggle 500000 0 interval

: led.blink
  myLED output pinMode
  led.toggle
;
```

Le mot **rerun** est précédé du numéro de timer activé avant la définition de **interval**. Le mot **rerun** doit être utilisé dans la définition du mot exécuté par le timer.

Le mot **led.blink** initialise la sortie GPIO utilisée par la LED, puis exécute **led.toggle**.

Dans cette séquence FORTH ' **led.toggle 500000 0 interval**, on initialise le timer 0 en récupérant le code d'exécution du mot utilisant **rerun**, suivi de l'intervalle de temps, ici 500 millisecondes, puis le numéro du timer à déclencher.

Le clignotement de la LED démarre immédiatement après exécution du mot **led.blink**.

L'interpréteur FORTH de ESP32forth reste accessible pendant le clignotement de la LED, chose impossible en langage C!

Minuterie pour femme de ménage

Préambule

On est en 1990. C'est un programmeur informatique qui travaille beaucoup. Donc il lui arrive de quitter son bureau un peu tard.

Et c'est lors d'une de ses sorties tardives de bureau qu'il s'engage dans le couloir, un de ces couloirs avec un bouton de minuterie à chaque extrémité. La lumière est déjà allumée. Mais par réflexe, notre ami programmeur appuie sur l'interrupteur et se pique le doigt. Une pointe en bois est fichée dans l'interrupteur pour bloquer la minuterie.



C'est la femme de ménage qui est en train de nettoyer le sol qui lui explique : "oui. La minuterie ne tient qu'une minute. Et je me retrouve souvent dans l'obscurité. Comme j'en ai assez de rappuyer sans cesse sur l'interrupteur de la minuterie, je bloque le bouton avec cette petite pointe en bois"...

Une solution

Cette anecdote a fait germer dans la tête de notre programmeur une idée. Comme il avait quelques connaissances sur les microcontrôleurs, il s'est mis en tête de trouver une solution pour la femme de ménage.

L'histoire ne dit pas dans quel langage il a programmé sa solution. Certainement en assembleur.

Il a dérivé la commande des lumières vers son circuit :

- un appui ordinaire déclenche la minuterie pour une minute;
- si la lumière est allumée, tout appui bref sur un bouton ramène le délai d'allumage à une minute;
- le secret de notre programmeur est d'avoir prévu un appui long de 3 secondes ou plus. Cet appui long enclenche la minuterie pour 10 minutes d'allumage;
- si la minuterie est en circuit long, un nouvel appui long ramène le délai de la minuterie à une minute;

- un bip sonore bref acquitte l'activation ou la désactivation d'un cycle long de minuterie.

La femme de ménage a fort apprécié cette amélioration de la minuterie. Elle n'a plus eu besoin de bloquer le bouton d'une quelconque manière.

Et les autres travailleurs ? Comme personne n'était informé de cette fonctionnalité, ils ont continué à utiliser la minuterie en appuyant brièvement sur l'interrupteur d'activation.

Une minuterie en FORTH pour ESP32Forth

Vous l'avez compris, on va utiliser **timers** pour gérer une minuterie en y intégrant le scénario décrit précédemment.

```
\ myLIGHTS connecté à GPIO18
18 constant myLIGHTS

\ définit temps max pour cycle normal ou étendu, en secondes
60 constant MAX_LIGHT_TIME_NORMAL_CYCLE
600 constant MAX_LIGHT_TIME_EXXTENDED_CYCLE

\ temps max pour cycle normal ou étendu, en secondes
0 value MAX_LIGHT_TIME

timers
\ coupe éclairage si MAX_LIGHT_TIME égal 0
: cycle.stop ( -- )
  -1 +to MAX_LIGHT_TIME      \ décrémente temps max de 1 seconde
  MAX_LIGHT_TIME 0 = if
    LOW myLIGHTS pin        \ coupe éclairage
  else
    0 rerun
  then
;

\ initialise timer 0
' cycle.stop 1000000 0 interval

\ démarre un cycle d'éclairage, n est délai en secondes
: cycle.start ( n -- )
  1+ to MAX_LIGHT_TIME      \ sélect. Temps max
  myLIGHTS output pinMode
  HIGH myLIGHTS pin        \ active éclairage
  0 rerun
;
```

On peut déjà tester notre minuterie :

```
3 cycle.start \ allume éclairage pour 3 secondes
10 cycle.start \ allume éclairage pour 10 secondes
```

Si on relance **cycle.start** pendant que la lumière est allumée, on repart pour un nouveau cycle d'allumage de n secondes.

Il nous reste donc à gérer l'activation de ces cycles depuis un interrupteur.

Gestion du bouton d'allumage lumière

C'est pas sorcier. On va gérer un bouton poussoir. Comme nous avons une carte ESP32 sous la main, programmable avec ESP32Forth, on va en profiter pour gérer ce bouton par interruptions. Les interruptions gérant les bornes GPIO sur la carte ESP32 sont des interruptions matérielles.

Notre bouton est monté sur la borne GPIO17 (G17).

On définit au passage deux mots, **intPosEdge** et **intNegEdge**, lesquels déterminent le type de déclenchement de l'interruption :

- **intPosEdge** pour déclencher l'interruption sur front montant ;
- **intNegEdge** pour déclencher l'interruption sur front descendant.

```
17 constant button \ mount button on GPIO17

interrupts \ select interrupts vocabulary

\ interrupt activated for upraising signal
: intPosEdge ( -- )
  button #GPIO_INTR_POSEDGE gpio_set_intr_type drop
;

\ interrupt activated for falldown signal
: intNegEdge ( -- )
  button #GPIO_INTR_NEGEDGE gpio_set_intr_type drop
;
```

Nous avons ensuite besoin de définir quelques variables et constantes :

- deux constantes, **CYCLE_SHORT** et **CYCLE_LONG** qui serviront à définir la durée d'allumage des lumières. Ici on a choisi 3 et 10 secondes pour faire nos tests.
- la variable **msTicksPositiveEdge** qui mémorise la position du compteur d'attente délivré par ms-ticks
- la constante **DELAY_LIMIT** qui détermine le seuil de détermination d'un appui bref ou long sur le bouton poussoir. Ici, c'est 3000 millisecondes, soit 3 secondes. Un usager normal n'appuiera pour ainsi dire JAMAIS 3 secondes sur le bouton

d'allumage de la lumière. Seule la femme de ménage connaît la manœuvre pour avoir un allumage continu long...

```
03 constant CYCLE_SHORT      \ durée éclairage pour appui bref, en
secondes
10 constant CYCLE_LONG       \ durée éclairage pour appui long

\ mémorise valeur de ms-ticks sur front montant
variable msTicksPositiveEdge

\ délai limite : si delai < DELAY_LIMIT, cycle court
3000 constant DELAY_LIMIT
```

Le mot **getButton** est lancé à chaque interruption déclenchée par appui sur le bouton poussoir connecté à GPIO17 (G17) sur notre carte ESP32.

Au début de l'exécution de **getButton**, les interruptions sur G17 sont désactivées. Cette interruption sera réactivée en fin de définition. Cette désactivation est nécessaire pour empêcher l'empilement des interruptions.

La désactivation est suivie de la boucle **70000 0 do loop**. Cette boucle sert à gérer les rebonds de contact. Ici on gère l'anti-rebond par logiciel.

```
\ mot exécuté par interruption
: getButton ( -- )
  button gpio_intr_disable drop
  70000 0 do loop \ anti rebond
  button digitalRead 1 =
  if
    ms-ticks msTicksPositiveEdge !
    intNegEdge
  else
    intPosEdge
    ms-ticks msTicksPositiveEdge @ -
    DELAY_LIMIT >
    if      CYCLE_LONG  cr ." BEEP"
    else    CYCLE_SHORT cr ." ----"
  then
  cycle.start
  button gpio_intr_enable drop
;
```

Au front montant, le mot **getButton** enregistre l'état du compteur de délai et positionne les interruptions sur front descendant. Puis on quitte ce mot en réactivant les interruptions.

Au front descendant, le mot **getButton** calcule le temps écoulé depuis le front montant. Si ce délai est supérieur à **DELAY_LIMIT**, on engage un cycle d'allumage long. Sinon, on

engage un cycle d'allumage court.

L'engagement d'un cycle d'allumage long est matérialisé par l'affichage sur le terminal de "BEEP".

Dans le scénario d'origine, c'est matérialisé par un bip sonore bref.

Pour finir, on initialise le bouton et l'interruption matérielle sur ce bouton :

```
\ initialise bouton et vecteurs d'interruption
button input pinMode          \ sélectionne G17 en mode entrée
button gpio_pulldown_en drop  \ active résistance interne de G17
' getButton button pinchange
intPosEdge

forth
```

Conclusion

Voir la vidéo du montage : https://www.youtube.com/watch?v=OHWMh_bIWz0

Ce cas d'école, très simple, montre comment gérer simultanément le timer et une interruption matérielle.

Ces deux mécanismes sont très peu préemptifs. Le timer laisse disponible l'accès à l'interpréteur FORTH. L'interruption matérielle est opérationnelle même si FORTH exécute un autre processus.

Nous ne sommes pas en multi-tâche. Il est important de le dire !

Je souhaite seulement que ce cas d'école vous donne maintenant beaucoup d'idées pour vos développements...

Horloge temps réel logicielle

Le mot MS-TICKS

Le mot **MS-TICKS** est utilisé dans la définition du mot **ms**:

```
DEFINED? ms-ticks [IF]
: ms ( n -- )
  ms-ticks >r
  begin
    pause ms-ticks r@ - over
  >= until
  rdrop drop
;
[THEN]
```

Ce mot **MS-TICKS** est au cœur de nos investigations. Si on met en route la carte ESP32, son exécution restitue le nombre de millisecondes écoulées depuis la mise en route de la carte ESP32. Cette valeur croît toujours. La valeur de saturation de ce comptage est de $2^{32}-1$, soit 4294967295 millisecondes, soit 49 jours environ...

A chaque redémarrage de la carte ESP32, cette valeur redémarre à zéro.

Gestion d'une horloge logicielle

A partir des données **HH MM SS** (Heures, minutes, secondes), il est aisé de reconstituer une valeur entière, en millisecondes, correspondant au temps écoulé depuis 00:00:00. Si à ce temps on soustrait la valeur de **MS-TICKS**, on a une valeur horaire de départ pour déterminer l'heure réelle. On initialise donc un compteur de base **currentTime** à partir du mot **RTC.set-time** :

```
0 value currentTime

\ store current time
: RTC.set-time { hh mm ss -- }
  hh 3600 *
  mm 60 *
  ss + + 1000 *
  MS-TICKS - to currentTime
;
```

Exemple d'initialisation: **22 52 00 RTC.set-time** initialise la base de temps pour 22:52:00...

Pour bien initialiser, préparez les trois valeurs **HH MM SS** suivies du mot **RTC.set-time**, surveillez votre montre. Quand l'heure attendue arrive, exécuter la séquence d'initialisation.

L'opération inverse récupère les valeurs **HH MM** et **SS** de l'heure courante, ceci grâce à ce mot :

```
\ récupère temps actuel en secondes
: RTC.get-time ( -- hh mm ss )
  currentTime MS-TICKS + 1000 /
  3600 /mod swap 60 /mod swap
;
```

Enfin, on définit le mot **RTC.display-time** qui vous permet d'afficher l'heure courante après initialisation de notre horloge logicielle:

```
\ used for SS and MM part of time display
: :## ( n -- n' )
  # 6 base ! # decimal [char] : hold
;

\ display current time
: RTC.display-time ( -- )
  currentTime MS-TICKS + 1000 /
  <# :## :## 24 MOD #S #> type
;
```

L'étape suivante serait de se connecter à un serveur de temps, avec le protocole NTP, pour initialiser automatiquement notre horloge logicielle.

Mesurer le temps d'exécution d'un mot FORTH

Mesurer la performance des définitions FORTH

Commençons par définir le mot **measure** : qui va effectuer ces mesures de temps d'exécution :

```
: measure: ( exec: -- <word> )
  ms-ticks >r
  ' execute
  ms-ticks r> -
  cr ." execution time: "
  <# # # # [char] . hold #s #> type ." sec." cr
;
```

Dans ce mot, on récupère le temps par **ms-ticks**, puis on récupère le code d'exécution du mot qui suit **measure** : on exécute ce mot, on récupère la nouvelle valeur de temps par **ms-ticks**. On fait la différence, laquelle correspond au temps écoulé, en millisecondes, pris par le mot pour s'exécuter. Exemple :

```
measure: words
\ affiche: execution time: 0.210sec.
```

Le mot **words** a été exécuté en 0,2 secondes. Ce temps ne tient pas compte des délais de transmission par le terminal. Ce temps ne tient pas compte non plus du délai pris par **measure** : pour récupérer le code d'exécution du mot à mesurer.

S'il y a des paramètres à passer au mot à mesurer, ceux-ci doivent être empilés avant d'appeler **measure** : suivi du mot à mesurer:

```
: SQUARE ( n -- n-exp2 )
  dup *
;
3 measure: SQUARE
\ affiche:
\ execution time: 0.000sec.
```

Ce résultat signifie que notre définition **SQUARE** s'exécute en moins d'une milliseconde.

On va réitérer cette opération un certain nombre de fois :

```
: test-square ( -- )
  1000 for
    3 SQUARE drop
  next
```

```
;
3 measure: test-square
\ affiche:
\ execution time: 0.001sec.
```

En exécutant 1000 fois le mot **SQUARE**, précédé d'un empilage de valeur et dépileage du résultat, on arrive à un délai d'exécution de 1 milliseconde. On peut raisonnablement déduire que **SQUARE** s'exécute en moins d'une micro-seconde!

Test de quelques boucles

On va tester quelques boucles, avec 1 million d'itérations. Commençons avec une boucle **do-loop** :

```
: test-loop ( -- )
  1000000 0 do
  loop
;
measure: test-loop
\ display:
\ execution time: 1.327sec.
```

Voyons maintenant avec une boucle **for-next** :

```
: test-for ( -- )
  1000000 for
  next
;
measure: test-for
\ affiche:
\ execution time: 0.096sec.
```

La boucle **for-next** s'exécute presque 14 fois plus rapidement que la boucle **do-loop**.

Voyons ce qu'une boucle **begin-until** a dans le ventre :

```
: test-begin ( -- )
  1000000 begin
    1- dup 0=
  until
;
measure: test-begin
\ affiche:
\ execution time: 0.273sec.
```

C'est plus performant que la boucle **do-loop**, mais quand même trois fois plus lent que la boucle **for-next**.

Vous voilà maintenant outillé pour réaliser des programmes FORTH encore plus performants.

Programmer un analyseur d'ensoleillement

Préambule

Dans le cadre d'un projet solaire utilisant plusieurs panneaux solaires et leur micro-onduleur, il apparaît quelques soucis de gestion de l'énergie électrique produite.

Le principal souci est d'activer des appareils gros consommateurs seulement si les panneaux solaires produisent par plein soleil. Un appareil en particulier est concerné, le cumulus eau chaude :

- activer l'appareil quand les panneaux sont en plein soleil ;
- désactiver l'appareil quand passent des nuages.

Les micro-onduleurs injectent du courant dans le réseau électrique général. Si un appareil gros consommateur d'électricité est actif quand passent des nuages, cet appareil sera alimenté en priorité par le réseau général.

Dans cet article, nous présentons une solution permettant la détection des nuages grâce à un panneau solaire miniature et une carte ESP32.

Code complet disponible ici :

<https://github.com/MPETREMANN11/ESP32forth/blob/main/ADC/solarLightAnalyzer.txt>

Le panneau solaire miniature

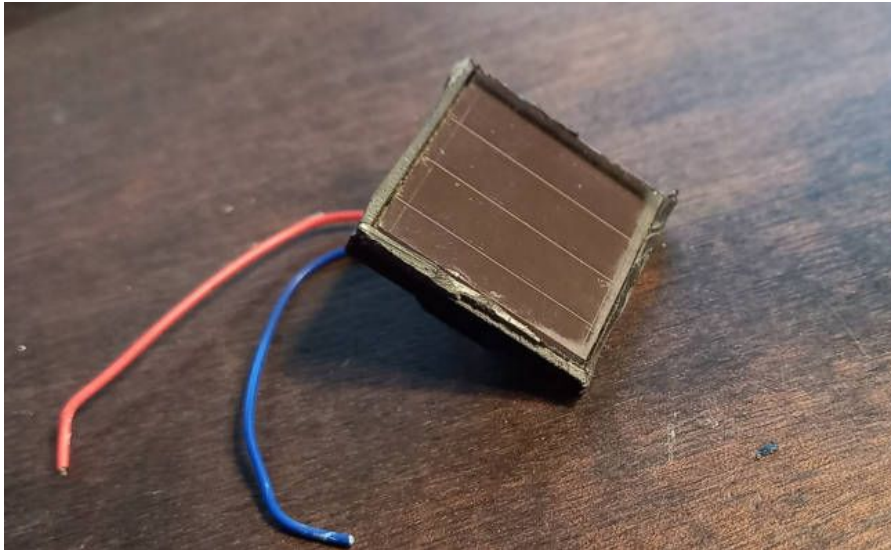
Pour réaliser notre détecteur de nuages, on va utiliser un panneau solaire de très petite taille, ici un panneau de 25mm x 25mm.

Récupération d'un panneau solaire miniature

Ce panneau solaire miniature est récupéré sur une lampe de jardin qui est hors d'usage :



Voici notre mini panneau solaire sorti de cette lampe de jardin :



On sacrifie deux connecteurs dupont pour permettre d'effectuer diverses mesures sur notre plaque de prototype. Ces connecteurs sont soudés sur les deux fils rouge et bleu sortant du mini panneau solaire.

Mesure de la tension du panneau solaire

On commence par relever la tension à vide de notre mini panneau solaire, ici avec un oscilloscope. Cette mesure de tension peut aussi s'effectuer avec un voltmètre :



En pleine lumière, la tension mesurée s'élève à 14,2 Volts!

Sous une lumière diffuse, la tension descend à 5,8 Volts.

En couvrant de la main le mini panneau solaire, la tension chute à quasiment 0 Volt.

Mesure du courant du panneau solaire

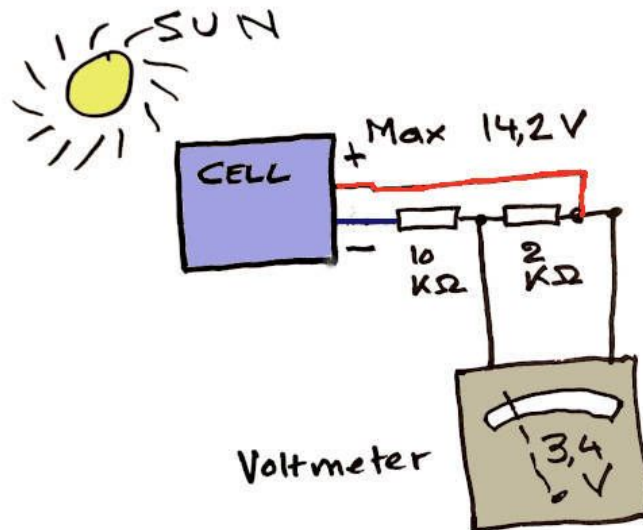
Le courant, c'est à dire l'intensité, doit être relevée à l'aide d'un ampèremètre. La fonction ampèremètre d'un contrôleur universel conviendra. La mise en court-circuit du mini panneau solaire en pleine lumière permet de mesurer un courant de 10 mA.

Notre mini panneau solaire a donc une puissance approximative de 0,2 Watt.

Avant de connecter notre mini panneau solaire à la carte ESP32, il faut impérativement procéder à un abaissement de la tension de sortie. Il est hors de question d'injecter cette tension de 14,2 Volts dans une entrée de la carte ESP32. Une telle tension détruirait les circuits internes de la carte ESP32.

Abaissement de la tension du panneau solaire

L'idée est d'abaisser la tension aux bornes de notre mini panneau solaire. Après quelques tests, on choisit deux résistances, une de 220 Ohms, l'autre de 1K Ohms. Montage de ces résistances :



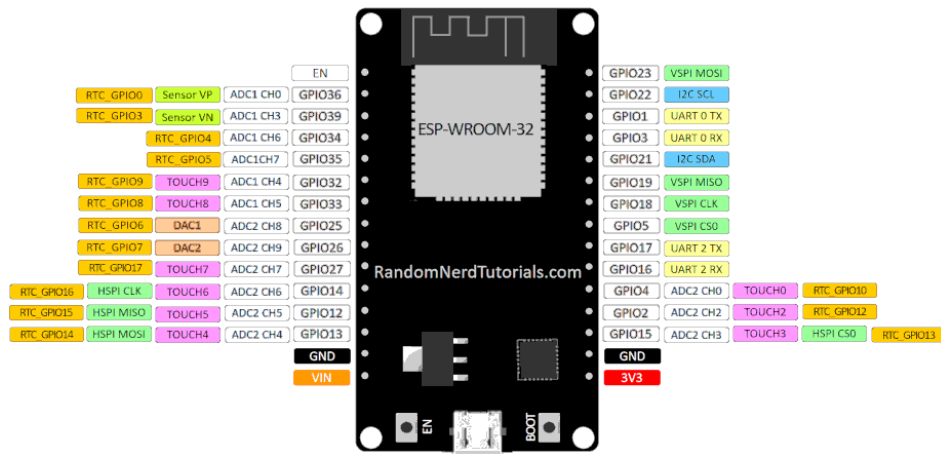
La mesure de tension est relevée entre les deux résistances et la borne positive du panneau solaire.

Le voltmètre indique maintenant une tension maximale de 3,2V en pleine lumière, une tension de 0,35V en lumière diffuse.

Programmation de l'analyseur solaire

La carte ESP32 dispose de 18 canaux 12 bits permettant la conversion analogique numérique (ADC). Pour analyser la tension de notre mini panneau solaire, un seul canal ADC est nécessaire et suffisant.

Seuls 15 canaux ADC sont disponibles :



Nous allons en utiliser un, le canal **ADC1_CH6** qui est rattaché au pin **G34** :

```
34 constant SOLAR_CELL

: init-solar-cell ( -- )
    SOLAR_CELL input pinMode
;

init-solar-cell
```

Pour lire la tension au point situé entre les deux résistances, il suffit d'exécuter **SOLAR_CELL analogRead**. Cette séquence dépose une valeur comprise entre 0 et 4095. La valeur la plus basse correspond à une tension nulle. La valeur la plus élevée correspond à une tension de 3,3 Volts.

Voici la définition **solar-cell-read** pour récupérer cette tension:

```
: solar-cell-read ( -- n )
    SOLAR_CELL analogRead
;
```

Testons cette définition dans une boucle :

```
: solar-cell-loop ( -- )
    init-solar-cell
    begin
        solar-cell-read cr .
        200 ms
    key? until
;
```

A l'exécution de **solar-cell-loop**, toutes les 200 millisecondes, la valeur de la conversion de tension ADC est affichée :

```
...
322
```

```
331
290
172
39
0
0
0
0
19
79
86
...
```

Ici les valeurs ont été obtenues en éclairant le mini panneau solaire avec une lampe de forte puissance. Les valeurs nulles correspondent à l'absence d'éclairage.

Des essais avec le vrai soleil font remonter des mesures dépassant 300.

Gestion activation et désactivation d'un appareil

Pour commencer, nous allons définir deux pins, un pin réservé à la gestion d'un signal d'activation, l'autre à un signal de désactivation:

- pin G17 connecté à une LED verte. Ce pin sert à activer un appareil.
- pin G16 connecté à une LED rouge. Ce pin sert à désactiver un appareil.

```
17 constant DEVICE_ON      \ green LED
16 constant DEVICE_OFF     \  red LED

: init-device-state ( -- )
  DEVICE_ON output pinMode
  DEVICE_OFF output pinMode
;
```

On aurait pu utiliser un seul pin pour gérer l'appareil distant. Mais certains appareils, comme les relais bistables ont deux bobines :

- on alimente la première bobine pour que les contacts commutent. L'état ne change pas quand la bobine n'est plus excitée ;
- pour revenir à l'état initial, on alimente la deuxième bobine.

Pour cette raison, notre programmation va tenir compte de ce type d'appareil.

```
\ define trigger high state delay
500 value DEVICE_DELAY

\ set HIGH level of trigger
```

```

: device-activation { trigger -- }
    trigger HIGH digitalWrite
    DEVICE_DELAY ?dup
    if
        ms
        trigger LOW digitalWrite
    then
;

```

Ici, la pseudo-constante **DEVICE_DELAY** sert à indiquer le délai pendant lequel le signal de commande doit être conservé à l'état haut. Passé ce délai, le signal de commande repasse à l'état bas.

Si la valeur de **DEVICE_DELAY** est nulle, le signal de commande reste à l'état haut.

C'est le mot **trigger-activation** qui gère l'activation du pin correspondant :

- **TRIGGER_ON trigger-activation** met à l'état haut de manière permanente ou transitoire le pin attaché à la LED verte ;
- **TRIGGER_OFF trigger-activation** met à l'état haut de manière permanente ou transitoire le pin attaché à la LED rouge.

On définit maintenant deux mots, **device-ON** et **device-OFF**, respectivement chargés d'activer et désactiver l'appareil destiné à être commandé par les pins G16 et G17 :

```

\ define device state: 0=LOW, -1=HIGH
0 value DEVICE_STATE

: enable-device ( -- )
    DEVICE_STATE invert
    if
        DEVICE_OFF LOW digitalWrite
        DEVICE_ON device-activation
        -1 to DEVICE_STATE
    then
;

: disable-device ( -- )
    DEVICE_STATE
    if
        DEVICE_ON LOW digitalWrite
        DEVICE_OFF device-activation
        0 to DEVICE_STATE
    then
;

```

L'état de l'appareil est mémorisé dans **DEVICE_STATE**. Cet état est testé avant une tentative de changement d'état. Si l'appareil est actif, il ne sera pas réactivé de manière répétée. Idem si l'appareil est inactif.

```
\ define trigger value for sunny or cloudy sky
300 value SOLAR_TRIGGER

\ if solar light > SOLAR_TRIGGER, activate action
: action-light-level ( -- )
  solar-cell-read SOLAR_TRIGGER >=
  if
    enable-device
  else
    disable-device
  then
;
;
```

Déclenchement par interruption timer

La manière la plus élégante consiste à exploiter une interruption par timer. On va utiliser le timer 0:

```
0 to DEVICE_DELAY
200 to SOLAR_TRIGGER
init-solar-cell
init-device-state

timers
: action ( -- )
  action-light-level
  0 rerun
;

' action 1000000 0 interval
```

A partir de maintenant, le timer va analyser le flux lumineux chaque seconde et agir en conséquence. Lien vers la vidéo : <https://youtu.be/IAjeev2u9fc>

Pour cette vidéo, on agit sur deux paramètres :

- **0 to DEVICE_DELAY** allume les LEDs de manière permanente. La LED rouge indique que l'appareil est désactivé. La LED verte indique l'activation de l'appareil;
- **200 to SOLAR_TRIGGER** détermine le seuil de déclenchement de l'état d'ensoleillement. Ce paramètre est ajustable pour s'adapter aux caractéristiques du mini panneau solaire.

Le mot **action** fonctionne par interruption timer. Il n'est donc pas nécessaire d'avoir une boucle générale pour que le détecteur fonctionne.

Appareils commandés par le capteur d'ensoleillement

En résumé, nous disposons de deux fils de commande, un fil correspondant à la LED verte sur la vidéo, l'autre fil correspondant à la LED rouge. Le programme est conçu pour que les deux fils de commande ne puissent être actifs en même temps.

Pour avoir un signal continu sur l'un ou l'autre fil de commande, il suffit que la valeur **DEVICE_DELAY** soit nulle. Voici comment initialiser ce scénario :

```
\ start with Constant Command Signal
: start-CCS ( -- )
    0 to DEVICE_DELAY
    200 to SOLAR_TRIGGER
    init-solar-cell
    init-device-state
    disable-device
    [ timers ] ['] action 1000000 0 interval
;
```

Et pour avoir des commandes temporisées, on affectera à **DEVICE_DELAY** le délai du niveau de la commande d'activation ou de désactivation de l'appareil.

```
\ start with Temporized Command Signal
: start-TCS ( -- )
    300 to DEVICE_DELAY
    200 to SOLAR_TRIGGER
    init-solar-cell
    init-device-state
    disable-device
    [ timers ] ['] action 1000000 0 interval
;
```

Le scénario **start-TCS** est typique d'une commande de relais bistable à commande par impulsion. Le relais s'active s'il reçoit une commande d'activation. Même si le signal d'activation retombe, le relais bistable reste actif. Pour désactiver le relais bistable, il faut lui transmettre une commande de désactivation sur la ligne de désactivation.

En conclusion, notre analyseur de lumière solaire peut commander une grande variété d'appareils. Il suffit d'adapter les interfaces de commande de ces appareils aux caractéristiques des ports GPIO de la carte ESP32.

Gestion des sorties N/A (Numériques/Analogiques)

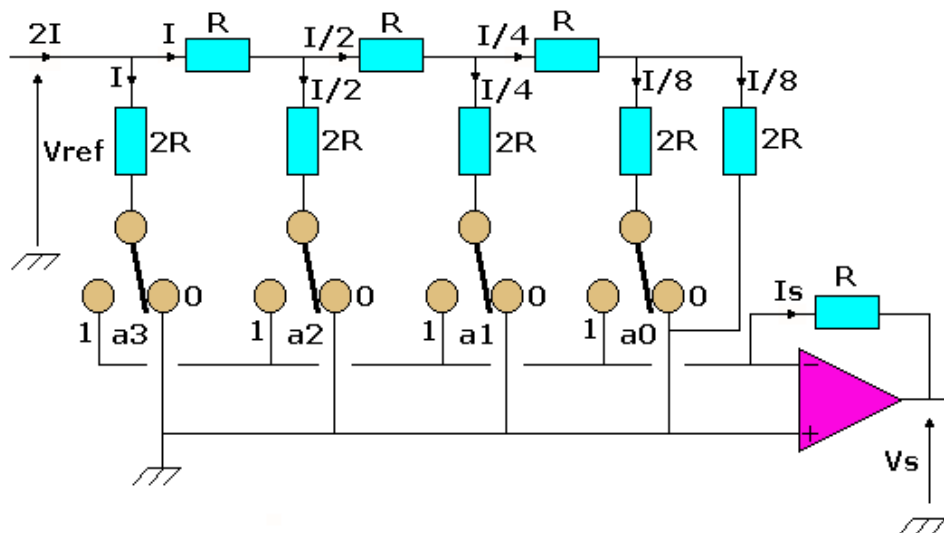
La conversion numérique / analogique

La conversion d'une grandeur numérique en une tension électrique proportionnelle à cette grandeur numérique est une fonctionnalité très intéressante sur un microcontrôleur.

Quand vous utilisez Internet et que vous passez un appel téléphonique en VOIP, votre voix est transformée en valeurs numériques. Celle de votre correspondant sera inversement transformée de chiffres vers des signaux sonores. Ce processus utilise la conversion analogique vers numérique et inversement.

La conversion N/A avec circuit R2R

Voici le schéma de base d'un convertisseur numérique analogique 4 bits :



La valeur à convertir, sur 4 bits, est répartie sur 4 pins a_0 à a_3 . La tension de référence est injectée en haut à gauche du circuit. Cette tension génère une intensité $2I$ si ce courant ne traverse aucune résistance.

Selon les bits activés, pour chaque bit la tension est divisée et additionnée à celle des autres bits actifs. Par exemple, si a_2 et a_0 sont actifs, le courant de sortie I_s sera la somme $I/2$ et $I/8$.

Pour ce circuit 4 bits, le pas de conversion est $I/16$. Avec ESP32, la conversion s'effectue sur 8 bits. Le pas de conversion sera donc $I/256$.

La conversion N/A avec ESP32

Aucune carte ARDUINO ne dispose de sortie de conversion N/A. Pour effectuer une conversion N/A avec une carte ARDUINO, il faut utiliser un composant externe.

Avec la carte ESP32, nous disposons de deux pins, G25 et G26, correspondant à des sorties de conversion N/A.

Pour notre première expérience de conversion N/A avec la carte ESP32, nous allons connecter deux LEDs aux pins G25 et G26 :

```
\ define Gx to LEDs
25 constant ledBLUE      \ blue led on G25
26 constant ledWHITE     \ white led on G26
```

Avant d'effectuer une conversion N/A, on prévoit l'initialisation des pins G25 et G26:

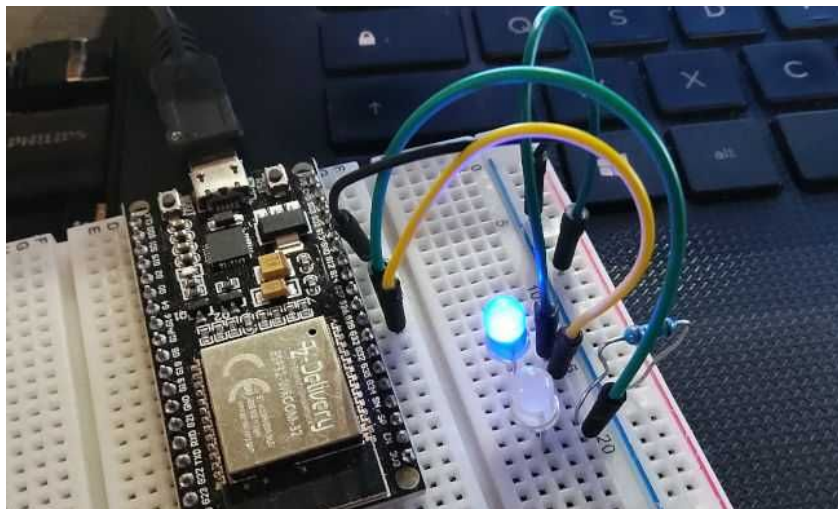
```
\ init Gx as output
: initLeds ( -- )
    ledBLUE output pinMode
    ledWHITE output pinMode
;
```

Et on définit deux mots permettant de contrôler l'intensité de nos deux LEDs:

```
\ set intensity for BLUE led
: BLset ( val -- )
    ledBLUE swap dacWrite
;

\ set intensity for WHITE led
: WHset ( val -- )
    ledWHITE swap dacWrite
;
```

Les mots **BLset** et **WHset** acceptent comme paramètre une valeur numérique dans l'intervalle 0..255.



Sur la photo, après **initLeds**, la séquence **200 BLset** allume la LED bleue à puissance réduite.

Pour l'allumer à pleine puissance, on utilisera la séquence **255 BLset**

Pour l'éteindre complètement, on enverra cette séquence **0 BLset**

Possibilités de la conversion N/A

Ici, avec nos deux LEDs, nous avons réalisé un montage simple et de peu d'intérêt.

Ce montage a le mérite de montrer que la conversion N/A fonctionne parfaitement. La conversion N/A permet:

- contrôle de puissance au travers d'un circuit dédié, un variateur pour moteur électrique par exemple;
- génération de signaux: sinusoïde, carré, triangle, etc...
- conversion de fichiers sons
- synthèse sonore...

Code complet disponible ici :

<https://github.com/MPETREMANN11/ESP32forth/blob/main/DAC/DAoutput.txt>

Installation de la librairie OLED pour SSD1306

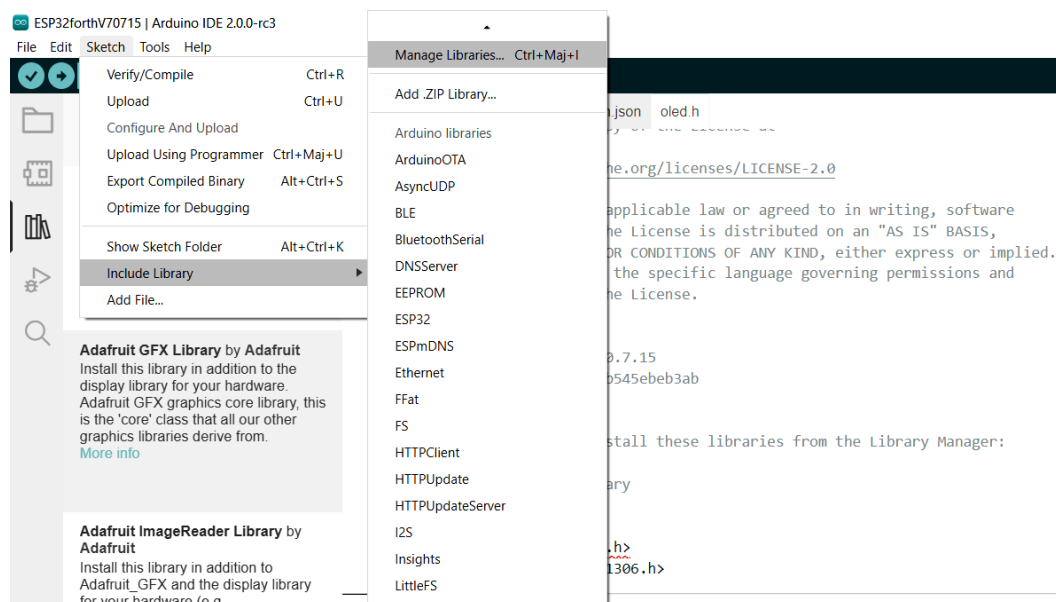
Depuis la version ESP32forth 7.0.7.15, les options sont disponibles dans le dossier **optional** :

Téléchargements > ESP32forth-7.0.7.15(1).zip > ESP32forth > optional		
	Nom	Type
✦	assemblers.h	Fichier H
✦	camera.h	Fichier H
✦	interrupts.h	Fichier H
✦	oled.h	Fichier H
✦	README-optional.txt	Document texte
	rmt.h	Fichier H
	serial-bluetooth.h	Fichier H
	spi-flash.h	Fichier H

Pour disposer du vocabulaire **oled**, copier le fichier **oled.h** vers le dossier contenant le fichier **ESP32forth.ino**.

Lancez ensuite ARDUINO IDE et sélectionnez le fichier **ESP32forth.ino** le plus récent.

Si la librairie OLED n'a pas été installée, dans ARDUINO IDE, cliquez sur *Sketch* et sélectionnez *Include Library*, puis sélectionnez *Manage Libraries*.



Dans le volet latéral gauche, cherchez la librairie **Adafruit SSD1306 by Adafruit**.

Vous pouvez maintenant lancer la compilation du croquis en cliquant sur *Sketch* et en sélectionnant *Upload*.

Une fois le croquis téléversé dans la carte ESP32, lancez le terminal TeraTerm. Vérifiez que le vocabulaire **oled** est bien présent :

```
oled vlist \ affiche:  
OledInit SSD1306_SWITCHCAPVCC SSD1306_EXTERNALVCC WHITE BLACK OledReset  
HEIGHT WIDTH OledAddr OledNew OledDelete OledBegin OledHOME OledCLS OledTextc  
OledPrintln OledNumln OledNum OledDisplay OledPrint OledInvert OledTextsize  
OledSetCursor OledPixel OledDrawL OledCirc OledCircF OledRect OledRectF  
OledRectR OledRectRF oled-builtins
```

L'interface I2C sur ESP32

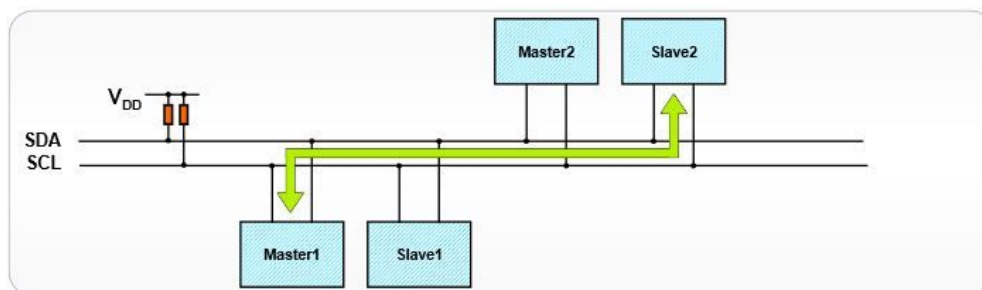
Introduction

I2C (signifie: Inter-Integrated Circuit, en anglais) est un bus informatique qui a émergé de la « guerre des standards » lancée par les acteurs du monde électronique. Conçu par Philips pour les applications de domotique et d'électronique domestique, il permet de relier facilement un microprocesseur et différents circuits, notamment ceux d'un téléviseur moderne : récepteur de la télécommande, réglages des amplificateurs basses fréquences, tuner, horloge, gestion de la prise péritel, etc.

Il existe d'innombrables périphériques exploitant ce bus, il est même implémentable par logiciel dans n'importe quel microcontrôleur. Le poids de l'industrie de l'électronique grand public a permis des prix très bas grâce à ces nombreux composants.

Ce bus porte parfois le nom de TWI (Two Wire Interface) ou TWSI (Two Wire Serial Interface) chez certains constructeurs.

Les échanges ont toujours lieu entre un seul maître et un (ou tous les) esclave(s), toujours à l'initiative du maître (jamais de maître à maître ou d'esclave à esclave). Cependant, rien n'empêche un composant de passer du statut de maître à esclave et réciproquement.



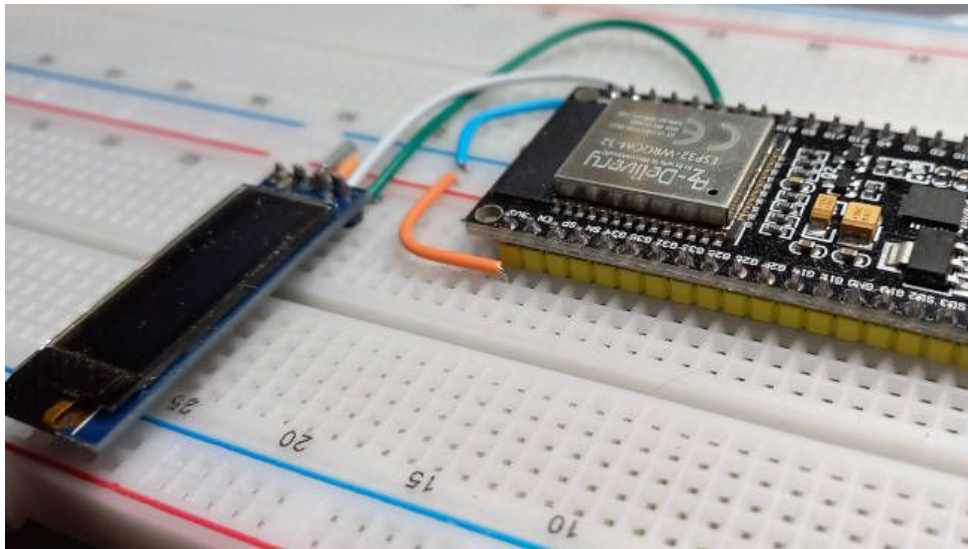
principe d'un bus I2C

La connexion est réalisée par l'intermédiaire de deux lignes:

- SDA (Serial Data Line) : ligne de données bidirectionnelle,
- SCL (Serial Clock Line) : ligne d'horloge de synchronisation bidirectionnelle.

Il ne faut pas oublier la masse qui doit être commune aux équipements.

Les deux lignes sont tirées au niveau de tension VDD à travers des résistances de pull-up (RP).



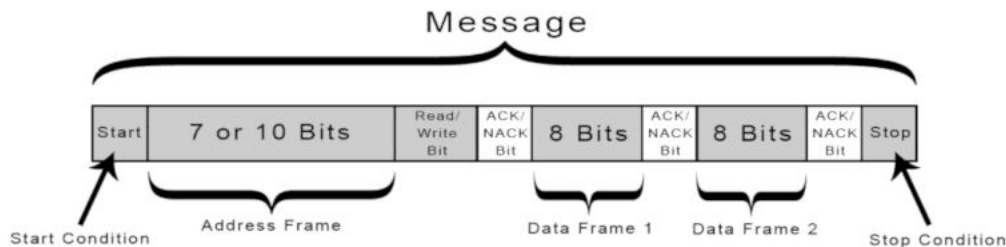
Afficheur OLED connecté au bus I2C

Échange maître esclave

Le message peut être décomposé en deux parties:

- Le maître est l'émetteur, l'esclave est le récepteur:
 - émission d'une condition de START par le maître (« S »),
 - émission de l'octet ou des octets d'adresse par le maître pour désigner un esclave, avec le bit R/W à 0 (voir la partie sur l'adressage ci-après),
 - réponse de l'esclave par un bit d'acquittement ACK (ou de non-acquittement NACK),
 - après chaque acquittement, l'esclave peut demander une pause (« PA »).
 - émission d'un octet de commande par le maître pour l'esclave,
 - réponse de l'esclave par un bit d'acquittement ACK (ou de non-acquittement NACK),
 - émission d'une condition de RESTART par le maître (« RS »),
 - émission de l'octet ou des octets d'adresse par le maître pour désigner le même esclave, avec le bit R/W à 1.
 - réponse de l'esclave par un bit d'acquittement ACK (ou de non-acquittement NACK).
- Le maître devient récepteur, l'esclave devient émetteur:
 - émission d'un octet de données par l'esclave pour le maître,
 - réponse du maître par un bit d'acquittement ACK (ou de non-acquittement NACK),

- émission d'autres octets de données par l'esclave avec acquittement du maître,
- pour le dernier octet de données attendu par le maître, il répond par un NACK pour mettre fin au dialogue,
- émission d'une condition de STOP par le maître (« P »).



Condition de démarrage: la ligne SDA passe d'un niveau de tension élevé à un niveau de tension bas avant que la ligne SCL ne passe de haut en bas.

Condition d'arrêt: la ligne SDA passe d'un niveau de basse tension à un niveau de tension élevé après le passage de la ligne SCL de bas à haut.

Cadre d'adresse: une séquence de 7 ou 10 bits unique pour chaque esclave qui identifie l'esclave lorsque le maître veut lui parler.

Bit de lecture/écriture: un seul bit spécifiant si le maître envoie des données à l'esclave (niveau bas de tension) ou en lui demandant des données (niveau haut de tension).

Bit ACK/NACK: chaque trame d'un message est suivie d'un acquittement/non-acquittement bit. Si une trame d'adresse ou une trame de données a été reçue avec succès, un bit ACK est renvoyé à l'expéditeur.

Adressage

I2C n'a pas de lignes de sélection d'esclaves comme SPI, il a donc besoin d'un autre moyen de laisser l'esclave savoir que des données lui sont envoyées, et non un autre esclave. Il le fait par adressage. La trame d'adresse est toujours la première trame après le bit de départ dans un nouveau message.

Le maître envoie l'adresse de l'esclave avec lequel il veut communiquer à chaque esclave connecté à celui-ci. Chaque esclave compare alors l'adresse envoyée par le maître à la sienne. Si l'adresse correspond, il renvoie un bit ACK basse tension au maître. Si l'adresse ne correspond pas, l'esclave ne fait rien et la ligne SDA reste haute.

C'est ainsi que le mot **wire.detect** détecte les périphériques connectés au bus i2c.

On peut connecter plusieurs périphériques différents sur le bus i2c. On ne peut pas connecter un même périphérique en plusieurs exemplaires sur le même bus i2c.

Définition des ports GPIO pour I2C

Le paramétrage des ports GPIO pour le bus I2C est très simple:

```
\ activate the wire vocabulary
wire
\ start the I2C interface using pin 21 and 22 on ESP32 DEVKIT V1
\ with 21 used as sda and 22 as scl.
21 22 wire.begin
```

Protocoles du bus I2C

Le dialogue se fait uniquement entre un maître et un esclave. Ce dialogue est toujours initié par le maître (condition Start): le maître envoie sur le bus I2C l'adresse de l'esclave avec qui il veut communiquer.

Le dialogue est toujours terminé par le maître (condition Stop).

Le signal d'horloge (SCL) est généré par le maître.

Détection d'un périphérique I2C

Cette partie sert à détecter la présence d'un périphérique connecté au bus I2C.

Vous pouvez compiler ce code pour tester la présence de modules connectés et actifs sur le bus I2C.

```
\ active le vocabulaire wire
wire
\ démarre l'interface I2C utilisant pin 21 et 22 sur ESP32 DEVKIT V1
\ avec 21 pour sda et 22 pour scl.
21 22 wire.begin

: spaces ( n -- )
  for
    space
  next
;

: .## ( n -- )
  <# # # #> type
;

\ not all bitpatterns are valid 7bit i2c addresses
: Wire.7bitaddr? ( a -- f )
  dup $07 >=
  swap $77 <= and
;

```



```

: Wire.detect ( -- )
  base @ >r hex
  cr
  ."      00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f"
  $80 $00 do
    i $0f and 0=
    if
      cr i .## ." : "
    then
      i Wire.7bitaddr? if
        i Wire.beginTransaction
        -1 Wire.endTransmission 0 =
        if
          i .## space
        else
          ." -- "
        then
        else
          2 spaces
        then
      loop
      cr r> base !
    ;

```

Ici, l'exécution du mot **Wire.detect** indique la présence du périphérique d'affichage OLED à l'adresse hexadécimale 3C:

```

Wire.detect
      00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
00 :
10 : -- -- -- -- -- -- -- -- -- -- -- -- -- --
20 : -- -- -- -- -- -- -- -- -- -- -- -- -- --
30 : -- -- -- -- -- -- -- -- -- -- -- 3c -- --
40 : -- -- -- -- -- -- -- -- -- -- -- -- -- --
50 : -- -- -- -- -- -- -- -- -- -- -- -- -- --
60 : -- -- -- -- -- -- -- -- -- -- -- -- -- --
70 : -- -- -- -- -- -- -- -- --

```

Ici, nous avons détecté un module à l'adresse hexadécimale 3c. C'est cette adresse que nous utiliserons pour nous adresser à ce module..... @TODO : à terminer

Programmer en assembleur XTENSA

Préambule

Pour ceux qui ne connaissent pas le langage assembleur, c'est la couche de plus bas niveau en programmation. En assembleur, on s'adresse directement au processeur.

C'est aussi un langage difficile, peu lisible. Mais en contrepartie, les performances sont exceptionnelles.

On programme en assembleur :

- quand il n'y a plus d'autre solution pour accéder à certaines fonctionnalités d'un processeur;
- pour rendre certaines parties de programme plus rapides. Le code généré par un assembleur est le plus rapide !
- pour le fun. La programmation en assembleur est un défi intellectuel ;
- parce qu'aucun langage évolué ne sait tout faire. Parfois, on peut programmer en assembleur des fonctions trop complexes à écrire dans un autre langage.

A titre d'exemple, voici le code du décodage de Huffman réalisé en assembleur XTENSA :

```
/* input in t0, value out in t1, length out in t2 */
srl t1, t0, 6
li t3, 3
beq t3, t4, 2f
li t2, 2
andi t3, t0, 0x20
beq t3, r0, 1f
li t2, 3
andi t3, t0, 0x10
beq t3, r0, 1f
li t2, 4
andi t3, t0, 0x08
beq t3, r0, 1f
li t2, 5
andi t3, t0, 0x04
beq t3, r0, 1f
li t2, 6
andi t3, t0, 0x02
beq t3, r0, 1f
li t2, 7
andi t3, t0, 0x01
```

```

    beq t3, r0, 1f
    li t2, 8
    b 2f
    li t1, 9
1: /* length = value */
    move t1, t2
2: /* done */

```

Depuis la version 7.0.7.4, ESP32forth intègre un assembleur XTENSA complet. Cet assembleur utilise une notation infixée :

```

\ en assembleur conventionnel:
\   andi t3, t0, 0x01

\ en assembleur XTENSA avec ESP32forth:
    a3 a0 $01 ANDI,

```

ESP32forth est le **tout premier langage de programmation de haut niveau** pour ESP32 qui intègre un assembleur XTENSA.

Cette particularité permet au programmeur de définir ses macros d'assemblage.

Tout mot écrit en langage assembleur XTENSA depuis ESP32forth est immédiatement utilisable dans n'importe quelle définition en langage FORTH.

Compiler l'assembleur XTENSA

Depuis la version 7.0.7.15, ESP32forth propose l'assembleur XTENSA comme option. Pour compiler cette option :

- ouvrir le dossier **optional** dans le dossier où vous avez décompressé le fichier ZIP de la version ESP32forth
- copier le fichier **assemblers.h** vers le dossier racine contenant le fichier **ESP32forth.ino**
- lancez ARDUINO IDE, compilez **ESP32forth.ino** et téléverser vers la carte ESP32

Si tout c'est bien passé, vous accéder à l'assembleur XTENSA en tapant une seule fois :

```
xtensa-assembler
```

Pour vérifier la bonne disponibilité du jeu d'instructions XTENSA :

```
assembler xtensa vlist
```

Programmer en assembleur

Afin de bien comprendre ce qui a été affirmé précédemment, voici une définition proposée comme exemple par Brad NELSON :

```
\ exemple proposé par Brad NELSON
code my2*
  a1 32 ENTRY,
  a8 a2 0 L32I.N,
  a8 a8 1 SLLI,
  a8 a2 0 S32I.N,
  RETW.N,
end-code
```

On vient de définir le mot **my2*** qui a exactement la même action que le mot **2***.
L'assemblage du code est immédiat. On peut donc tester notre définition de **my2*** depuis le terminal :

```
--> 3 my2*
ok
6 --> 21 my2*
ok
6 42 -->
```

Cette possibilité de tester immédiatement un code assemblé permet de le tester in situ. Si on est amené à écrire un code un peu complexe, il sera aisé de le découper en fragments et tester chaque partie de ce code depuis l'interpréteur de ESP32forth.

Le code assembleur XTENSA est placé après le mot à définir. C'est la séquence **code my2*** qui crée le mot **my2***.

Les lignes suivantes contiennent le code assembleur XTENSA. La définition en assembleur s'achève avec l'exécution de **end-code**.

Résumé des instructions de base

Liste des instructions de base incluses dans toutes les versions de l'architecture Xtensa. Le reste de cette section donne un aperçu des instructions de base.

Load / chargement

```
L8UI, L16SI, L16UI, L32I, L32R,
```

Store / stockage

```
S8I, S16I, S32I,
```

Mise en ordre mémoire

```
MEMW, EXTW,
```

Sauts

```
CALL0, CALLX0, RET, J, JX,
```

Branchement conditionnel

```
BALL, BNALL, BANY, BNONE, BBC, BBCI, BBS, BBSI, BEQ, BEQI, BEQZ,
BNE,
    BNEI, BNEZ, BGE, BGEI, BGEU, BGEUI, BGEZ, BLT, BLTI, BLTU, BLTUI,
BLTZ,
```

Déplacement

```
MOVI, MOVEQZ, MOVGEZ, MOVLTZ, MOVNEZ,
```

Arithmétique

```
ADDMI, ADD, ADDX2, ADDX4, ADDX8, SUB, SUBX2, SUBX4, SUBX8, NEG,
ABS,
```

Logique binaire

```
AND, OR, XOR,
```

Décalage

```
EXTUI, SRLI, SRAI, SLLI, SRC, SLL, SRL, SRA, SSL, SSR, SSAI,
SSA8B, SSA8L,
```

Contrôle processeur

```
RSR, WSR, XSR, RUR, WUR, ISYNC, RSYNC, ESYNC, DSYNC, NOP,
```

Un dés-assembleur en prime

Un assembleur, c'est très bien. Un code facile à intégrer aux définitions FORTH, c'est merveilleux. Mais disposer d'un dés-assembleur XTENSA, alors là, c'est royal !

Reprenons la définition de **my2*** précédemment assemblée. Il est facile d'en obtenir le désassemblage :

```
' my2* cell+ @ 20 disasm
\ affiche:
\ 1074338656 -- a1 32 ENTRY,          -- 004136
\ 1074338659 -- a8 a2 0 L32I.N,        -- 0288
\ 1074338661 -- a8 a8 1 SLLI,          -- 1188F0
\ 1074338664 -- a8 a2 0 S32I.N,        -- 0289
\ 1074338666 -- RETW.N,               -- F01D
\ 1074338668 -- .....
```

Le code de notre mot **my2*** n'est accessible que par indirection dont l'adresse est placée dans le champ des paramètres.

Chaque ligne affiche :

- l'adresse du code assemblé

- le code désassemblé à cette adresse sur 2 ou 3 octets
- le code hexadécimal correspondant au code désassemblé

Le dés-assembleur peut aussi agir sur l'ensemble du code déjà compilé ou assemblé.

Voyons le code du mot **2***:

```
' 2* @ 20 disasm
\ display:
\ 1074606252 -- a12 a3 0 L32I.N,          -- 03C8
\ 1074606254 -- a5 a5 1 SLLI,             -- 1155F0
\ 1074606257 -- a15 a12 0 L32I.N,         -- 0CF8
\ 1074606259 -- a3 a3 4 ADDI.N,           -- 334B
\ 1074606261 -- 1074597318 J,            -- F74346
```

Le désassemblage indique que le code mène à un saut inconditionnel **1074597318 J**,. Il est facile de poursuivre le désassemblage vers cette nouvelle adresse :

```
1074597318 20 disasm
\ display:
\ 1074597318 -- a15 JX,                    -- 000FA0
\ 1074597321 -- a10 64672 L32R,            -- FCA0A1
\ 1074597324 -- a5 a7 1 S32I,             -- 016752
\ 1074597327 -- 1074633168 CALL8,         -- 08C025
\ 1074597330 -- a12 a3 0 L32I,            -- 0023C2
\ 1074597333 -- a2 a7 4 ADDI,             -- 04C722
\ 1074597336 .....
```

Premiers pas en assembleur XTENSA

Préambule

Le code assembleur n'est pas portable dans un autre environnement, ou alors au prix d'énormes efforts de compréhension et d'adaptation du code assemblé.

Une version FORTH n'est pas complète si elle n'a pas d'assembleur.

La programmation en assembleur n'est pas une obligation. Mais dans certains cas, créer une définition en assembleur peut s'avérer bien plus aisé qu'une version en langage C ou en langage FORTH pur.

Mais surtout, une définition écrite en assembleur aura une rapidité d'exécution inégalable.

Nous allons voir, à partir d'exemples très simples et très courts comment maîtriser la programmation de définitions FORTH écrites en assembleur Xtensa.

Invocation de l'assembleur Xtensa

Au démarrage de ESP32forth, impossible de définir des mots en assembleur Xtensa sans invoquer le mot **xtensa-assembler**. Ce mot va charger le contenu du vocabulaire **xtensa**. Ce mot ne doit être invoqué qu'une seule fois au démarrage de ESP32forth et avant toute définition d'un mot en code xtensa:

```
forth
DEFINED? code invert [IF] xtensa-assembler [THEN]
```

Maintenant, si on tape **order**, ESP32forth affiche:

```
xtensa >> asm >> FORTH
```

C'est cet ordre de vocabulaires qu'il faudra respecter quand on veut définir un nouveau mot en assembleur Xtensa à l'aide des mots de définition **code** et **end-code**.

Xtensa et la pile FORTH

Le processeur Xtensa dispose de 16 registres, a0 à a15. En réalité, il y a 64 registres, mais on ne peut accéder qu'à une fenêtre de 16 registres parmi ces 64 registres, accessibles dans l'intervalle 00..15.

Le registre a2 contient le pointeur de pile FORTH.

A chaque empilement de valeur, le pointeur de pile est incrémenté de quatre unités :

```
SP@ .    \ affiche 1073632236
1
SP@ .    \ display 1073632240
```

```

2
SP@ . \ display 1073632244
drop drop
SP@ . \ 1073632236

```

Voici comment on pourrait réécrire ce mot **SP@** en assembleur Xtensa :

```

\ récupère Pointeur de Pile SP - equivalent à SP@
code mySP@
    a1 32      ENTRY,
    a8 a2      MOV.N, \ copie contenu de a2 dans a8
    a2 a2 4    ADDI,  \ incremente a2
    a8 a2 0    S32I.N, \ copie a8 dans adresse pointée par a2+0
                    RETW.N,
end-code

```

Testons ce nouveau mot **mySP@** :

```

mySP@ .
\ affiche 1073632240
SP@ .
\ affiche 1073632240

```

Ecriture d'une macro instruction Xtensa

Dans notre définition du mot **mySP@**, la séquence **a2 a2 4 ADDI**, incrémente de quatre unités le pointeur de pile. Sans cette incrémentation, impossible de renvoyer une valeur au sommet de la pile FORTH. Avec FORTH, nous allons écrire une macro qui automatise cette opération.

Pour commencer, nous allons étendre le vocabulaire **asm** :

```

asm definitions

: macro:
  :
;

```

Notre définition **macro:** est redondante avec **:** mais à l'avantage de rendre ensuite le code FORTH un peu plus lisible quand on définit une macro-instruction qui étendra le vocabulaire **xtensa**:

```

xtensa definitions

macro: sp++,
    a2 a2 4    ADDI,
;

```


Avec cette nouvelle macro-instruction **sp++**, nous pouvons réécrire la définition de **mySP@**:

```
forth definitions
asm xtensa

\ get Stack Pointer SP - equivalent for SP@
code mySP@
  a1 32      ENTRY,
  a8 a2      MOV.N, \ copy content of a2 in a8
    sp++,
  a8 a2 0    S32I.N, \ copy a8 in address pointed by a2+0
              RETW.N,
end-code
```

Il est parfaitement possible d'intégrer une macro dans une autre. Dans le code de **mySP@**, la ligne de code **a8 a2 0 S32I.N**, copie le contenu du registre a8 à l'adresse pointée par a2. Voici cette nouvelle macro instruction :

```
xtensa definitions

\ increment Stack Pointer and store content of ar in addr pointed by
Stack Pointer
macro: arPUSH, { ar -- }
  sp++,
  ar a2 0 S32I.N,
;
```

Cette macro instruction utilise une variable locale **ar**. On aurait pu s'en passer, mais l'intérêt de cette variable est que le code de la macro est plus lisible.

Voici le code de **mySP@** avec cette macro-instruction.

```
forth definitions
asm xtensa

\ get Stack Pointer SP - equivalent to SP@
code mySP@3
  a1 32      ENTRY,
  a8 a2      MOV.N,
  a8 arPUSH,
              RETW.N,
end-code
```

Complétons notre liste de macro instructions :

```
xtensa definitions
```

```

\ décrémente pointeur de pile
macro: sp--,    ( -- )
    a2 a2 -4    ADDI,
;

\ Store content of addr pointed by Stack Pointer in ar and decrement
Stack Pointer
macro: arPOP,    { ar -- }
    ar a2 0      L32I.N,
    sp--,
;

```

Avec ces nouvelles macros, réécrivons **swap**:

```

forth definitions
asm xtensa

code mySWAP
    a1 32      ENTRY,
    a9 arPOP,
    a8 arPOP,
    a9 arPUSH,
    a8 arPUSH,
                RETW.N,
end-code

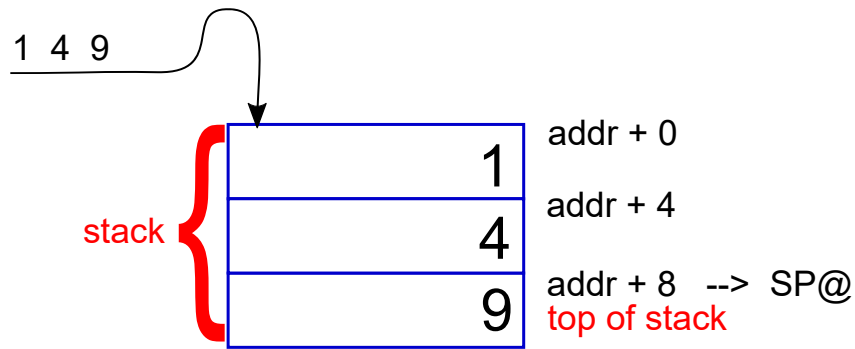
17 24 mySWAP

```

Gestion de la pile FORTH en assembleur Xtensa

La position du pointeur de pile FORTH est accessible par **SP@**. L'empilement d'un entier 32 bits (taille par défaut pour ESP32forth) incrémente ce pointeur de pile de quatre unités.

Nous avons abordé la manière de gérer l'incrémementation ou décrémentation de ce pointeur de pile au travers des macro-instructions **sp++**, et **sp--**,. Ces macro-instructions déplacent le pointeur de pile de quatre unités.



Ici, on a empilé trois valeurs, **1**, **4** et **9**. A chaque empilement, le pointeur de pile est incrémenté automatiquement. En assembleur Xtensa, le pointeur de pile se retrouve dans le registre a2. Nous avons vu, que nous pouvons manipuler le contenu de ce registre avec les macro-instructions **sp++**, et **sp--**,. La manipulation de ce registre a une action directe sur le pointeur de pile géré par ESP32forth.

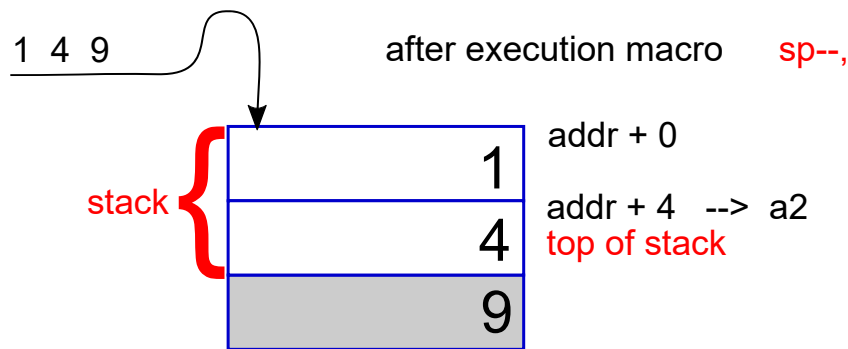
Voici comment nous avons réécrit en assembleur le mot **+** en manipulant le pointeur de pile au travers de nos macro-instructions **arPOP**, et **arPUSH**, :

```
code my+
  a1 32      ENTRY,
  a7 arPOP,
  a8 arPOP,
  a7 a8 a9    ADD,
  a9 arPUSH,
              RETW.N,
end-code
```

Il y a un autre moyen pour récupérer les données depuis la pile en utilisant l'instruction **L32I.N**,. Cette instruction utilise un index immédiat :

```
code my+
  a1 32 ENTRY,
      sp--,
  a7 a2 0    L32I.N,
  a8 a2 1    L32I.N,
  a7 a8 a9    ADD,
  a9 a2 0    S32I.N,
  RETW.N,
end-code
```

Avant de récupérer les données de la pile, on décrémente le pointeur de pile avec notre macro-instruction **sp--**,. De cette manière, le pointeur recule de 4 unités.



Mais ce n'est pas parce que le pointeur recule que les données préalablement empilées disparaissent. Voyons en détail cette ligne de code :

```
a7 a2 0    L32I.N,
```

Cette instruction charge le registre a7 avec le contenu de l'adresse pointée par $(a2)+n*4$. Ici, n vaut 0. Cette instruction va mettre la valeur 4 dans notre registre a7.

Voyons la ligne suivante :

```
a8 a2 1    L32I.N,
```

Le registre a8 est chargé avec le contenu pointé par $(a2)+1*4$. Cette instruction met la valeur 9 dans notre registre a8.

```
a9 a2 0    S32I.N,
```

Ici, le contenu du registre a9 est stocké à l'adresse pointée par $(a2)+1*0$. En fait, on écrase la valeur 4 avec le résultat de l'addition du contenu des registres a7 et a8.

Voyons un dernier exemple où nous traitons deux paramètres et en restituons deux sur la pile de données. Dans cet exemple, nous réécrivons le mot **/MOD** :

```
code my/MOD ( n1 n2 -- rem quot )
  a1 32      ENTRY,
  a7 arPOP,   \ diviseur dans a7
  a8 arPOP,   \ valeur à diviser dans a8
  a7 a8 a9    REMS, \ a9 = a8 MOD a7
  a9 arPUSH,
  a7 a8 a9    QUOS, \ a9 = a8 / a7
  a9 arPUSH,
              RETW.N,
end-code

5 2 my/MOD . . \ display 2 1
-5 -2 my/MOD . . \ display 2 -1
```

Dans le mot **my/MOD** on exploite les mêmes données n1 et n2 placées respectivement dans les registres a8 et a7. Ce sont ensuite les instructions **REMS**, et **QUOT**, qui permettent de calculer les résultats restitués par **my/MOD**.

Efficacité des mots écrits en assembleur XTENSA

Dans notre tout dernier exemple ci-dessus, nous avons réécrit le mot **/MOD**. La question à se poser est: "est-ce que le mot **my/MOD** est réellement plus rapide en exécution que le mot **/MOD**?".

Pour ce faire, nous allons utiliser le mot **measure**: dont le code FORTH est expliqué dans le chapitre *Mesurer le temps d'exécution d'un mot FORTH*.

```
: test1
  1000000 for
    5 2 /MOD
    drop drop
  next
;

: test2
  1000000 for
    5 2 my/MOD
    drop drop
  next
;

measure: test1 \ display: execution time: 0.856sec.
measure: test2 \ display: execution time: 0.600sec.
```

Les mots **test1** et **test2** sont semblables, à la différence que **test2** exécute **my/MOD**. Sur 1 million d'itérations, le gain de temps s'élève à 0.144 seconde. C'est peu, mais le ratio semble quand même significatif.

A contrario, on constate que le langage FORTH est très rapide en temps d'exécution.

Boucles et branchements en assembleur XTENSA

L'instruction LOOP en assembleur XTENSA

La boucle LOOP en assembleur XTENSA fonctionne en utilisant l'instruction **LOOP** pour indiquer au processeur de répéter un bloc d'instructions jusqu'à ce qu'un compteur spécifié atteigne zéro. La boucle est initialisée en définissant la valeur initiale du compteur, puis en exécutant l'instruction **LOOP** avec cette valeur en argument. À chaque itération de la boucle, le compteur est décrémenté de 1 jusqu'à ce qu'il atteigne zéro, moment où la boucle s'arrête. En assembleur classique :

```
; Initialization of the counter to 10
MOVI a0, 10

; Beginning of the LOOP loop
loop:
; Instruction(s) to repeat
...
; Decrement the counter and test the stop condition
LOOP a0, loop
```

Ici, la boucle LOOP répète les instructions situées entre **loop:** et **LOOP a0**, boucle 10 fois, en décrémentant le compteur a0 à chaque itération. Lorsque le compteur atteint zéro, la boucle s'arrête.

Quand le processeur XTENSA rencontre l'instruction **LOOP**, il initialise trois registres spéciaux :

- **LCOUNT** \leftarrow **AR[s] - 1**
Le registre spécial LCOUNT est initialisé avec le contenu du registre as, ici a0 dans notre exemple, décrémenté de une unité. Quand le compteur atteint la valeur 0, l'instruction LOOP achève la boucle;
- **LBEG** \leftarrow **PC + 3**
Le registre spécial LBEG contient l'adresse de début de la boucle LOOP en cours d'exécution. Cette adresse est définie par l'instruction LOOP.
- **LEND** \leftarrow **PC + (024 | imm8) + 4** Le registre spécial LEND contient l'adresse de fin de la boucle LOOP en cours d'exécution. Cette adresse est définie par l'instruction LOOP.

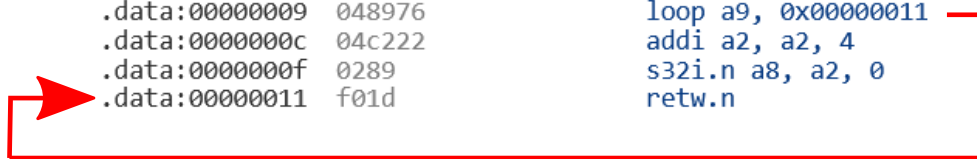
En assembleur XTENSA, l'instruction LOOP admet deux paramètres:

```
LOOP as, label
```

label correspond à un décalage 8 bits après l'instruction **LOOP**. On ne peut pas répéter un code de plus de 256 octets de longueur.

Voici un code XTENSA désassemblé utilisant une boucle LOOP :

.data:00000000	004136	entry a1, 32
.data:00000003	01a082	movi a8, 1
.data:00000006	04a092	movi a9, 4
.data:00000009	048976	loop a9, 0x00000011
.data:0000000c	04c222	addi a2, a2, 4
.data:0000000f	0289	s32i.n a8, a2, 0
.data:00000011	f01d	retw.n



Le désassembleur indique une adresse de branchement. En réalité, le code assemblé ne contient que cet offset indiqué par le label sous forme de valeur positive 8 bits.

Gérer une boucle en assembleur XTENSA avec ESP32forth

Le langage FORTH ne sait pas résoudre une référence vers l'avant. Sauf à tatonner, il est difficile d'exploiter l'instruction **LOOP**, sans trouver une astuce.

Définition de macro-instructions de gestion de boucle

Pour utiliser facilement l'instruction **LOOP**, on va définir deux macro instructions, respectivement **For**, et **Next**, dont voici le code en langage FORTH :

```
: For, { as n -- }
  as n MOVI,
  as 0 LOOP,
  chere 1- to LOOP_OFFSET
;

: Next, ( -- )
  chere LOOP_OFFSET - 2 -
  LOOP_OFFSET [ internals ] ca! [ asm xtensa ]
;
```

La macro instruction **For**, accepte les mêmes paramètres que l'instruction **LOOP**, :

```
as n For,
```

- as est le registre qui contient le nombre d'itérations de la boucle ;
- n est le nombre d'itérations.

Utilisation des macros For, et Next,

On définit un mot **myLOOP** pour tester l'instruction **LOOP**, par l'intermédiaire des macro instructions **For**, **Next**, :

```
code myLOOP ( n -- n' )
  a1 32          ENTRY,
  a8 1           MOVI,
  a9 4           For,           \ LOOP start here
    a8 a8 1      ADDI,
    a8           arPUSH,       \ push result on stack
  Next,
                                RETW.N,
end-code
```

Le registre a8 est initialisé avec la valeur 1. La boucle **For**, **Next**, effectue une incrémentation du contenu de a8 et empile son contenu. Voici ce que donne l'exécution de **MyLOOP** :

```
ok
--> myLoop
ok
2 3 4 5 -->
```

ATTENTION: si le nombre d'itérations est nul, le nombre d'itérations passe à 232.

Les instructions de branchement en assembleur XTENSA

L'assembleur XTENSA dans le vocabulaire **xtensa** dispose de plusieurs types d'instructions de branchement :

- les branchements utilisant des flags booléens définis dans le registre spécial **BR** : **BF**, **BT**,
- les branchements effectuant des tests sur les registres : **BALL**, **BANY**, **BBC**, **BBS**, **BEQ**, **BGE**, **BLT**, **BNE**, **BNONE**,

C'est cette seconde catégorie de branchements qui nous intéresse.

Définition de macros de branchement

L'assembleur xtensa de ESP32forth ne dispose pas de mécanisme de gestion de labels comme c'est le cas pour un assembleur classique. Pour être efficace, la gestion de labels doit fonctionner en plusieurs étapes s'il faut résoudre des branchements vers l'avant. Ceci est incompatible avec le fonctionnement du langage FORTH qui compile ou assemble en une seule passe.

On lève cette difficulté en définissant deux macros instructions, **If**, et **Then**,, qui vont gérer ces branchements vers l'avant :

```
: If, ( -- BRANCH_OFFSET )
```



```

    chere 1-
;

: Then, { BRANCH_OFFSET -- }
    chere BRANCH_OFFSET - 2 -
    BRANCH_OFFSET [ internals ] ca! [ asm xtensa ]
;

```

La macro instruction doit être précédée d'une autre macro instruction. Pour notre premier test, on définit la macro **<**, qui assemblera un branchement non résolu :

```

: <, ( as at -- )
    0 BGE,
;

```

Utilisation de ces macros dans notre premier exemple :

```

code my< ( n1 n2 -- fl ) \ fl=1 if n1 < n2
    a1 32          ENTRY,
    a8            arPOP,          \ a8 = n2
    a9            arPOP,          \ a9 = n1
    a7 0          MOVI,          \ a7 = 1
    a8 a9 <, If,
        a7 1      MOVI,          \ a7 = 0
    Then,
    a7            arPUSH,
                RETW.N,
end-code

```

Syntaxe des macro instructions de branchement

Dans notre exemple, nous avons utilisé la macro instruction **<**, qui est associé à l'instruction de branchement **BGE**, et dont la signification est: "Branch if Greater Than or Equal" (Branchement si plus grand ou égal). Normalement, il faudrait traduire par "**>=**". Pourquoi a-t-on utilisé "**<**"?

C'est parce que notre macro instruction **If, Then**, a une logique inverse à celle du branchement à effectuer. Le code placé entre **If, Then**, s'exécutera si la condition requise n'est pas valide. Voici le tableau qui récapitule cette logique inversée expliquant le choix du nom de ces macro instructions utilisées avant **If, Then**, :

XTENSA branch instruction			Macro
BEQ	Branch if Equal	AR[s] = AR[t]	<> ,
BGE	Branch if Greater Than or Equal	AR[s] ≥ AR[t]	< ,
BLT	Branch if Less Than	AR[s] < AR[t]	>= ,
BNE	Branch if Not Equal	AR[s] ≠ AR[t]	= ,

Revenons à notre exemple d'assemblage **my<**. Voici ce que donne l'exécution du mot **my<** :

```
10 20 my< .      \ affiche: 1
20 20 my< .      \ affiche: 0
20 10 my< .      \ affiche: 0
-5 35 my< .      \ affiche: 1
-10 -3 my< .     \ affiche: 1
-3 -10 my< .     \ affiche: 0
```

Nous constatons que cette logique inversée est respectée.

Une fois cette logique comprise, on peut définir une nouvelle macro-instruction **>=**, :

```
: >=, ( as at -- )
    0 BLT,
;
```

Et test de cette macro-instruction :

```
code my>= ( n1 n2 -- fl )    \ fl=1 if n1 < n2
    a1 32                    ENTRY,
    a8                        arPOP,          \ a8 = n2
    a9                        arPOP,          \ a9 = n1
    a7 0                      MOVI,          \ a7 = 1
    a8 a9 >=, If,
        a7 1                  MOVI,          \ a7 = 0
    Then,
    a7                        arPUSH,
                                RETW.N,
end-code

10 20 my>= .      \ display: 0
20 20 my>= .      \ display: 1
20 10 my>= .      \ display: 1
-5 35 my>= .      \ display: 0
-10 -3 my>= .     \ display: 0
-3 -10 my>= .     \ display: 1
```

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

- **ESP32 Forth** site en deux langues (français, anglais) avec plein d'exemples
<https://esp32.arduino-forth.com/>

GitHub

- **Ueforth** ressources maintenues par Brad NELSON. Contient tous les fichiers sources en Forth et en langage C de ESP32forth
<https://github.com/flagxor/ueforth>
- **ESP32forth** codes sources et documentation pour ESP32forth. Ressources maintenues par Marc PETREMANN
<https://github.com/MPETREMANN11/ESP32forth>
- **ESP32forthStation** ressources maintenues par Ulrich HOFFMAN. Stand alone Forth computer with LillyGo TTGO VGA32 single board computer and ESP32forth .
<https://github.com/uho/ESP32forthStation>
- **ESP32Forth** ressources maintenues par F. J. RUSSO
<https://github.com/FJRusso53/ESP32Forth>
- **esp32forth-addons** ressources maintenues par Peter FORTH
<https://github.com/PeterForth/esp32forth-addons>
- **Esp32forth-org** Code repository for members of the Forth2020 and ESp32forth groups
<https://github.com/Esp32forth-org>

-

Index lexical

analogRead.....	115	heure réelle.....	107	structures.....	41, 43, 56
ansi.....	56	HEX.....	46	Tera Term.....	65
BASE.....	46	HOLD.....	47	thru.....	73
bg.....	28	include.....	77	to.....	33
c!.....	25	interval.....	99	transmettre gros fichier.....	81
c@.....	25	liste des fichiers.....	77	value.....	26
canaux ADC.....	114	liste fichiers.....	77	variable.....	25
Codage ANSI.....	28	load.....	73	variables locales.....	32
constant.....	26	ls.....	77	voir contenu fichier.....	78
conversion analogique		mémoire.....	25	wipe.....	72
numérique.....	120	ms-ticks.....	107	Wire.detect.....	127
Couleurs de texte.....	28	normal.....	28	xtensa-assembler.....	135
DECIMAL.....	46	oled.....	56, 123	;	22
defPin:.....	84	page.....	28	:	22
démarrage automatique.....	63	pile de retour.....	24	."	50
drop.....	24	position de l'affichage.....	28	{	32
dup.....	24	pseudo répertoire.....	77	}	32
editor.....	56, 71	renommer fichiers.....	78	@	25
effacer fichier.....	78	rerun.....	99	#	47
EMIT.....	49	rm.....	78	#>	47
fg.....	28	S"	50	#S	47
flush.....	73	SPACE.....	51	+to	33
forget.....	22	SPIFFS.....	77	<#	47
gpio_set_intr_type.....	89	struct.....	41		