

Le grand livre

de ESP32forth

version 1.16 - 2 janvier 2024



Auteur(s)

- Marc PETREMANN petremann@arduino-forth.com

Collaborateur(s)

- Bob EDWARDS
- Vaclav POSELT
- Thomas SCHREIN

Table des matières

Auteur(s).....	1
Collaborateur(s).....	1
Introduction.....	12
Aide à la traduction.....	12
Découverte de la carte ESP32.....	13
Présentation.....	13
Les points forts.....	13
Les entrées/sorties GPIO sur ESP32.....	14
Périphériques de l'ESP32.....	15
Installer ESP32Forth.....	17
Télécharger ESP32forth.....	17
Compilation et installation de ESP32forth.....	17
Paramètres pour ESP32 WROOM.....	19
Lancer la compilation.....	19
Résoudre l'erreur de connexion au téléchargement.....	21
Pourquoi programmer en langage FORTH sur ESP32?.....	23
Préambule.....	23
Limites entre langage et application.....	24
C'est quoi un mot FORTH?.....	24
Un mot c'est une fonction?.....	24
Le langage FORTH comparé au langage C.....	25
Ce que FORTH permet de faire par rapport au langage C.....	26
Mais pourquoi une pile plutôt que des variables?.....	27
Êtes-vous convaincus?.....	27
Existe-t-il des applications professionnelles écrites en FORTH?.....	27
Utiliser les nombres avec ESP32Forth.....	30
Les nombres avec l'interpréteur FORTH.....	30
Saisie des nombres avec différentes base numérique.....	31
Changement de base numérique.....	32
Binaire et hexadécimal.....	32
Taille des nombres sur la pile de données FORTH.....	34
Accès mémoire et opérations logiques.....	36
Un vrai FORTH 32 bits avec ESP32Forth.....	38
Les valeurs sur la pile de données.....	38
Les valeurs en mémoire.....	38
Traitement par mots selon taille ou type des données.....	39

Conclusion.....	40
Commentaires et mise au point.....	42
Ecrire un code FORTH lisible.....	42
Indentation du code source.....	43
Les commentaires.....	44
Les commentaires de pile.....	44
Signification des paramètres de pile en commentaires.....	45
Commentaires des mots de définition de mots.....	45
Les commentaires textuels.....	46
Commentaire en début de code source.....	46
Outils de diagnostic et mise au point.....	47
Le décompilateur.....	47
Dump mémoire.....	48
Moniteur de pile.....	48
Dictionnaire / Pile / Variables / Constantes.....	50
Étendre le dictionnaire.....	50
Gestion du dictionnaire.....	50
Piles et notation polonaise inversée.....	51
Manipulation de la pile de paramètres.....	52
La pile de retour et ses utilisations.....	53
Utilisation de la mémoire.....	53
Variables.....	53
Constantes.....	54
Valeurs pseudo-constantes.....	54
Outils de base pour l'allocation de mémoire.....	55
Couleurs de texte et position de l'affichage sur terminal.....	56
Codage ANSI des terminaux.....	56
Coloration du texte.....	57
Position de l'affichage.....	58
Les variables locales avec ESP32Forth.....	60
Introduction.....	60
Le faux commentaire de pile.....	60
Action sur les variables locales.....	61
Structures de données pour ESP32forth.....	64
Préambule.....	64
Les tableaux en FORTH.....	64
Tableau de données 32 bits à une dimension.....	64
Mots de définition de tableaux.....	65
Lire et écrire dans un tableau.....	65
Exemple pratique de gestion d'un écran virtuel.....	66

Gestion de structures complexes.....	69
Définition de sprites.....	71
Les nombres réels avec ESP32forth.....	74
Les réels avec ESP32forth.....	74
Precision des nombres réels avec ESP32forth.....	74
Constantes et variables réelles.....	75
Opérateurs arithmétiques sur les réels.....	75
Opérateurs mathématiques sur les réels.....	75
Opérateurs logiques sur les réels.....	76
Transformations entiers ↔ réels.....	76
Affichage des nombres et chaînes de caractères.....	78
Changement de base numérique.....	78
Définition de nouveaux formats d'affichage.....	79
Affichage des caractères et chaînes de caractères.....	81
Variables chaînes de caractères.....	83
Code des mots de gestion de variables texte.....	83
Ajout de caractère à une variable alphanumérique.....	85
Les vocabulaires avec ESP32forth.....	86
Liste des vocabulaires.....	86
Les vocabulaires essentiels.....	86
Liste du contenu d'un vocabulaire.....	87
Utilisation des mots d'un vocabulaire.....	87
Chainage des vocabulaires.....	88
Les mots à action différée.....	89
Définition et utilisation de mots avec defer.....	90
Définition d'une référence avant.....	90
Dépendance envers le contexte d'exploitation.....	91
Un cas pratique.....	92
Les mots de création de mots.....	94
Utilisation de does>.....	94
Exemple de gestion de couleur.....	95
Exemple, écrire en pinyin.....	96
Adapter les plaques d'essai à la carte ESP32.....	98
Les plaques d'essai pour ESP32.....	98
Construire une plaque d'essai adaptée à la carte ESP32.....	98
Alimenter la carte ESP32.....	101
Choix de la source d'alimentation.....	101
Alimentation par le connecteur mini-USB.....	101
Alimentation par le pin 5V.....	101

Démarrage automatique d'un programme.....	103
Installer et utiliser le terminal Tera Term sous Windows.....	105
Installer Tera Term.....	105
Paramétrage de Tera Term.....	105
Utilisation de Tera Term.....	108
Compiler du code source en langage Forth.....	109
Accéder à ESP32Forth par TELNET.....	111
Changer le nom DNS de la carte ESP32.....	111
Connexion aux cartes ESP32 par leur nom d'hôte.....	112
Gestion des fichiers sources par blocs.....	115
Les blocs.....	115
Ouvrir un fichier de blocs.....	115
Editer le contenu d'un bloc.....	116
Compilation du contenu des blocs.....	117
Exemple pratique pas à pas.....	117
Conclusion.....	118
Edition des fichiers sources avec VISUAL Editor.....	119
Editer un fichier source FORTH.....	119
Edition du code FORTH.....	119
Compilation du contenu des fichiers.....	120
RECORDFILE et gestion de projets FORTH.....	121
Enregistrer RECORDFILE dans le fichier autoexec.fs.....	121
Utiliser le contenu modifié du fichier autoexec.fs.....	123
Découpage d'un projet avec ESP32forth.....	123
Exemple de projet.....	123
La notion de boîte noire.....	125
Le système de fichiers SPIFFS.....	128
Accès au système de fichiers SPIFFS.....	128
Manipulation des fichiers.....	129
Organiser et compiler ses fichiers sur la carte ESP32.....	130
Edition et transmission des fichiers source.....	130
Organiser ses fichiers.....	130
Conclusion.....	131
Edition et gestion des fichiers sources pour ESP32forth.....	133
Les éditeurs de fichiers texte.....	133
Utiliser un IDE.....	134
Stockage sur GitHub.....	136
Quelques bonnes pratiques.....	136
Le fichier main.fs.....	137

Gérer un feu tricolore avec ESP32.....	139
Les ports GPIO sur la carte ESP32.....	139
Montage des LEDs.....	140
Gestion des feux tricolores.....	141
Conclusion.....	142
Acces direct aux registres GPIO.....	143
Utilisation des mots m! et m@.....	143
Le registre GPIO_OUT_REG.....	146
Les registres d'activation et désactivation.....	147
Acces direct aux registres GPIO.....	151
Utilisation des mots m! et m@.....	151
Le registre GPIO_OUT_REG.....	154
Les registres d'activation et désactivation.....	155
Les interruptions matérielles avec ESP32forth.....	159
Les interruptions.....	159
Montage d'un bouton poussoir.....	159
Consolidation logicielle de l'interruption.....	160
Informations complémentaires.....	161
Utilisation de l'encodeur rotatif KY-040.....	162
Présentation de l'encodeur.....	162
Montage de l'encodeur sur la plaque d'essai.....	163
Analyse des signaux de l'encodeur.....	164
Programmation de l'encodeur.....	165
Test de l'encodage.....	166
Incrémenter et décrémenter une variable avec l'encodeur.....	166
Clignotement d'une LED par timer.....	168
Débuter en programmation FORTH.....	168
Clignotement par TIMER.....	169
Les interruptions matérielles et logicielles.....	170
Utiliser les mots interval et rerun.....	170
Minuterie pour femme de ménage.....	173
Préambule.....	173
Une solution.....	173
Une minuterie en FORTH pour ESP32Forth.....	174
Gestion du bouton d'allumage lumière.....	175
Conclusion.....	177
Horloge temps réel logicielle.....	178
Le mot MS-TICKS.....	178
Gestion d'une horloge logicielle.....	178

Mesurer le temps d'exécution d'un mot FORTH.....	180
Mesurer la performance des définitions FORTH.....	180
Test de quelques boucles.....	181
Programmer un analyseur d'ensoleillement.....	182
Préambule.....	182
Le panneau solaire miniature.....	182
Récupération d'un panneau solaire miniature.....	182
Mesure de la tension du panneau solaire.....	183
Mesure du courant du panneau solaire.....	184
Abaissement de la tension du panneau solaire.....	184
Programmation de l'analyseur solaire.....	185
Gestion activation et désactivation d'un appareil.....	186
Déclenchement par interruption timer.....	188
Appareils commandés par le capteur d'ensoleillement.....	189
Gestion des sorties N/A (Numériques/Analogiques).....	191
La conversion numérique / analogique.....	191
La conversion N/A avec circuit R2R.....	191
La conversion N/A avec ESP32.....	192
Possibilités de la conversion N/A.....	193
Installation de la librairie OLED pour SSD1306.....	194
L'interface I2C sur ESP32.....	196
Introduction.....	196
Échange maître esclave.....	197
Adressage.....	198
Définition des ports GPIO pour I2C.....	199
Protocoles du bus I2C.....	199
Détection d'un périphérique I2C.....	199
L'afficheur OLED SSD1306.....	202
Choix d'un interface d'affichage.....	202
Documentation en ligne.....	203
Branchement de l'afficheur OLED SSD1306.....	203
Organisation de la mémoire.....	204
Organiser le projet SSD1306.....	205
Création du fichier main.fs.....	205
Création du fichier config.fs.....	205
Création du fichier oledTools.fs.....	206
Tester notre projet SSD1306.....	206
Exploiter le vocabulaire oled.....	208
Initialisation du bus I2C pour l'afficheur OLED SSD1306.....	208
Initialisation de l'affichage pour SSD1306.....	209

Étendre le vocabulaire oled.....	211
TEMPVS FVGIT.....	213
Romani non ustulo nulla.....	213
Romani horas et minuta.....	214
Haec omnia integramus pro ESP32forth.....	215
Ajouter la librairie SPI.....	217
Modifications du fichier ESP32forth.ino.....	217
Première modification.....	217
Seconde modification.....	218
Troisième modification.....	218
Quatrième modification.....	218
Communiquer avec le module d'affichage MAX7219.....	219
Répérage du port SPI sur la carte ESP32.....	220
Les connecteurs SPI sur le module d'affichage MAX7219.....	220
Couche logicielle du port SPI.....	221
Installation du client HTTP.....	222
Modification du fichier ESP32forth.ino.....	222
Test du client HTTP.....	223
Récupérer l'heure depuis un serveur WEB.....	226
Transmission et réception du temps depuis un serveur web.....	226
Comprendre la transmission par GET vers un serveur WEB.....	228
Transmission de données vers un serveur par GET.....	228
Les paramètres dans un URL.....	228
Passage de plusieurs paramètres.....	228
Gestion du passage de paramètres avec ESP32forth.....	229
Transmission de données vers un serveur WEB.....	231
Enregistrement des données côté serveur web.....	231
Protection de l'accès.....	231
Consulter les données enregistrées.....	232
Rajouter des données à transmettre.....	233
Conclusion.....	235
Synthèse sonore avec ESP32Forth.....	236
Synthèse sonore simple.....	236
Définition du tableau des fréquences sonores.....	236
Récupération de la fréquence d'une note de musique.....	237
Gestion de la durée des notes.....	238
Soutien d'une note.....	239
Création des notes musicales.....	239
Test des notes.....	240

Le vol du bourdon.....	241
Programmer en assembleur XTENSA.....	243
Préambule.....	243
Compiler l'assembleur XTENSA.....	244
Programmer en assembleur.....	244
Résumé des instructions de base.....	245
Un dés-assembleur en prime.....	246
Premiers pas en assembleur XTENSA.....	248
Préambule.....	248
Invocation de l'assembleur Xtensa.....	248
Xtensa et la pile FORTH.....	248
Ecriture d'une macro instruction Xtensa.....	249
Gestion de la pile FORTH en assembleur Xtensa.....	251
Efficacité des mots écrits en assembleur XTENSA.....	253
Boucles et branchements en assembleur XTENSA.....	254
L'instruction LOOP en assembleur XTENSA.....	254
Gérer une boucle en assembleur XTENSA avec ESP32forth.....	255
Définition de macro-instructions de gestion de boucle.....	255
Utilisation des macros For, et Next,.....	255
Les instructions de branchement en assembleur XTENSA.....	256
Définition de macros de branchement.....	256
Syntaxe des macro instructions de branchement.....	257
Définition et manipulation de registres.....	259
Définition de registres.....	259
Accès au contenu des registres.....	260
Manipulation des bits des registres.....	261
Définition de masques.....	261
Passer du langage C au langage FORTH.....	262
Le générateur de nombres aléatoires.....	265
Caractéristique.....	265
Procédure de programmation.....	266
Fonction RND en assembleur XTENSA.....	266
Le système de transmission LoRa.....	268
Câblage du transmetteur LoRa REYAX LR890.....	268
Le transmetteur LoRa pour ESP32.....	268
Sécurité des transmission LoRa.....	269
Test du transmetteur LoRa REYAX RYLR890.....	271
Environnement de test requis.....	271
Préparer la communication avec le transmetteur LoRa.....	271

Paramétrage du transmetteur LoRa REYAX RYLR890.....	274
Paramètres essentiels.....	274
ADDRESS Définit l'adresse du module.....	275
AT Test Disponibilité LoRa.....	276
BAND Réglage de la fréquence RF.....	276
CPIN Définit le mot de passe AES128 du réseau.....	276
CRFOP Sélectionne la puissance RF de sortie.....	277
FACTORY Règle tous les paramètres actuels sur les valeurs par défaut.....	277
IPR Règle le débit UART en bauds.....	278
MODE Sélectionne le mode de travail.....	278
NETWORKID Sélectionne l'ID réseau.....	279
PARAMETER définition des paramètres RF.....	279
RESET logiciel.....	281
SEND envoi de données à l'adresse désignée.....	281
VER pour demander la version du firmware.....	282
Codes de résultat d'erreur.....	282
Vectorisation des émissions de caractères.....	282
Comprendre la vectorisation en FORTH.....	283
La vectorisation dans ESP32Forth.....	284
Vectoriser type vers le port série UART2.....	284
Réécriture d'un listing complet.....	285
Paramétrage des transmetteurs LoRa.....	287
Détermination de l'adresse des transmetteurs LoRa.....	288
Communication entre deux transmetteurs LoRa REYAX RYLR890.....	290
Transmission depuis BOSS vers SLAV2.....	291
Interfacer une transmission LoRa avec ESP32Forth.....	293
Le programme côté transmetteur LoRa nommé BOSS.....	294
Réception et exécution des commandes FORTH par SLAV1.....	295
Exécution d'une commande reçue par LoRa.....	295
Boucle de gestion des transmissions LoRa.....	296
Un interface WEB simple pour ESP32Forth.....	299
Contenu détaillé des vocabulaires ESP32forth.....	303
Version v 7.0.7.15.....	303
FORTH.....	303
asm.....	304
bluetooth.....	305
editor.....	305
ESP.....	305
httpd.....	305
insides.....	305

internals.....	305
interrupts.....	306
ledc.....	306
oled.....	306
registers.....	306
riscv.....	306
rtos.....	307
SD.....	307
SD_MMC.....	307
Serial.....	307
sockets.....	307
spi.....	307
SPIFFS.....	307
streams.....	307
structures.....	307
tasks.....	307
telnetd.....	308
visual.....	308
web-interface.....	308
WiFi.....	308
Wire.....	308
xtensa.....	308
Annexe A – Sommaire des registres.....	310
GPIO registers.....	310
Ressources.....	314
En anglais.....	314
En français.....	314
GitHub.....	314

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

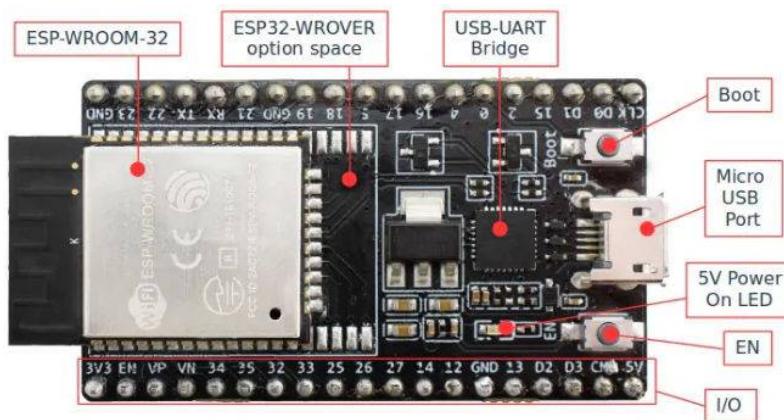


Figure 1: la carte de base a 38 connecteurs

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 quelles 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.

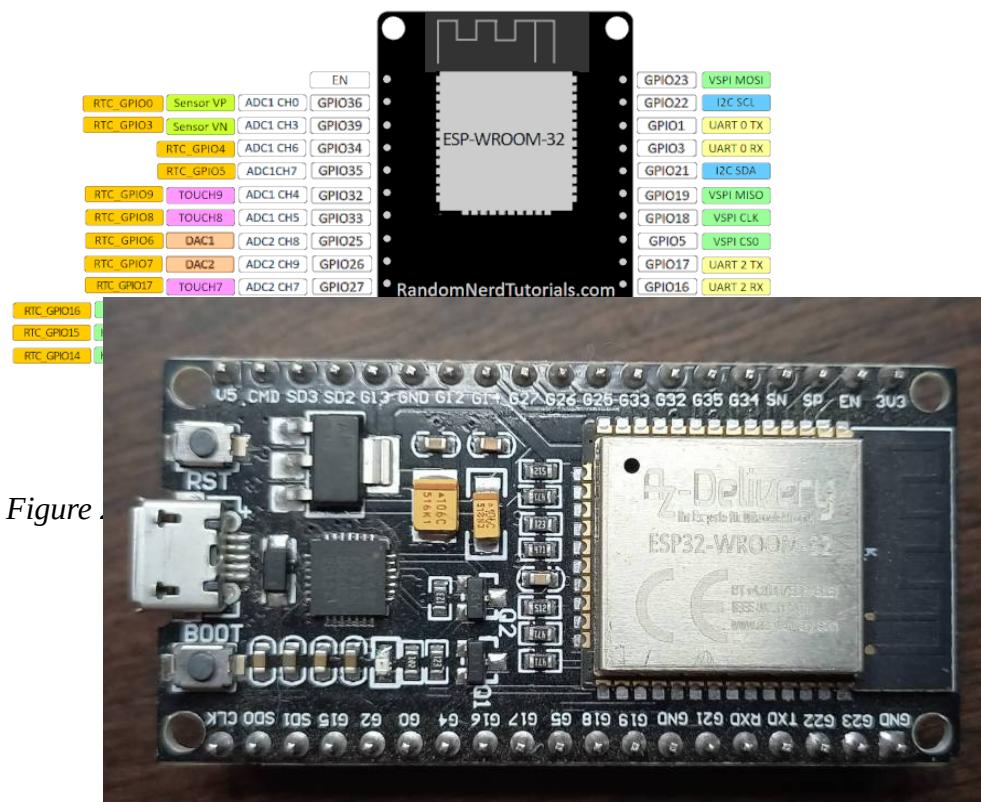


Figure .

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.

Installer ESP32Forth

Télécharger ESP32forth

La première étape consiste à récupérer le code source, en langage C, de ESP32forth.

Utilisez de préférence la version la plus récente :

<https://esp32forth.appspot.com/ESP32forth.html>

Contenu du fichier téléchargé :

```
ESP32forth-7.0.x.x
  ESP32forth
    readme.txt
    esp32forth.ino
    optional
      SPI-flash.h
      serial-blueooth.h
      ...etc...
```

Compilation et installation de ESP32forth

Décompressez le fichier **esp32forth.ino** dans un répertoire de travail. Le répertoire **optional** contient des fichiers permettant l'extension de ESP32forth. Pour notre première compilation et téléversement de ESP32forth, ces fichiers ne sont pas nécessaires.

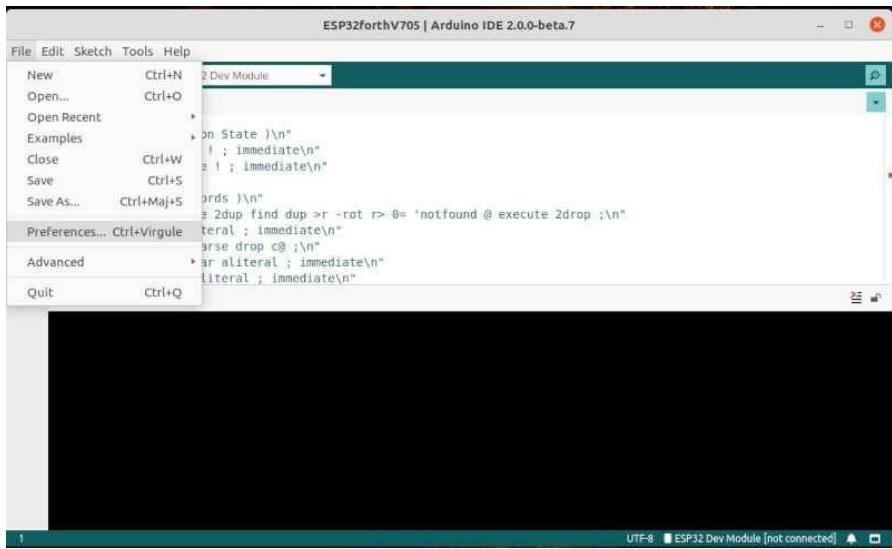
Pour compiler ESP32forth, vous devez disposer de ARDUINO IDE déjà installé sur votre ordinateur :

<https://docs.arduino.cc/software/ide-v2>

Une fois ARDUINO IDE installé, lancez-le. ARDUINO IDE est ouvert, ici la version 2.0¹.

Cliquez sur *file* (fichier) et sélectionnez *Preferences*:

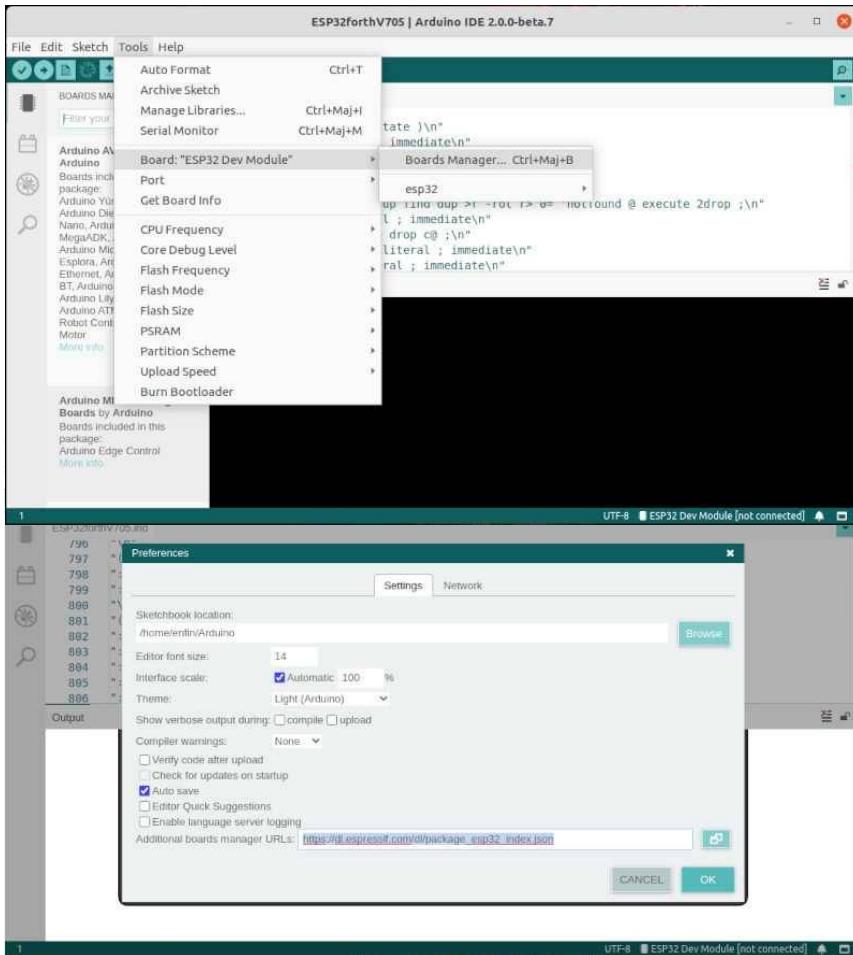
¹ Remarque concernant les versions ESP32forth – la version dite stable 7.0.6.19 nécessite pour une compilation correcte les bibliothèques de la carte Espressif 1.0.6, la version récente 7.0.7.15 nécessite les bibliothèques 2.0.x.



Dans la fenêtre qui s'affiche, allez dans la zone de saisie marquée *Additional boards manager URLs*: et entrez cette ligne :

https://dl.espressif.com/dl/package_esp32_index.json

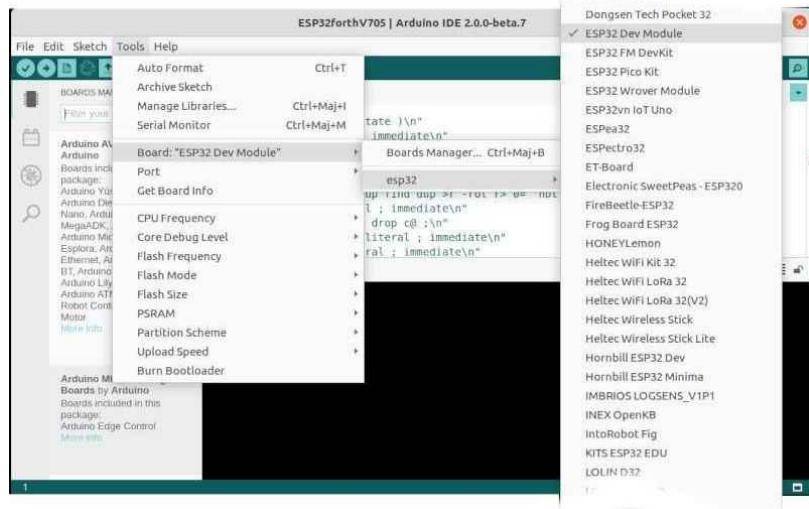
Ensuite, cliquez sur *Tools* et sélectionnez *Board*:



Cette sélection doit vous proposer l'installation des packages pour ESP32. Acceptez cette installation.

Vous deviez ensuite pouvoir accéder à la sélection des cartes ESP32:

Sélection de la carte **ESP32 Dev Module**:



Paramètres pour **ESP32 WROOM**

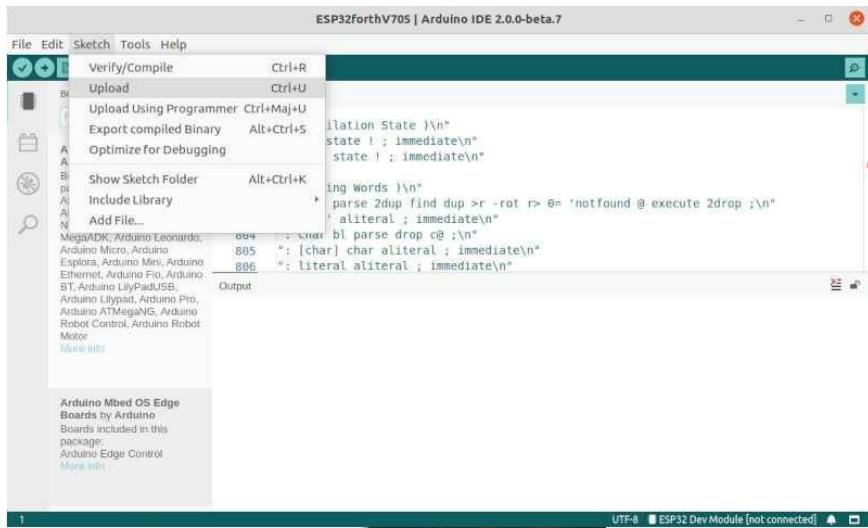
Voici les autres réglages nécessaires avant la compilation de ESP32forth. Accès des réglages en cliquant à nouveau sur *Tools*:

```
-- TOOLS----+-- BOARD      -----+-- ESP32 -----+-- ESP32 Dev Module
      +-- Port: -----+-- COMx
      |
      +-- CPU Frequency -----+-- 240 Mhz
      +-- Core Debug Level -----+-- None
      +-- Erase All Flash...-----+-- Disabled
      +-- Events Run On -----+-- Core 1
      +-- Flash Frequency -----+-- 80 Mhz
      +-- Flash Mode -----+-- QIO
      +-- Flash Size -----+-- 4MB
      +-- JTAG Adapter -----+-- FTDI Adapter
      +-- Arduino Runs on -----+-- Core 1
      +-- PSRAM -----+-- Disabled
      +-- Partition Scheme -----+-- Default 4MB with SPIFFS
      +-- Upload Speed -----+-- 921600
```

Lancer la compilation

Il ne reste plus qu'à compiler ESP32forth. Chargez le code source par *File* et *Open*.

On suppose que votre carte ESP32 est connectée à un port USB. Lancez la compilation en cliquant sur *Sketch* et en sélectionnant *Upload*:



Si tout se déroule correctement, vous devriez transférer le code binaire automatiquement dans la carte ESP32. Si la compilation se passe sans erreur, mais qu'il y a une erreur de transfert, refaites la compilation du fichier **esp32forth.ino**. Au moment du transfert, appuyez sur bouton marqué **BOOT** sur la carte ESP32. Ceci devrait rendre la carte disponible pour le transfert du code binaire de ESP32forth.

Installation et paramétrage de ARDUINO IDE en vidéo:

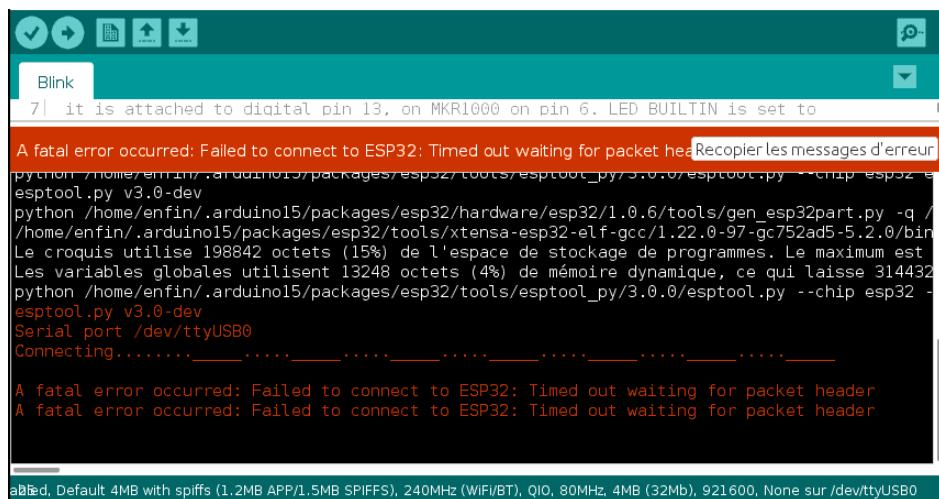
- Windows: <https://www.youtube.com/watch?v=2AZQfieHv9g>
- Linux : https://www.youtube.com/watch?v=JeD3nz0_nc

Résoudre l'erreur de connexion au téléversement

Apprenez à corriger l'erreur fatale qui s'est produite: "Failed to connect to ESP32: Timed out waiting for packet header" lors de la tentative de téléversement d'un nouveau code sur votre Carte ESP32 une fois pour toutes.

Certaines cartes de développement ESP32 (lisez Meilleures cartes ESP32) n'entrent pas en mode flashage/téléversement automatiquement lors du téléchargement d'un nouveau code.

Cela signifie que lorsque vous essayez de télécharger un nouveau croquis sur votre carte ESP32, ARDUINO IDE ne parvient pas à se connecter à votre carte et vous obtenez le message d'erreur suivant:



The screenshot shows the Arduino IDE's Serial Monitor window. The title bar says "Blink". The main area displays the following text:

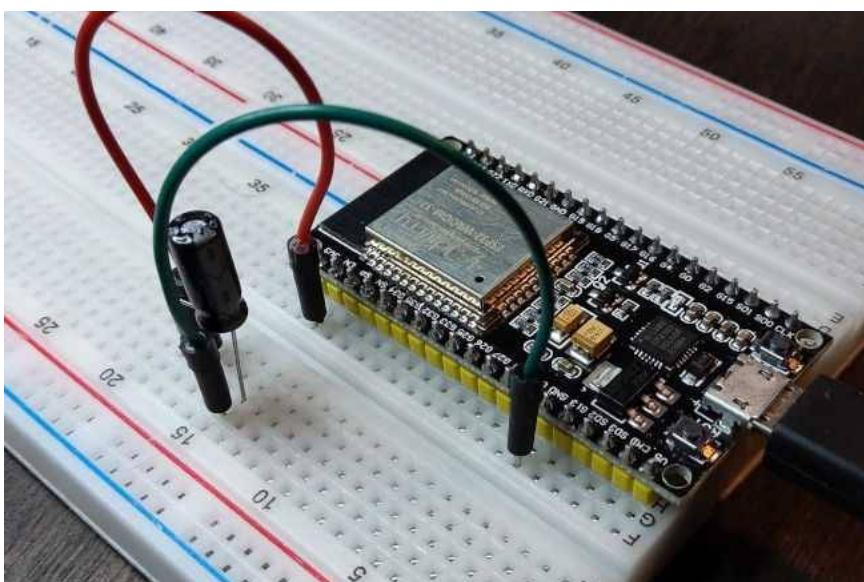
```
it is attached to digital pin 13, on MKR1000 on pin 6. LED BUILTIN is set to
A fatal error occurred: Failed to connect to ESP32: Timed out waiting for packet header
python /home/enfin/.arduino15/packages/esp32/tools/esptool_py/3.0.0/esptool.py --chip esp32 -e
esptool.py v3.0-dev
python /home/enfin/.arduino15/packages/esp32/hardware/esp32/1.0.6/tools/gen_esp32part.py -q /
/home/enfin/.arduino15/packages/esp32/tools/xtensa-esp32-elf-gcc/1.22.0-97-gc752ad5-5.2.0/bin
Le croquis utilise 198842 octets (15%) de l'espace de stockage de programmes. Le maximum est
Les variables globales utilisent 13248 octets (4%) de mémoire dynamique, ce qui laisse 314432
python /home/enfin/.arduino15/packages/esp32/tools/esptool_py/3.0.0/esptool.py --chip esp32 -
esptool.py v3.0-dev
Serial port /dev/ttyUSB0
Connecting.....
```

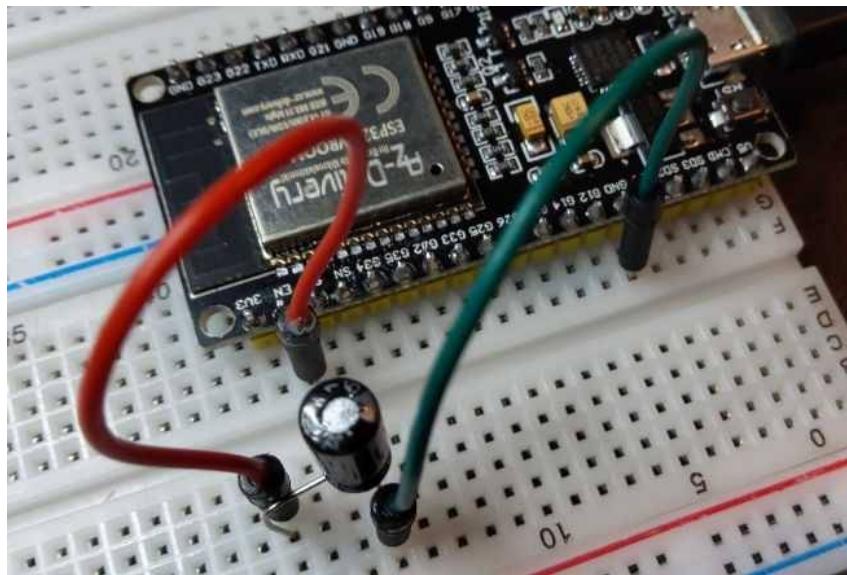
At the bottom of the window, there are two red error messages:

```
A fatal error occurred: Failed to connect to ESP32: Timed out waiting for packet header
A fatal error occurred: Failed to connect to ESP32: Timed out waiting for packet header
```

The status bar at the bottom of the IDE window shows: "at86d, Default 4MB with spiffs (1.2MB APP/1.5MB SPIFFS), 240MHz (WiFi/BT), QIO, 80MHz, 4MB (32Mb), 921600, None sur /dev/ttyUSB0"

Pour que la carte ESP32 passe automatiquement en mode flash/téléchargement, on peut connecter un condensateur électrolytique de 10uF entre la broche EN et GND:





Cette manipulation n'est nécessaire que si vous êtes dans la phase de téléversement de ESP32forth depuis ARDUINO IDE. Une fois ESP32forth installé sur la carte ESP32, l'utilisation de ce condensateur n'est plus nécessaire.

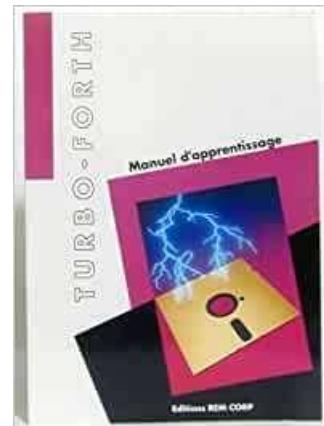
Pourquoi programmer en langage FORTH sur ESP32?

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:B0014IGOZO)
- Tours de FORTH (ed Eyrolles - 1985 - ISBN-13: 978-2212082258)
- FORTH pour CP/M et MSDOS (ed Loisitech - 1986)
- TURBO-Forth, manuel d'apprentissage (ed Rem CORP - 1990)
- TURBO-Forth, guide de référence (ed Rem CORP - 1991)



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

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

Celà faisait un mois que les développeurs en langage C tournaient le problème dans tous les sens, jusqu'à réaliser du reverse engineering avec SOURCER (un désassemblleur) 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 téléchargement. Exemple :

```
: >gray ( n -- n' )
    dup 2/ xor      \ n' = n xor ( 1 décalage logique à 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 reçus
    rc5_ok = 1;                    // Le processus de décodage est OK
    detachInterrupt(0); // Désactiver 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 prechargement 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 !       \ Definit valeur prechargement 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

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 librairies disponibles. Mais vous resterez enfermés dans les capacités de ces librairies. L'adaptation des codes en langage C requiert une réelle connaissance en programmation en langage C et maîtriser l'architecture des cartes ESP32. La mise au point de programmes complexes sera toujours un souci.
- tenter l'aventure FORTH et explorer un monde nouveau et passionnant. Certes, ce ne sera pas facile. Il faudra comprendre l'architecture 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  
;  
\ Desactivez une broche de port, ne changez pas les autres.  
: \__ ( pinmask portadr -- )  
    mclr  
;
```

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

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

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

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

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

Bonne programmation.

Utiliser les nombres avec ESP32Forth

Nous avons démarré ESP32Forth sans souci. Nous allons maintenant approfondir quelques manipulations sur les nombres pour comprendre comment maîtriser le micro-contrôleur en langage FORTH.

Comme beaucoup d'ouvrages, nous pourrions commencer par un exemple de programme trivial, clignotement de LED par exemple. Dans ce genre par exemple :

```
\ define LEDs GPIOs
25 constant ledRED
26 constant ledYELLOW
27 constant ledGREEN
\ define masks for red yellow and green LEDs
1 ledRED      defMASK: mLED_RED
1 ledYELLOW   defMASK: mLED_YELLOW
1 ledGREEN    defMASK: mLED_GREEN

\ initialisation GPIO G25 G26 and G27 in output mode
: GPIO.init ( -- )
  1 mLED_RED      GPIO_ENABLE_REG regSet
  1 mLED_YELLOW   GPIO_ENABLE_REG regSet
  1 mLED_GREEN    GPIO_ENABLE_REG regSet
;
\ define a ON and OFF sequence
: GPIO.on.off.sequence { position mask delay -- }
  1 position mask GPIO_OUT_W1TS_REG regSet
  delay ms
  1 position mask GPIO_OUT_W1TC_REG regSet ;
```

Ce code, simple en apparence, nécessite déjà une base de connaissances, comme la notion d'adresse mémoire, registre, masques binaires, nombres hexadécimaux.

Nous allons donc commencer par aborder ces notions élémentaires en vous invitant à effectuer des manipulations simples.

Les nombres avec l'interpréteur FORTH

Au démarrage de ESP32Forth, la fenêtre du terminal TERA TERM (ou tout autre programme de terminal de votre choix) doit indiquer la disponibilité de ESP32Forth. Appuyez une ou deux fois sur la touche *ENTER* du clavier. ESP32Forth répond avec la confirmation de bonne exécution **ok..**

On va tester l'entrée de deux nombres, ici **25** et **33**.

Tapez ces nombres, puis *ENTER* au clavier.

ESP32Forth répond toujours par **ok..** Vous venez d'empiler deux nombres sur la pile du langage

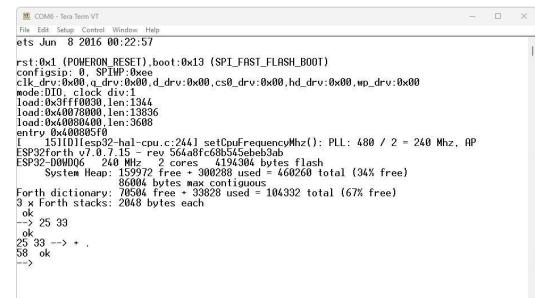


Figure 3: première opération avec
ESP32forth

ESP32Forth. Entrez maintenant **+ .** puis sur la touche *ENTER*. ESP32Forth affiche le résultat :

Cette opération a été traitée par l'interpréteur FORTH.

ESP32Forth, comme toutes les versions du langage FORTH a deux états :

- **interpréteur** : l'état que vous venez de tester en effectuant une simple somme de deux nombres ;
- **compilateur** : un état qui permet de définir de nouveaux mots. Cet aspect sera approfondi ultérieurement.

Saisie des nombres avec différentes base numérique

Afin de bien assimiler les explications, vous êtes invité à tester tous les exemples via la fenêtre du terminal TERA TERM.

Les nombres peuvent être saisis de manière naturelle. En décimal, ce sera TOUJOURS une séquence de chiffres, exemple :

```
-1234 5678 + .
```

Le résultat de cet exemple affichera **4444**. Les nombres et mots FORTH doivent être séparés par au moins un caractère *espace*. L'exemple fonctionne parfaitement si on tape un nombre ou mot par ligne :

```
-1234  
5678  
+  
. 
```

Les nombres peuvent être préfixés si on souhaite saisir des valeurs autrement que sous leur forme décimale :

- le signe **\$** pour indiquer que le nombre est une valeur hexadécimale ;

Exemple :

```
255 .      \ display 255  
$ff .      \ display 255
```

L'intérêt de ces préfixes est d'éviter toute erreur d'interprétation en cas de valeurs similaires :

```
$0305  
0305
```

ne sont **pas** des nombres **égaux** si la base numérique hexadécimale n'est pas explicitement définie!

Changement de base numérique

ESP32Forth dispose de mots permettant de changer de base numérique :

- **hex** pour sélectionner la base numérique hexadécimale ;
- **binary** pour sélectionner la base numérique binaire ;
- **decimal** pour sélectionner la base numérique décimale.

Tout nombre saisi dans une base numérique doit respecter la syntaxe des nombres dans cette base :

```
3E7F
```

provoquera une erreur si vous êtes en base décimale.

```
hex 3e7f
```

fonctionnera parfaitement en base hexadécimale. La nouvelle base numérique reste valable tant qu'on ne sélectionne pas une autre base numérique :

```
hex
$0305
0305
```

sont des nombres égaux!

Une fois un nombre déposé sur la pile de données dans une base numérique, sa valeur ne change plus. Par exemple, si vous déposez la valeur **\$ff** sur la pile de données, cette valeur qui est **255** en décimal, ou **11111111** en binaire, ne changera pas si on revient en décimal :

```
hex ff decimal . \ display: 255
```

Au risque d'insister, **255** en décimal est **la même valeur** que **\$ff** en hexadécimal !

Dans l'exemple donné en début de chapitre, on définit une constante en hexadécimal :

```
25 constant ledRED
```

Si on tape :

```
hex ledRED .
```

Ceci affichera le contenu de cette constante sous sa forme hexadécimale. Le changement de base n'a **aucune conséquence** sur le fonctionnement final du programme FORTH.

Binaire et hexadécimal

Le système de numération binaire moderne, base du code binaire, a été inventé par Gottfried Leibniz en 1689 et apparaît dans son article Explication de l'Arithmétique Binaire en 1703.

Dans son article, LEIBNITZ se sert des seuls caractères **0** et **1** pour décrire tous les nombres :

```
: bin0to15 ( -- )
    binary
```

```

$10 0 do
    cr i .
loop
    cr decimal ;
bin0to15 \ display:
0
1
10
11
100
101
110
111
1000
1001
1010
1011
1100
1101
1110
1111

```

Est-ce nécessaire de comprendre le codage binaire ? Je dirai oui et non. **Non** pour les usages de la vie courante. **Oui** pour comprendre la programmation des micro-contrôleurs et la maîtrise des opérateurs logiques.

C'est Georges Boole qui a décrit de manière formelle la logique. Ses travaux ont été oubliés jusqu'à l'apparition des premiers ordinateurs. C'est Claude Shannon qui se rend compte qu'on peut appliquer cet algèbre dans la conception et l'analyse de circuits électriques.

L'algèbre de Boole manipule exclusivement des **0** et des **1**.

Les composants fondamentaux de tous nos ordinateurs et mémoires numériques utilisent le codage binaire et l'algèbre de Boole.

La plus petite unité de stockage est l'octet. C'est un espace constitué de 8 bits. Un bit ne peut avoir que deux états : **0** ou **1**. La valeur la plus petite pouvant être stockée dans un octet est **00000000**, la plus grande étant **11111111**. Si on coupe en deux un octet, on aura :

- quatre bits de poids faible, pouvant prendre les valeurs **0000** à **1111** ;
- quatre bits de poids fort pouvant prendre une de ces mêmes valeurs.

Si on numérote toutes les combinaisons entre 0000 et 1111, en partant de 0, on arrive à 15 :

```

: bin0to15 ( -- )
    binary
    $10 0 do

```

```

cr i .
i hex . binary
loop
cr decimal ;
bin0to15 \ display:
0 0
1 1
10 2
11 3
100 4
101 5
110 6
111 7
1000 8
1001 9
1010 A
1011 B
1100 C
1101 D
1110 E
1111 F

```

Dans la partie droite de chaque ligne, on affiche la même valeur que dans la partie gauche, mais en hexadécimal : **1101** et **D** sont les mêmes valeurs !

La représentation hexadécimale a été choisie pour représenter des nombres en informatique pour des raisons pratiques. Pour la partie de poids fort ou faible d'un octet, sur 4 bits, les seuls combinaisons de représentation hexadécimale seront comprises entre **0** et **F**. Ici, les lettres A à F **sont des chiffres** hexadécimaux !

```
$3E \ is more readable as 00111110
```

La représentation hexadécimale offre donc l'avantage de représenter le contenu d'un octet dans un format fixe, de **00** à **FF**. En décimal, il aurait fallu utiliser 0 à 255.

Taille des nombres sur la pile de données FORTH

ESP32forth utilise une pile de données de 32 bits de taille mémoire, soit 4 octets (8 bits x 4 = 32 bits). La plus petite valeur hexadécimale pouvant être empilée sur la pile FORTH sera **00000000**, la plus grande sera **FFFFFFFF**. Toute tentative d'empiler une valeur de taille supérieure se solde par un écratage de cette valeur :

```

hex
abcdefabcdefabcdef . \ display: EFABCDE

```

Empilons la plus grande valeur possible au format hexadécimal sur 32 bits (4 octets) :

```

decimal
$ffffffff . \ display: -1

```

Je vous voit surpris, mais ce résultat est **normal** ! Le mot `.` Affiche la valeur qui est au sommet de la pile de données sous sa forme signée. Pour afficher la même valeur non signée, il faut utiliser le mot `u.` :

\$fffffff u. \ display: 4294967295

C'est parce que sur les 32 bits utilisés par FORTH pour représenter un nombre entier, le bit de poids fort est utilisé comme signe :

- si le bit de poids fort est à **0**, le nombre est positif ;
 - si le bit de poids fort est à **1**, le nombre est négatif.

Donc, si vous avez bien suivi, nos valeurs décimales 1 et -1 sont représentées sur la pile, au format binaire sous cette forme :

Et c'est là qu'on va faire appel à notre mathématicien, Mr LEIBNITZ, pour additionner en binaire ces deux nombres. Si on fait comme à l'école, en commençant par la droite, il faudra simplement respecter cette règle : $1 + 1 = 10$ en binaire. On met les résultats sur une troisième ligne :

Etape suivante :

Arrivé à la fin, on aura comme résultat :

Mais comme ce résultat a un 33ème bit de poids fort à 1, sachant que le format des entiers est strictement limité à 32 bits, le résultat final est **0**. C'est surprenant ? C'est pourtant ce que fait toute horloge digitale. Masquez les heures. Arrivé à 59, rajoutez 1, l'horloge affichera 0.

Les règles de l'arithmétique décimale, à savoir $-1 + 1 = 0$ ont été parfaitement respectées en logique binaire !

Accès mémoire et opérations logiques

La pile de données n'est en aucun cas un espace de stockage de données. Sa taille est d'ailleurs très limitée. Et la pile est partagée par beaucoup de mots. L'ordre des paramètres est fondamental. Une erreur peut générer des dysfonctionnements. Prenons le cas du mot **dump** qui affiche le contenu d'un espace mémoire :

```
hex
0 variable score
score 10 dump \ display:
1073670412          00 00 00 00
1073670416      55 51 54 55 48 51
```

En gras et en rouge on retrouve les quatre octets réservés au stockage d'une valeur dans notre variable **score**. Stockons une valeur quelconque dans **score** :

```
decimal
1900 score !
hex
score 10 dump \ display :
3FEE90C          6C 07 00 00
3FEE910      37 33 36 37 30 33 34 34 79 64 31 30
```

On retrouve les quatre octets contenant notre valeur décimale **1900, 0000076C** en hexadécimal. Encore surpris ? Alors c'est l'effet du codage binaire et ses subtilités qui en sont la cause. En mémoire, les octets sont stockés en commençant par ceux de poids faible. A la récupération, le mécanisme de transformation est transparent :

```
score @ . \ display 1900
```

Revenons au code qui fait clignoter une LED. Extrait :

```
1 mLED_RED     GPIO_ENABLE_REG regSet
```

Ce code active une sortie GPIO associée à une LED. Le mot **GPIO_ENABLE_REG** est une constante, dont le contenu est un masque pointant cette LED. On aurait aussi bien pu écrire ceci :

```
1 25 lshift GPIO_ENABLE_REG !
```

Ici, le mot **lshift** effectue un décalage logique de 25 bits vers la gauche :

```
\ before shift: %00000000000000000000000000000000
\ after shift: %00000010000000000000000000000000
```

Pour rappel, les GPIOs² sont numérotés de 0 à 31. Pour activer un autre GPIO, par exemple GPIO17, on aurait exécuté ceci :

```
1 17 lshift GPIO_ENABLE_REG !
```

Supposons que nous souhaitions activer en une seule commande les GPIOs 17 et 25. On exécutera ceci :

```
1 25 lshift
```

2 General Purpose Input/Output = Entrée-sortie à usage général

1 17 lshift or GPIO ENABLE REG !

Qu'est-ce que nous avons fait ? Voici le détail des opérations :

```
\ 1 25 lshift \ %000000100000000000000000000000000000000  
\ 1 17 lshift \ %000000100000000100000000000000000000000  
\ or \ %000000100000000100000000000000000000000
```

Le mot **or** a réalisé une opération qui combine les deux décalages en un seul masque binaire.

Revenons à notre variable `score`. On souhaite isoler l'octet de poids faible. Plusieurs solutions s'offrent à nous. Une solution exploite le masquage binaire avec l'opérateur logique `and` :

```
hex  
score @ . \ display: 76C  
score @  
$000000FF and . \ display: 6C
```

Pour isoler le second octet en partant de la droite :

```
score @  
$0000FF00 and .      \ display: 0700
```

Ici, nous nous sommes amusés avec le contenu d'une variable. Pour maîtriser un micro-contrôleur comme celui monté sur la carte ESP32, les mécanismes ne sont guère différents. Le plus difficile est de trouver les bons registres. Ce sera l'objet d'un autre chapitre.

Pour conclure ce chapitre, il y a encore beaucoup à apprendre sur la logique binaire et les différents codages numériques possibles. Si vous avez testé les quelques exemples donnés ici, vous comprenez certainement que FORTH est un langage intéressant :

- grâce à son interpréteur qui permet d'effectuer de nombreux tests, ce de manière interactive sans nécessiter de recompilation en téléversement de code ;
 - un dictionnaire dont la plupart des mots sont accessibles depuis l'interpréteur ;
 - un compilateur permettant de rajouter de nouveaux mots *à la volée*, puis les tester immédiatement.

Enfin, ce qui ne gâche rien, le code FORTH, une fois compilé, est certainement aussi performant que son équivalent en langage C.

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 obsolument 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....

Commentaires et mise au point

Il n'existe pas d'IDE³ pour gérer et présenter le code écrit en langage FORTH de manière structurée. Au pire, vous utilisez un éditeur de texte ASCII, au mieux un vrai IDE et des fichiers texte :

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

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

```
: cycle.stop -1 +to MAX_LIGHT_TIME MAX_LIGHT_TIME 0 = if  
LOW myLIGHTS pin else 0 rerun then ;
```

Ce code sera parfaitement compilé par ESP32forth. Mais restera-t-il compréhensible dans le futur s'il faut le modifier ou le réutiliser dans une autre application ?

Ecrire un code FORTH lisible

Commençons par le nomage du mot à définir, ici **cycle.stop**. ESP32forth permet d'écrire des noms de mots très longs. La taille des mots définis n'a aucune influence sur les performances de l'application finale. On dispose donc d'une certaine liberté pour écrire ces mots :

- à la manière de la programmation objet en JavaScript: **cycle.stop**
- à la manière CamelCoding **cycleStop**
- pour programmeur voulant un code très compréhensible **cycle-stop-lights**
- programmeur qui aime le code concis **csl**

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

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

³ Integrated Development Environment = Environnement de Développement Intégré

- Mot d'affichage de donnée **.date**

Et qu'en est-il du nommage des mots FORTH dans une langue autre qu'en anglais ? Là encore, une seule règle : **liberté totale** ! Attention cependant, ESP32forth n'accepte pas les noms écrits dans des alphabets différents de l'alphabet latin. Vous pouvez cependant utiliser ces alphabets pour les commentaires :

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

ou

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

Indentation du code source

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

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

Exemple :

```
60 constant MAX_LIGHT_TIME_NORMAL_CYCLE
: cycle.stop
  -1 +to MAX_LIGHT_TIME
  MAX_LIGHT_TIME 0 =
  if
    LOW myLIGHTS pin
  else
    0 rerun
  then
;
```

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

```
: cycle.stop
  -1 +to MAX_LIGHT_TIME
  MAX_LIGHT_TIME 0 =
  if           LOW myLIGHTS pin
  else         0 rerun
  then
;
```

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

```
: socketError ( -- )
  errno dup
```

```

case
  2 of      ." No such file "
  5 of      ." I/O error "
  9 of      ." Bad file number "
  22 of     ." Invalid argument "
endcase
. quit
;

```

Les commentaires

Comme tout langage de programmation, le langage FORTH permet le rajout de commentaires dans le code source. Le rajout de commentaires n'a aucune conséquence sur les performances de l'application après compilation du code source.

En langage FORTH, nous disposons de deux mots pour délimiter des commentaires :

- le mot **(** suivi impérativement d'au moins un caractère espace. Ce commentaire est achevé par le caractère **)** ;
- le mot **** suivi impérativement d'au moins un caractère espace. Ce mot est suivi d'un commentaire de taille quelconque entre ce mot et la fin de la ligne.

Le mot **(** est largement utilisé pour les commentaires de pile. Exemples :

```

dup  ( n - n n )
swap ( n1 n2 - n2 n1 )
drop  ( n -- )
emit  ( c -- )

```

Les commentaires de pile

Comme nous venons de le voir, ils sont marqués par **(** et **)**. Leur contenu n'a aucune action sur le code FORTH en compilation ou en exécution. On peut donc mettre n'importe quoi entre **(** et **)**. Pour ce qui concerne les commentaires de pile, on restera très concis. Le signe **--** symbolise l'action d'un mot FORTH. Les indications figurant avant **--** correspondent aux données déposées sur la pile de données avant l'exécution du mot. Les indications figurant après **--** correspondent aux données laissées sur la pile de données après exécution du mot. Exemples :

- **words** **(--)** signifie que ce mot ne traite aucune donnée sur la pile de données ;
- **emit** **(c --)** signifie que ce mot traite une donnée en entrée et ne laisse rien sur la pile de données ;
- **bl** **(-- 32)** signifie que ce mot ne traite pas de donnée en entrée et laisse la valeur décimale 32 sur la pile de données ;

Il n'y a aucune limitation sur le nombre de données traitées avant ou après exécution du mot. Pour rappel, les indications entre **(** et **)** sont seulement là pour information.

Signification des paramètres de pile en commentaires

Pour commencer, une petite mise au point très importante s'impose. Il s'agit de la taille des données en pile. Avec ESP32Forth, les données de pile occupent 4 octets. Ce sont donc des entiers au format 32 bits. Cependant, certains mots traitent des données au format 8 bits. Alors on met quoi sur la pile de données ? Avec ESP32Forth, ce seront **TOUJOURS DES DONNEES 32 BITS** ! Un exemple avec le mot **c!** :

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

Ici, le paramètre **c** indique qu'on empile une valeur entière au format 32 bits, mais dont la valeur sera toujours comprise dans l'intervale [0..255].

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

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

```
create myDelimiter
  0 c,
64 myDelimiter c!  ( n1 n2 -- )
```

Mais c'est nettement moins explicite que la version précédente. Voici quelques symboles que vous serez amené à voir au fil des codes sources :

- **addr** indique une adresse mémoire littérale ou délivrée par une variable ;
- **c** indique une valeur 8 bits dans l'intervale [0..255]
- **d** indique une valeur double précision.
Non utilisé avec ESP32Forth qui est déjà au format 32 bits ;
- **f1** indique une valeur booléenne, 0 ou non zéro ;
- **n** indique un entier. Entier signé 32 bits pour ESP32Forth ;
- **str** indique une chaîne de caractère. Equivaut à **addr len --**
- **u** indique un entier non signé

Rien n'interdit d'être un peu plus explicite :

```
: SQUARE ( n -- n-exp2 )
  dup *
;
```

Commentaires des mots de définition de mots

Les mots de définition utilisent **create** et **does>**. Pour ces mots, il est conseillé d'écrire les commentaires de pile de cette manière :

```
\ define a command or data stream for SSD1306
```

```

: streamCreate: ( comp: <name> | exec: -- addr len )
  create
    here      \ leave current dictionary pointer on stack
    0 c,      \ initial lenght data is 0
  does>
    dup 1+ swap c@
    \ send a data array to SSD1306 connected via I2C bus
    sendDatasToSSD1306
;

```

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

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

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

Les commentaires textuels

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

```

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

```

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

```

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

```

Commentaire en début de code source

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

```
\ ****
```

```

\ Manage commands for OLED SSD1306 128x32 display
\   Filename:      SSD10306commands.fs
\   Date:         21 may 2023
\   Updated:       21 may 2023
\   File Version: 1.0
\   MCU:          ESP32-WROOM-32
\   Forth:         ESP32forth all versions 7.x++
\   Copyright:     Marc PETREMAN
\   Author:        Marc PETREMAN
\   GNU General Public License
\ ****

```

Toutes ces informations sont à votre libre choix. Elles peuvent devenir très utiles quand on revient des mois ou des années plus tard sur le contenu d'un fichier.

Pour conclure, n'hésitez pas à commenter et indenter vos fichiers sources en langage FORTH.

Outils de diagnostic et mise au point

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

```

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

```

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

Le décompilateur

Dans un compilateur conventionnel, le code source est transformé en code exécutable contenant les adresses de référence à une bibliothèque équipant le compilateur. Pour disposer d'un code exécutable , il faut linker le code objet. A aucun moment le programmeur ne peut avoir accès au code exécutable contenu dans ses bibliothèques avec les seules ressources du compilateur.

Avec ESP32Forth, le développeur peut décompiler ses définitions. Pour décompiler un mot, il suffit de taper **see** suivi du mot à décompiler :

```

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

```

Beaucoup de mots du dictionnaire FORTH de ESP32Forth peuvent être décompilés.

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

Dump mémoire

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

```
create myDATAS 01 c, 02 c, 03 c, 04 c,  
hex  
myDATAS 4 dump      \ displays :  
3FEE4EC                                01 02 03 04
```

Moniteur de pile

Le contenu de la pile de données peut être affiché à tout moment grâce au mot **.s**. Voici la définition du mot **.DEBUG** qui exploite **.s** :

```
variable debugStack  
  
: debugOn ( -- )  
    -1 debugStack !  
;  
  
: debugOff ( -- )  
    0 debugStack !  
;  
  
: .DEBUG  
    debugStack @  
    if  
        cr ." STACK: " .s  
        key drop  
    then  
;
```

Pour exploiter **.DEBUG**, il suffit de l'insérer dans un endroit stratégique du mot à mettre au point :

```
\ example of use:  
: myTEST  
    128 32 do  
        i .DEBUG  
        emit  
    loop  
;
```

Ici, on va afficher le contenu de la pile de données après exécution du mot **i** dans notre boucle **do loop**. On active la mise au point et on exécute **myTEST** :

```
debugOn  
myTest  
\ displays:  
\ STACK: <1> 32  
\ 2
```

```
\ STACK: <1> 33
\ 3
\ STACK: <1> 34
\ 4
\ STACK: <1> 35
\ 5
\ STACK: <1> 36
\ 6
\ STACK: <1> 37
\ 7
\ STACK: <1> 38
```

Quand la mise au point est activée par **debugOn**, chaque affichage du contenu de la pile de données met en pause notre boucle **do loop**. Exécuter **debugOff** pour que le mot **myTEST** s'exécute normalement.

Dictionnaire / Pile / Variables / Constantes

Étendre le dictionnaire

Forth appartient à la classe des langages d'interprétation tissés. Cela signifie qu'il peut interpréter les commandes tapées sur la console, ainsi que compiler de nouveaux sous-programmes et programmes.

Le compilateur Forth fait partie du langage et des mots spéciaux sont utilisés pour créer de nouvelles entrées de dictionnaire (c'est-à-dire des mots). Les plus importants sont : (commencer une nouvelle définition) et ; (termine la définition). Essayons ceci en tapant:

```
: *+ * + ;
```

Ce qui s'est passé? L'action de : est de créer une nouvelle entrée de dictionnaire nommée *+ et passer du mode interprétation au mode compilation. En mode compilation, l'interpréteur recherche les mots et, plutôt que de les exécuter, installe des pointeurs vers leur code. Si le texte est un nombre, au lieu de le pousser sur la pile, ESP32forth construit le nombre dans le dictionnaire l'espace alloué pour le nouveau mot, suivant le code spécial qui met le numéro stocké sur la pile chaque fois que le mot est exécuté. L'action d'exécution de *+ est donc d'exécuter séquentiellement les mots définis précédemment * et +.

Le mot ; est spécial. C'est un mot immédiat et il est toujours exécuté, même si le système est en mode compilation. Ce que fait ; est double. Tout d'abord, il installe le code qui renvoie le contrôle au niveau externe suivant de l'interpréteur et, deuxièmement, il revient du mode compilation au mode interprétation.

Maintenant, essayez votre nouveau mot :

```
decimal 5 6 7 *+ . \ affiche 47 ok<#,ram>
```

Cet exemple illustre deux activités principales de travail dans Forth: ajouter un nouveau mot au dictionnaire, et l'essayer dès qu'il a été défini.

Gestion du dictionnaire

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

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

Piles et notation polonaise inversée

Forth a une pile explicitement visible qui est utilisée pour passer des nombres entre les mots (commandes). Utiliser Forth efficacement vous oblige à penser en termes de pile. Cela peut être difficile au début, mais comme pour tout, cela devient beaucoup plus facile avec la pratique.

En FORTH, La pile est analogue à une pile de cartes avec des nombres écrits dessus. Les nombres sont toujours ajoutés au sommet de la pile et retirés du sommet de la pile. 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
```

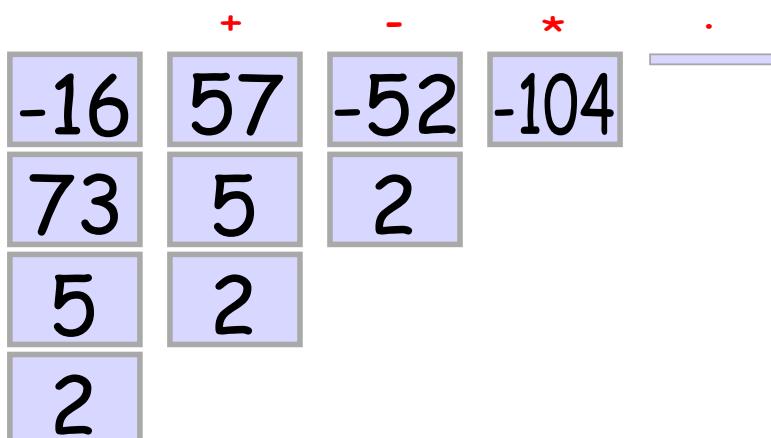
laisse la pile de paramètres dans l'état

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 placé sur la pile de retour de sorte que, lorsque le mot courant est terminé en cours d'exécution, le système sait où passer au mot suivant. Comme les mots peuvent être imbriqués, il doit y avoir une pile de ces adresses de retour.

En plus de servir de réservoir d'adresses de retour, l'utilisateur peut également stocker et récupérer à partir de la pile de retour, mais cela doit être fait avec soin car la pile de retour est essentielle à l'exécution du programme. Si vous utilisez la pile de retour pour le stockage temporaire, vous devez la remettre dans son état d'origine, sinon vous ferez probablement planter le système 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
```

Variabes

Une variable est un emplacement nommé en mémoire qui peut stocker un nombre, tel que le résultat intermédiaire d'un calcul, hors de la pile. Par exemple:

```
variable x
```

crée un emplacement de stockage nommé, **x**, qui s'exécute en laissant l'adresse de son emplacement de stockage au sommet de la pile:

```
x .      \ affiche l'adresse
```

Nous pouvons alors aller chercher ou stocker à cette adresse :

```
variable x
3 x !
x @ .      \ affiche: 3
```

Constantes

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

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

\ définit la fréquence du port SPI
4000000 constant SPI_FREQ

\ sélectionne le vocabulaire SPI
only FORTH SPI also

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

Valeurs pseudo-constantes

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

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

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

Outils de base pour l'allocation de mémoire

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

```
create graphic-array ( --- addr )
    %00000000 c,
    %00000010 c,
    %00000100 c,
```

```
%00001000 c,  
%00010000 c,  
%00100000 c,  
%01000000 c,  
%10000000 c,
```

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

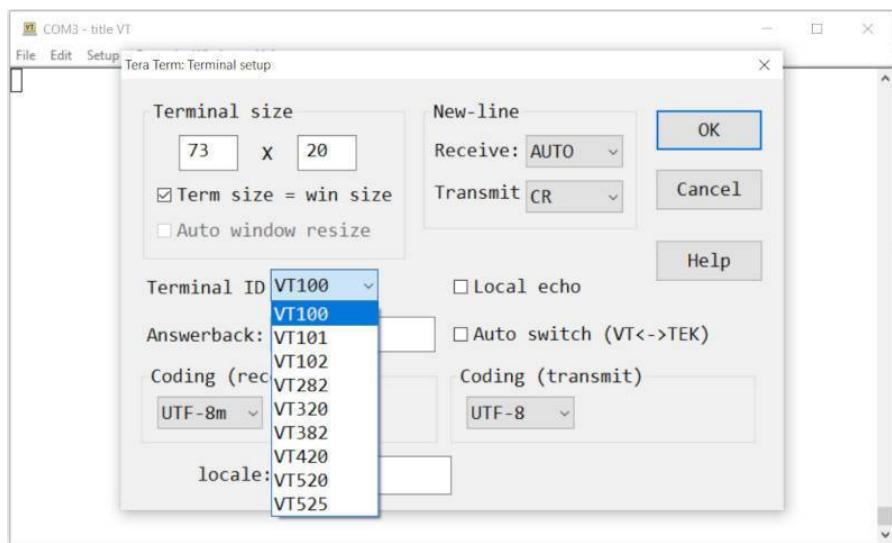
Nous pouvons maintenant accéder à la mémoire allouée à **graphic-array** en utilisant les mots de récupération et de stockage expliqués plus tôt. Pour calculer l'adresse du troisième octet attribué à **graphic-array** on peut écrire **graphic-array 2 +**, en se rappelant que les indices commencent à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

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

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**.

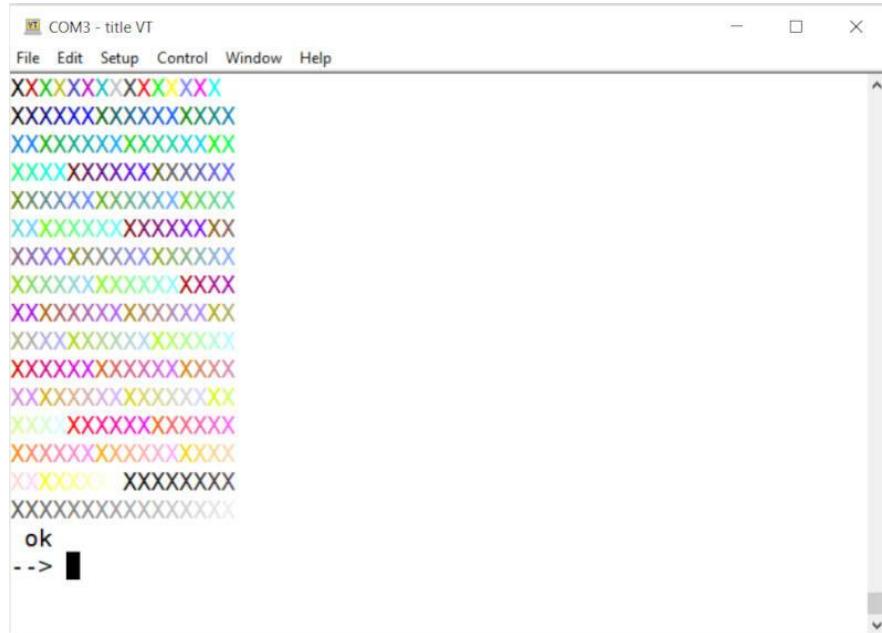
Le mot **page** vide l'écran du terminal et positionne le curseur au coin supérieur gauche de l'écran.

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:



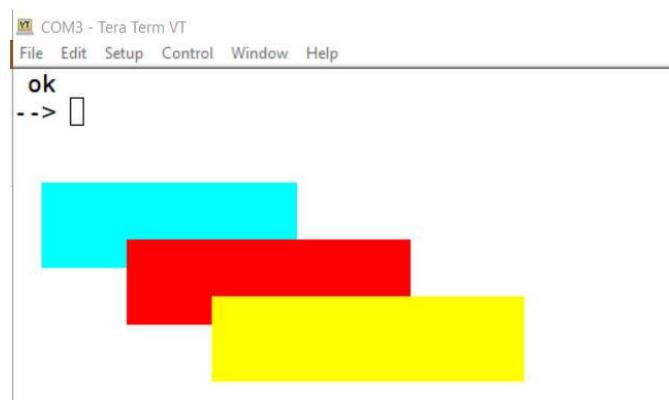
Position de l'affichage

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 varC varD varA varB }
....
```

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

Action sur les variables locales

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

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

A le même effet que ceci:

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

Dans cet exemple, var est défini explicitement par value.

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

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

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

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

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

```
\ addition deux entiers non signés, laisse sum et carry sur la pile
: um+ { u1 u2 -- sum carry }
  0 { sum }
```

```

cell for
    aft
        u1 $100 /mod to u1
        u2 $100 /mod to u2
        +
        cell 1- i - 8 * lshift +to sum
    then
next
sum
u1 u2 + abs
;

```

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

```

\ variables locales dans DUMP:
\ START_ADDR      \ première adresse pour dump
\ END_ADDR        \ dernière adresse pour dump
\ OSTART_ADDR     \ première adresse pour la boucle dans dump
\ LINES            \ nombre de lignes pour la boucle dump
\ myBASE          \ base numérique courante
internals
: dump ( start len -- )
    cr cr ." --addr--- "
    ." 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  -----chars-----"
    2dup + { END_ADDR }           \ store latest address to dump
    swap { START_ADDR }          \ store START address to dump
    START_ADDR 16 / 16 * { OSTART_ADDR } \ calc. addr for loop start
    16 / 1+ { LINES }
    base @ { myBASE }           \ save current base
hex
\ outer loop
LINES 0 do
    OSTART_ADDR i 16 * +          \ calc start address for current line
    cr <# # # # [char] - hold # # # #> type
    space space      \ and display address
    \ first inner loop, display bytes
    16 0 do
        \ calculate real address
        OSTART_ADDR j 16 * i +
        ca@ <# # # #> type space \ display byte in format: NN
loop
space
\ second inner loop, display chars
16 0 do
    \ calculate real address
    OSTART_ADDR j 16 * i +
    \ display char if code in interval 32-127
    ca@      dup 32 < over 127 > or
    if       drop [char] . emit
    else     emit
    then

```

```

    loop
loop
myBASE base !           \ restore current base
cr cr
;
forth

```

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

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

```

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

```

Structures de données pour 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 4×8 bits = 32 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 aisément 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:

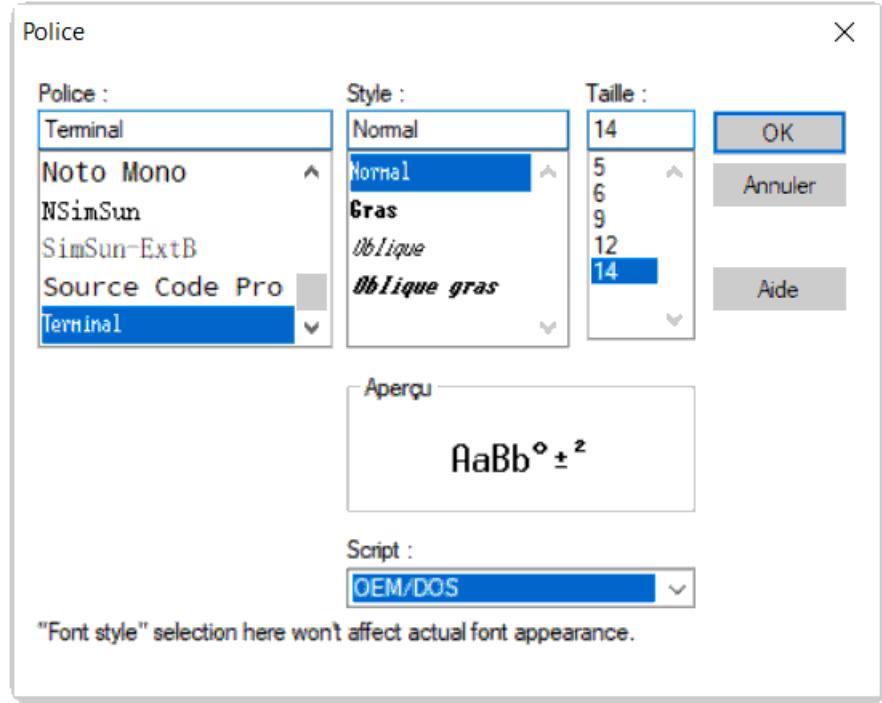
```
: xySCRaddr { x y -- addr }
    SCR_WIDTH y *
    x + mySCREEN +
;
: SCR@ ( x y -- c )
    xySCRaddr c@
;
: SCR! ( c x y -- )
    xySCRaddr c!
;
char X 15 5 SCR!      \ stocke char X à col 15 ligne 5
15 5 SCR@ emit        \ affiche X
```

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:

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

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

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

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

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

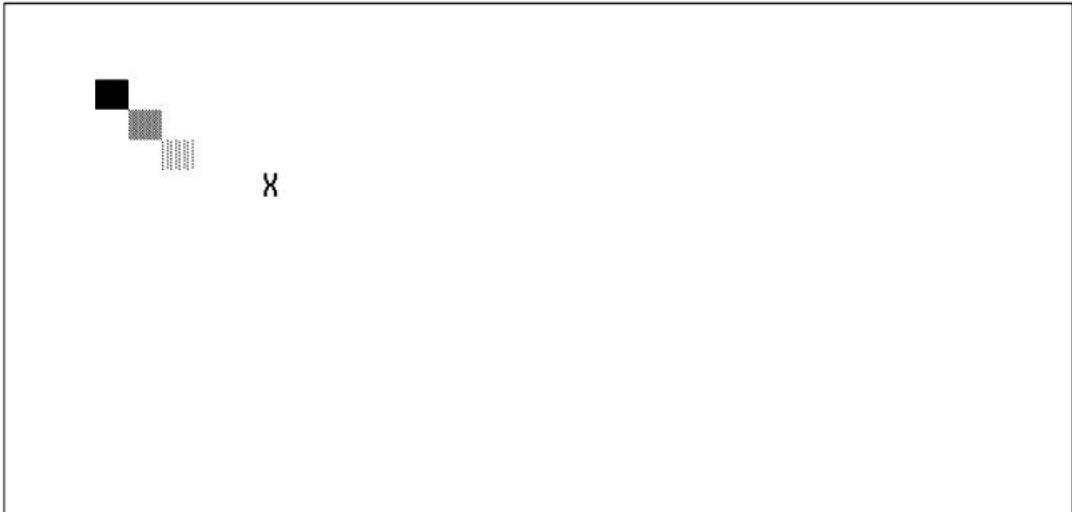
Au passage, on va créer une définition permettant d'afficher n fois un même caractère:

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

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

```
: dispScreen
    0 0 at-xy
    \ affiche bord superieur
    $da emit    $c4 SCR_WIDTH nEmit    $bf emit    cr
    \ affiche contenu ecran virtuel
    SCR_HEIGHT 0 do
        $b3 emit    i dispLine          $b3 emit    cr
    loop
    \ affiche bord inferieur
    $c0 emit    $c4 SCR_WIDTH nEmit    $d9 emit    cr
;
```

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,

```

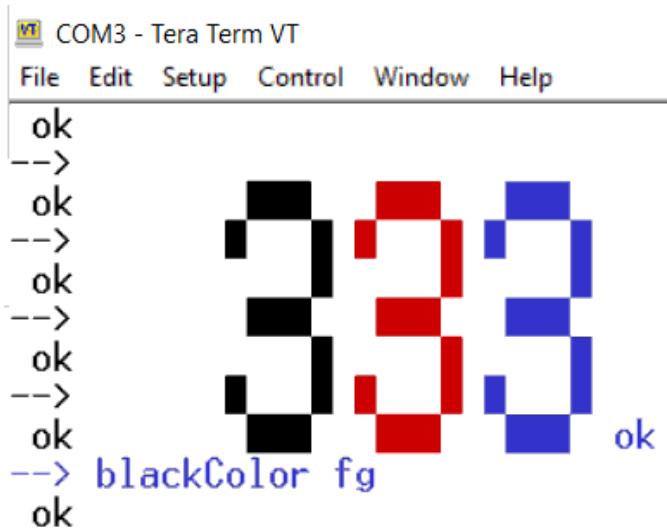
Pour l'affichage du sprite, à partir d'une position x y dans la fenêtre du terminal, une simple boucle suffit:

```

: .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:

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

Les nombres réels avec ESP32forth

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

Ce n'est pas surprenant. De base, ESP32forth n'utilise que des nombres entiers 32 bits via la pile de données. Les nombres entiers offrent certains avantages :

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

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

Mais les nombres entiers ont aussi des limites :

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

Depuis la version 7.0.6.5, ESP32forth intègre des opérateurs traitant des nombres réels.

Les nombres réels sont aussi dénommés nombres à virgule flottante.

Les réels avec ESP32forth

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

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

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

Precision des nombres réels avec ESP32forth

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

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

La précision limite de traitement des nombres réels avec ESP32forth est de six décimales :

```
12 set-precision
1.987654321e f.      \ affiche 1.987654668777
```

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

Constantes et variables réelles

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

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

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

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

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

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

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

Opérateurs arithmétiques sur les réels

ESP32Forth dispose de quatre opérateurs arithmétiques **F+ F- F* F/**:

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

ESP32forth dispose aussi de ces mots :

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

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

Opérateurs mathématiques sur les réels

ESP32forth dispose de plusieurs opérateurs mathématiques :

- **F**** élève un réel r_val à la puissance r_exp
- **FATAN2** calcule l'angle en radian à partir de la tangente.
- **FCOS** (r1 -- r2) Calcule le cosinus d'un angle exprimé en radians.

- **FEXP** ($\ln r$ -- r) calcule le réel correspondant à $e^{\ln r}$
- **FLN** (r -- $\ln r$) calcule le logarithme naturel d'un nombre réel.
- **FSIN** (r_1 -- r_2) calcule le sinus d'un angle exprimé en radians.
- **FSINCOS** (r_1 -- $r_{\cos} r_{\sin}$) calcule le cosinus et le sinus d'un angle exprimé en radians.

Quelques exemples :

```

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

4.605170e FEXP F.    \ affiche 100.000018

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

```

Opérateurs logiques sur les réels

ESP32forth permet aussi d'effectuer des tests logiques sur les réels :

- **F0<** (r -- f_l) teste si un nombre réel est inférieur à zéro.
- **F0=** (r -- f_l) indique vrai si le réel est nul.
- **f<** ($r_1 r_2$ -- f_l) f_l est vrai si $r_1 < r_2$.
- **f<=** ($r_1 r_2$ -- f_l) f_l est vrai si $r_1 \leq r_2$.
- **f<>** ($r_1 r_2$ -- f_l) f_l est vrai si $r_1 \neq r_2$.
- **f=** ($r_1 r_2$ -- f_l) f_l est vrai si $r_1 = r_2$.
- **f>** ($r_1 r_2$ -- f_l) f_l est vrai si $r_1 > r_2$.
- **f>=** ($r_1 r_2$ -- f_l) f_l est vrai si $r_1 \geq r_2$.

Transformations entiers ↔ réels

ESP32forth dispose de deux mots pour transformer des entiers en réels et inversement :

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

Exemple :

```
35 S>F  
F.    \ affiche 35.000000  
  
3.5e F>S .    \ affiche 3
```

Affichage des nombres et chaînes de caractères

Changement de base numérique

FORTH ne traite pas n'importe quels nombres. Ceux que vous avez utilisés en essayant les précédents exemples sont des entiers signés simple précision. 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+          \ incrémenté 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. Avec ESP32forth, ces primitives traitent les nombres entiers :

- **<#** débute une séquence de définition de format ;
- **#** insère un digit dans une séquence de définition de format ;
- **#\$** 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 #$ #>
type space ." EUR" ;
1245 .euros

```

Exemples d'exécution :

35 .EUROS	\ affiche	0,35 EUR
3575 .EUROS	\ affiche	35,75 EUR
1015 3575 + .EUROS	\ affiche	45,90 EUR

Dans la définition de **.EUROS**, le mot **<#** débute la séquence de définition de format d'affichage. Les deux mots **#** placent les chiffres des unités et des dizaines dans la chaîne

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

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

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

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

```
: :00 ( ---)
    DECIMAL #
    6 BASE !
    #
    [char] : HOLD
    DECIMAL ;
: 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
```

Ce répertoire téléphonique, 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 1 )
  DUP ABS <# #S ROT SIGN #>
;
: .R ( n 1 -- )
  >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

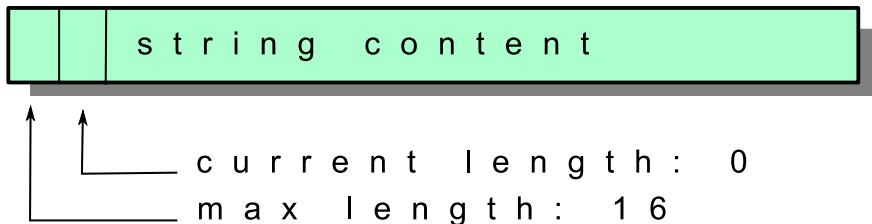
Les variables alpha-numérique texte n'existent pas nativement dans ESP32forth. Voici le premier essai de définition du mot **string** :

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

Une variable chaîne de caractères se définit comme ceci :

```
16 string strState
```

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



Code des mots de gestion de variables texte

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

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

\ compare two strings
: $= ( addr1 len1 addr2 len2 --- f1)
    str=
    ;

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

```

;

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

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

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

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

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

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

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

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

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

```

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

```
64 string myNewString
```

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

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

Exemple :

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

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

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

Ajout de caractère à une variable alphanumérique

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

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

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

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

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

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

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

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 communs.

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 ;

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;

Les mots à action différée

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

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

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

Exemple:

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

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

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

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

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

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

```
vector @ EXECUTE
```

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

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

Définition et utilisation de mots avec defer

L'initialisation d'un mot défini par **defer** est réalisée par **is** :

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

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

```
vector      \ execute  words
```

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

```
' page is vector
```

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

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

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

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

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

Définition d'une référence avant

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

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

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

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

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

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

Dépendance envers le contexte d'exploitation

ESP32forth utilise nativement comme flux d'entrée et sortie une liaison via le port série 1.

Dans le code source de ESP32forth, nous retrouvons ces lignes :

```
defer type
defer key
defer key?
```

Pour passer par le port série, ESP32forth initialise le mot `type` comme ceci :

```
' default-type is type
```

Si on active ESP32forth en mode serveur, le flux de `type` sera redirigé ainsi :

```
: server ( port -- )
  server
  ['] serve-key is key
  ['] serve-type is type
  webserver-task start-task
;
```

Voici la redirection de `type` si on utilise un flux TELNET :

```
: connection ( n -- )
  dup 0< if drop exit then to clientfd
  0 echo !
  ['] telnet-key is key
  ['] telnet-type is type quit ;
```

Et si on voulait rediriger l'affichage de texte vers un afficheur oled, il suffirait d'agir sur `type` de la même manière. Dans le chapitre *Paramétrage du transmetteur LoRa REYAX RYLR890*, on exploite cette propriété de `type` comme suit :

```
serial \ Select Serial vocabulary
: serial2-type ( a n -- )
  Serial2.write drop ;
: typeToLoRa ( -- )
  0 echo ! \ disable display echo from terminal
  ['] serial2-type is type
;
: typeToTerm ( -- )
  ['] default-type is type
  -1 echo ! \ enable display echo from terminal
;
```

En procédant ainsi, il devient très facile de transmettre un flux de texte vers le port série 2 :

```
: optionChoice
  ." choice option:" ;
```

```

optionChoice      \ display      choice options: on terminal
typeToLoRa
optionChoice      \ display      choice options: thru serial2
typeToTerm        \ restaure normal display

```

Dans ce cas précis, on définit plein de mots permettant de commander un transmetteur LoRa en utilisant des mots ordinaires comme **emit**, **type**, **.** Etc. Si nous n'activons pas la transmission vers le port série 2, donc vers le transmetteur LoRa, les mots qui communiquent avec ce transmetteur pourront être mis au point facilement :

```

\ Set the ADDRESS of LoRa transmitter:
\ s" <address>" value in interval [0..65535][?] (default 0)
: ATaddress ( addr len -- )
    ." AT+ADDRESS="
    type crlf
;

```

Si on exécute **ATaddress**, le flux texte sera affiché sur le terminal. Si vous avez bien suivi, vous savez quel mot exécuter pour rediriger le flux de **ATaddress** vers le port série 2.

En résumé, grâce aux mots à exécution différée, on peut agir sur l'action des mots FORTH déjà définis.

Un cas pratique

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

```

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

create ENdayNames ( --- addr)
    / / / / /

```

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

```

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

create FRdayNames ( --- addr)
    / / / / /

```

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

```
defer dayNames

: in-ENGLISH
  ['] ENdayNames is dayNames  ;

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

Voici maintenant les mots permettant de gérer ces deux tableaux :

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

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

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

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

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

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

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

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

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

Les mots de création de mots

FORTH est plus qu'un langage de programmation. C'est un méta-langage. Un méta-langage est un langage utilisé pour décrire, spécifier ou manipuler d'autres langages.

Avec ESP32forth, on peut définir la syntaxe et la sémantique de mots de programmation au-delà du cadre formel des définitions de base.

On a déjà vu les mots définis par **constant**, **variable**, **value**. Ces mots servent à gérer des données numériques.

Dans le chapitre Structures de données pour ESP32forth, on a également utilisé le mot **create**. Ce mot crée un en-tête permettant d'accéder à une zone de données mis en mémoire. Exemple :

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

Ici, chaque valeur est stockée dans la zone des paramètres du mot **temperatures** avec le mot **,**.

Avec ESP32forth, on va voir comment personnaliser l'exécution des mots définis par **create**.

Utilisation de does>

Cependant, il y a une combinaison de mots-clés "**CREATE**" et "**DOES>**", qui est souvent utilisée ensemble pour créer des mots (mots de vocabulaire) personnalisés avec des comportements spécifiques.

Voici comment cela fonctionne généralement en Forth :

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

Ensemble, cela ressemble à quelque chose comme ceci :

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

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

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

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

```
$3FF44004 constant GPIO_OUT_REG
```

OU

```
$3FF44004 defREG: GPIO_OUT_REG
```

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

- un code ESP32forth source plus lisible. On détecte facilement toutes les constantes nommant un registre ESP32 ;
- on se laisse la possibilité de modifier la partie **does>** de **defREG:** sans avoir ensuite à réécrire les lignes de code qui n'utiliseraient pas **defREG:**

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

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

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

Exemple de gestion de couleur

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

```
0 value currentCOLOR

\ define word as COLOR constant
: color: ( n -- <name> )
```

```

create
,
does>
@ to currentCOLOR
;

$00 color: setBLACK
$ff color: setWHITE

```

L'exécution du mot **setBLACK** ou **setWHITE** simplifie considérablement le code ESP32forth. Sans ce mécanisme, il aurait fallu répéter régulièrement une de ces lignes :

```
$00 currentCOLOR !
```

Ou

```
$00 constant BLACK
BLACK currentCOLOR !
```

Exemple, écrire en pinyin

Le pinyin est couramment utilisé dans le monde entier pour enseigner la prononciation du chinois mandarin, et il est également utilisé dans divers contextes officiels en Chine, comme les panneaux de signalisation, les dictionnaires et les manuels d'apprentissage. Il facilite l'apprentissage du chinois pour les personnes dont la langue maternelle utilise l'alphabet latin.

Pour écrire en chinois sur un clavier QWERTY, les Chinois utilisent généralement un système appelé "pinyin input" ou "saisie pinyin". Pinyin est un système de romanisation du chinois mandarin, qui utilise l'alphabet latin pour représenter les sons du mandarin.

Sur un clavier QWERTY, les utilisateurs tapent les sons du mandarin en utilisant la romanisation pinyin. Par exemple, si quelqu'un veut écrire le caractère "你" ("ní" signifiant "tu" ou "toi" en français), il peut taper "ni".

Dans ce code très simplifié, on peut programmer des mots pinyin pour écrire en mandarin. Le code ci-après ne fonctionne qu'avec le terminal PuTTY :

```
\ Work only with PuTTY terminal
internals
: chinese:
    create ( c1 c2 c3 -- )
        c, c, c,
    does>
        3 serial-type
    ;
forth
```

Pour trouver le code UTF8 d'un caractère chinois, copiez le caractère chinois, depuis Google Translate par exemple. Exemple :

```
Good Morning --> 早安 (Zao an)
```

Copiez 早 et allez dans le terminal PuTTy et tapez :

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

collez le caractère 早. ESP32forth doit afficher les codes suivants :

```
230 151 169
```

Pour chaque caractère chinois, on va exploiter ces trois codes ainsi :

```
169 151 230 chinese: Zao  
137 174 229 chinese: An
```

Utilisation :

```
Zao An \ display 早安
```

Avouez quand même que programmer ainsi c'est autre chose que ce qu'on peut faire en langage C. Non ?

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

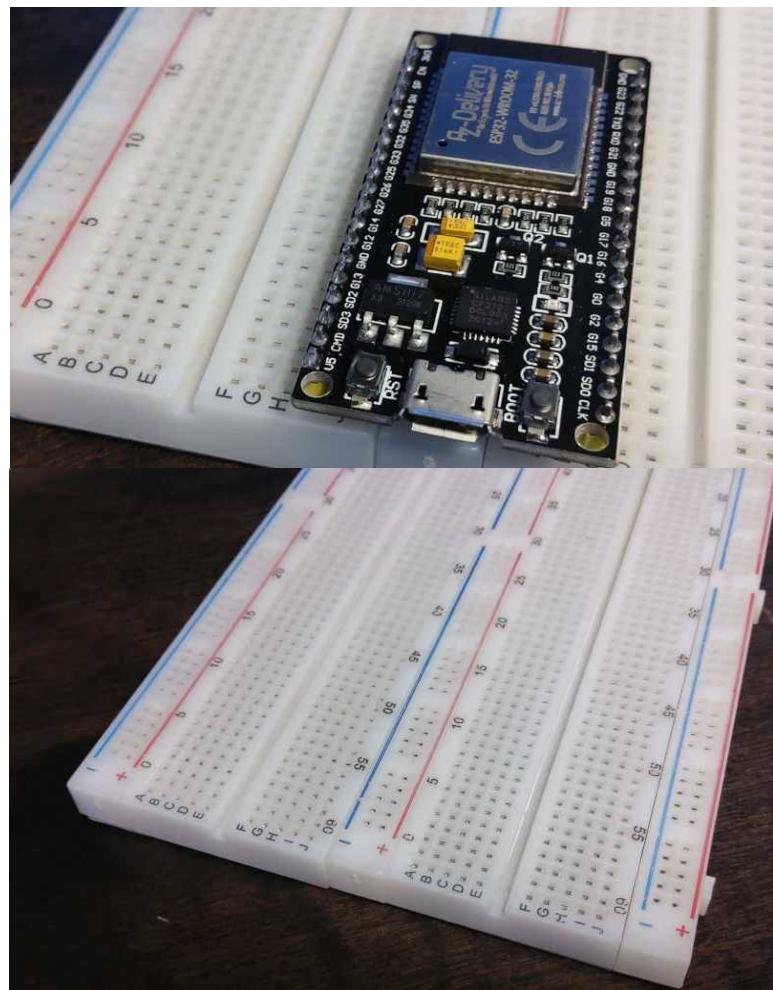
On va construire notre propre plaque d'essai. Pour cela, il faut disposer de deux plaques d'essai identiques.

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

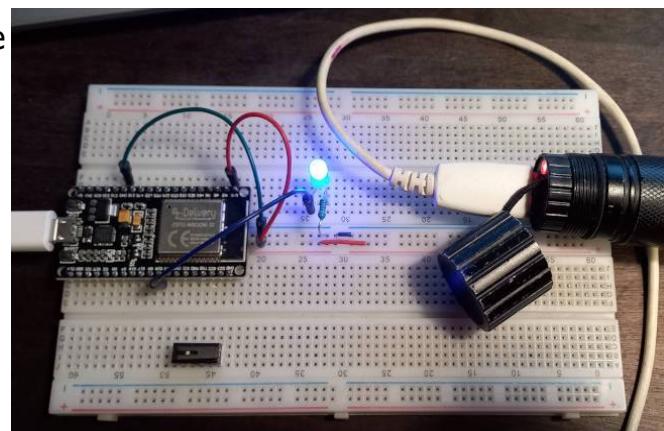
Nous allons voir ici comment alimenter une carte ESP32. Le but est de donner des solutions pour exécuter les programmes FORTH compilés par ESP32forth.

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.

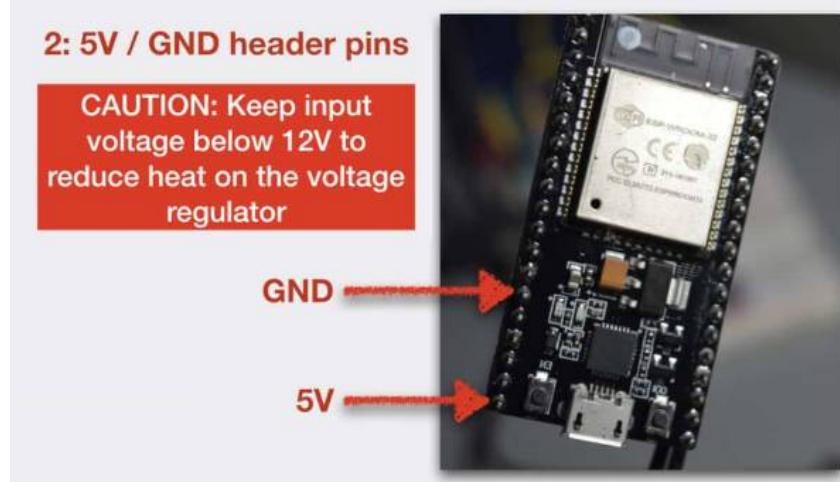


Alimentation par le pin 5V

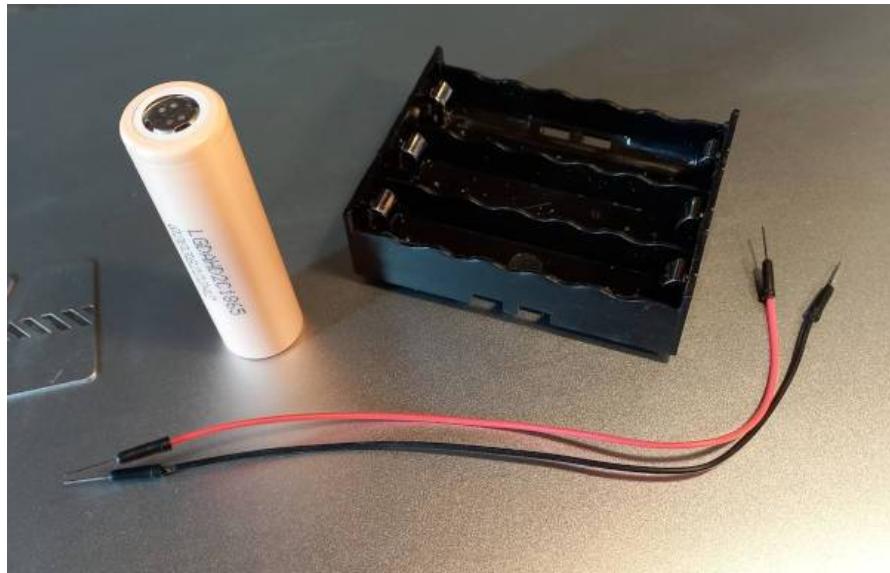
La deuxième option consiste à connecter une alimentation externe non régulée à la broche 5 V et à la masse. Tout ce qui se situe entre 5 et 12 volts devrait fonctionner.

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.

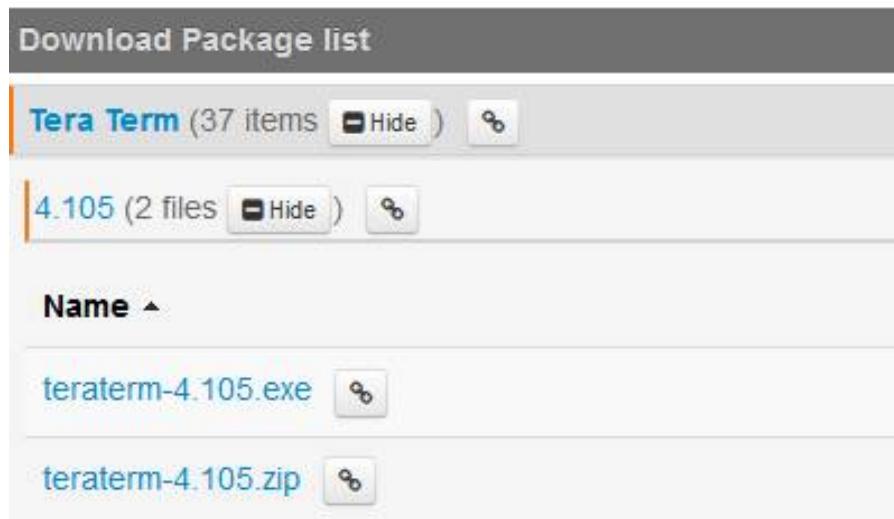
Tout le secret tient dans la séquence **startup: led.blink**. Cette séquence fige le code FORTH compilé par ESP32forth et désigne le mot **led.blink** comme mot à exécuter au démarrage de ESP32forth une fois la carte ESP32 sous tension.

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>



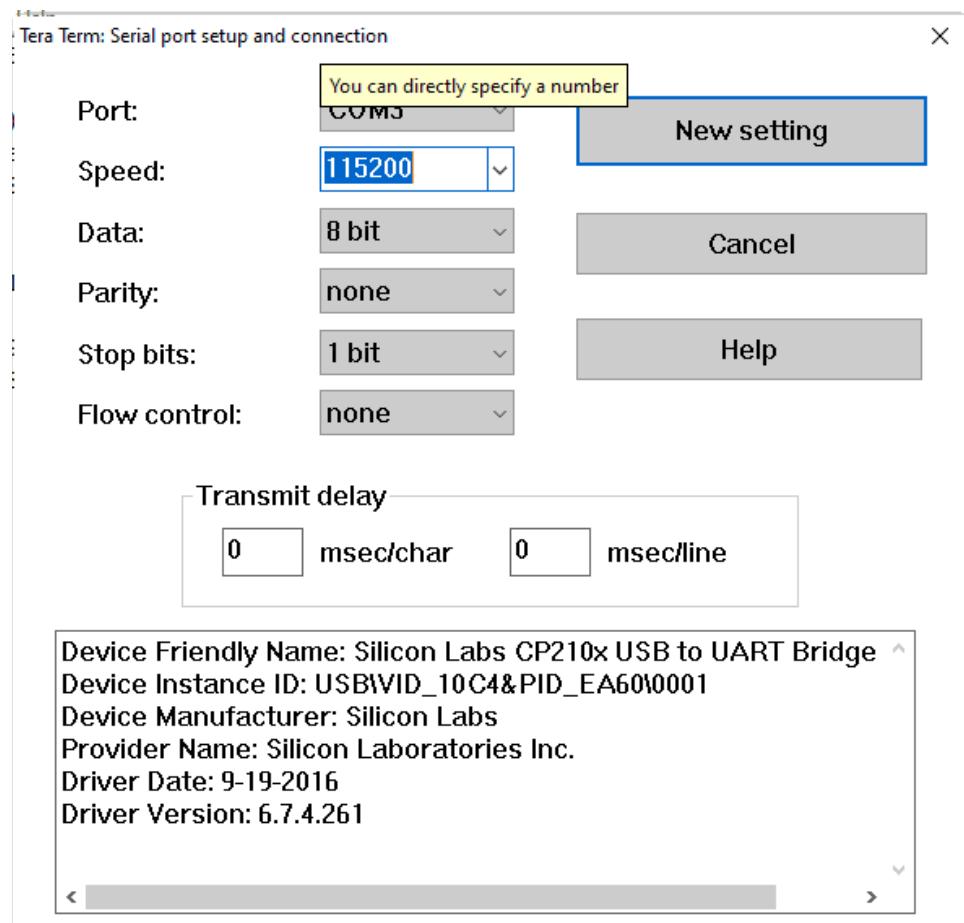
Allez sur la page téléchargement, récupérez le fichier exe ou zip:

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

Paramétrage de Tera Term

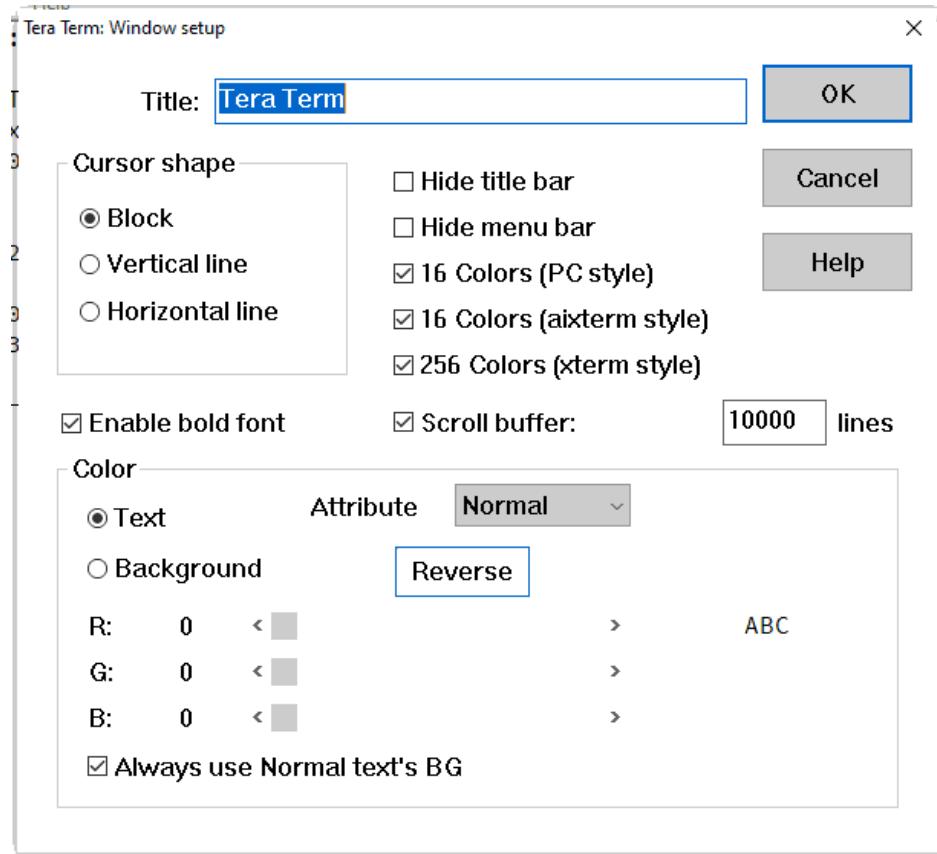
Pour communiquer avec la carte ESP32, il faut régler certains paramètres:

- cliquez sur Configuration -> port série



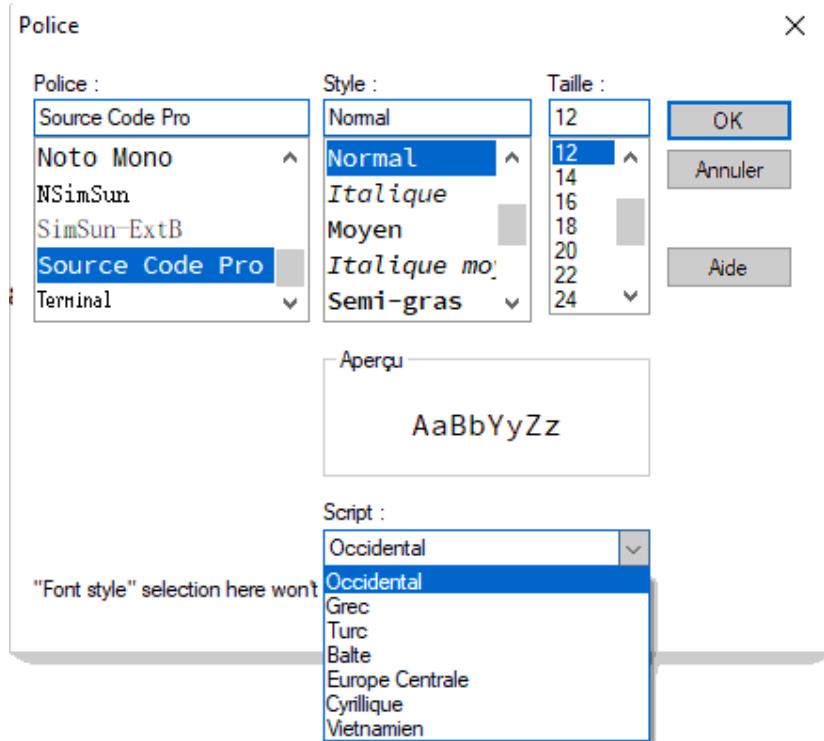
Pour un affichage confortable:

- cliquez sur Configuration -> fenêtre



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**.

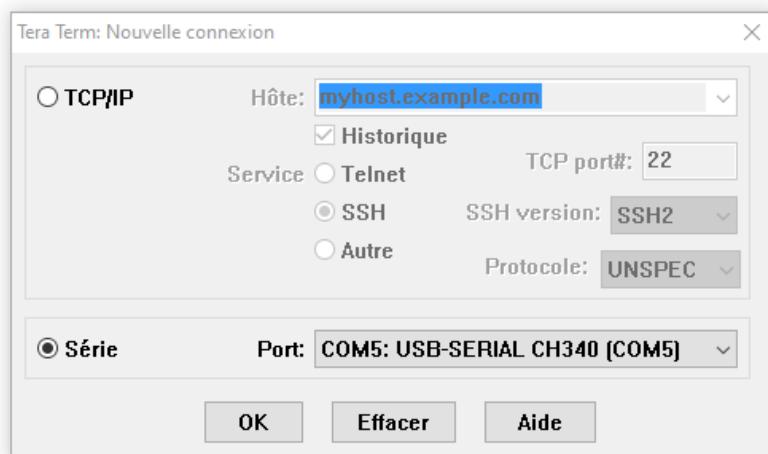
Utilisation de Tera Term

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:



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:

```
\ *** 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 -- fl )
  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.

Accéder à ESP32Forth par TELNET

Avant de gérer une connexion, il faut établir une liaison réseau. La carte ESP32 dispose d'un interface WiFi. Pour établir une liaison WiFi, il faut :

- avoir un modem/routeur qui gère les liaisons en WiFi
- disposer du SSID de port WiFi disponible et de sa clé d'accès

La connexion au réseau WiFi est assurée par le mot **login** :

```
\ connection to local WiFi LAN
: myWiFiConnect ( -- )
  " Mariloo"
  " 1925144D91DE5373C3XXXXXXXXX"
  login
;
```

L'exécution de **myWiFiConnect** affiche :

```
--> myWiFiConnect
192.168.1.8
MDNS started
```

Changer le nom DNS de la carte ESP32

Pour se connecter à une carte ESP32, il y a deux méthodes :

- en connaissant son adresse IP sur le réseau interne. Dans le cas ci-dessus, l'adresse IP est 192.168.1.8. Cette adresse peut changer si elle n'est pas verrouillée par le routeur WiFi;
- par le nom DNS déclaré au moment de la connexion au réseau WiFi. Par défaut, ESP32forth attribue le nom **forth** à la carte qui se connecte au réseau WiFi.

Voici comment initialiser PuTTY pour utiliser le nom d'hôte **forth** au lieu de l'adresse IP:

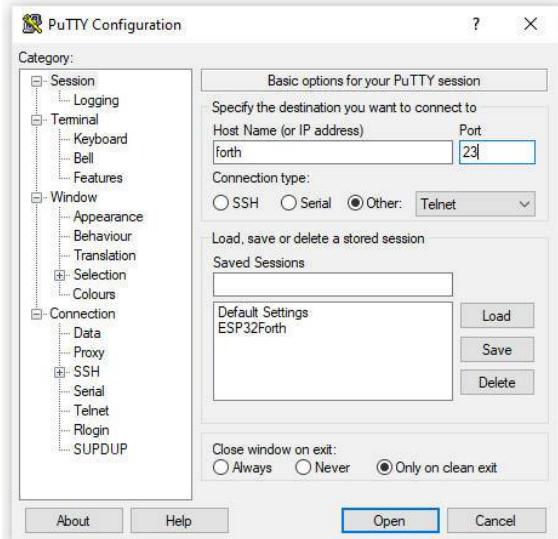


Figure 4: utiliser le nom DNS avec PuTTY

Si on veut communiquer avec plusieurs cartes ESP32 sur un même réseau, il faut que chaque carte déclare un nom d'hôte distinct. Exemple de code pour deux cartes ESP32 :

```
\ set forthCOM3 for 1st ESP32 card
z" Mariloo"
z" 1925144D91DE5373C3C2D7XXXX"
login
z" forthCOM3" MDNS.begin
cr telnetd 552 server
```

Code pour la seconde carte ESP32 :

```
\ set forthCOM6 for 2nd ESP32 card
z" Mariloo"
z" 1925144D91DE5373C3C2D7959F"
login
z" forthCOM6" MDNS.begin
cr telnetd 552 server
```

L'exécution de ce code sur chacune des cartes affecte les noms d'hôte **forthCOM3** et **forthCOM6** sur le réseau interne.

Connexion aux cartes ESP32 par leur nom d'hôte

Lancer PuTTY. On saisit le nom d'hôte et le port ouvert pour accéder à **forthCOM3**:



Figure 5: accès de PuTTY à forthCOM3

Puis on lance une nouvelle session de PuTTY et on change simplement le nom d'hôte pour cette session, ici **forthCOM6**. Voici deux sessions PuTTY permettant de communiquer avec ces deux cartes ESP32 :

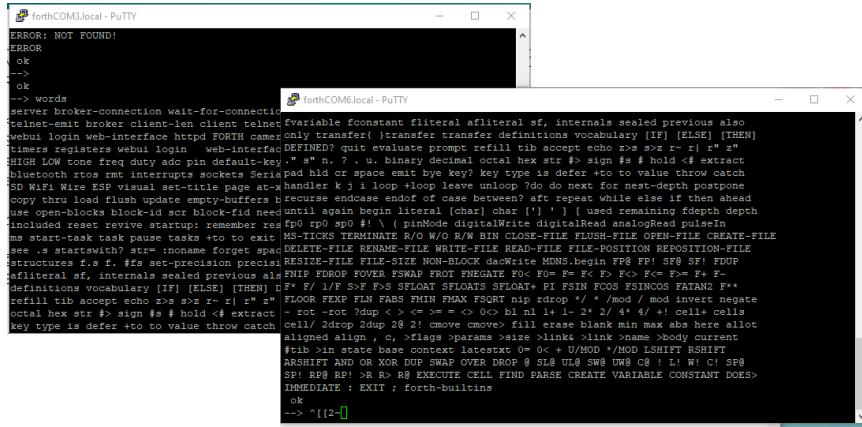


Figure 6: PuTTY accède à deux cartes ESP32 distinctes

Pour lancer automatiquement le client TELNET sur la carte ESP32, on va intégrer notre code de connexion dans **autoexec.fs**. Voici le code à taper depuis le terminal. Tapez d'abord :

```
visual edit /spiffs/autoexec.fs
```

Entrez ensuite ces quelques lignes :

```
\ set forthCOM3 for 1st ESP32 card
z" Mariloo"
z" 1925144D91DE5373C3C2DXXXXX"
login
z" forthCOM3" MDNS.begin
cr telnetd 552 server
forth
```

Faites ensuite CTRL-X et Y. Le code est sauvegardé et sera chargé au prochain démarrage de ESP32forth. Le client TELNET sera relancé automatiquement au démarrage de ESP32forth. Il n'est plus nécessaire d'utiliser le terminal pour communiquer avec la carte ESP32 déclarée avec le nom d'hôte **forthCOM3**:

- débranchez la carte ESP32;

- rebranchez la carte ESP32, mais n'ouvrez pas le terminal!
- attendez quelques secondes...
- lancez puTTY et activez une connexion TELNET avec **forthCOM3** sur le port 552

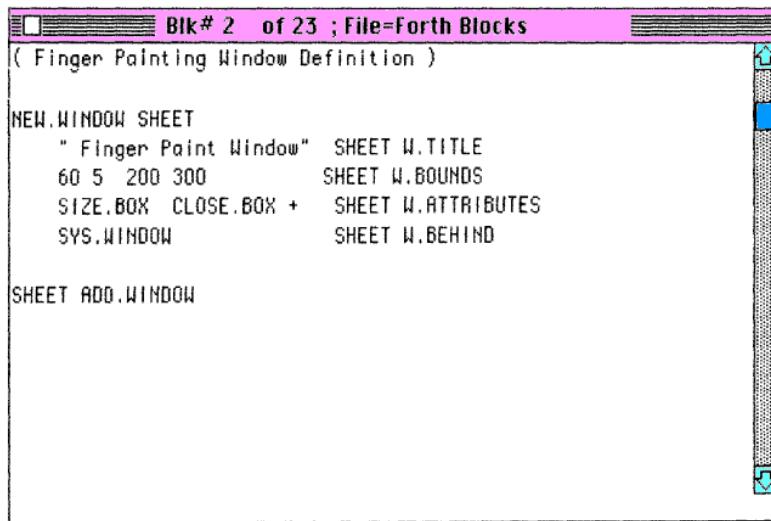
L'accès TELNET par PuTTY permet les mêmes manipulations que par le terminal. Seule restriction : si vous transmettez du code FORTH par copié/collé, limitez la taille du code transmis.

NOTE : les cartes ESP32 ainsi configurées peuvent être accédées depuis Internet si la configuration du routeur WiFi le permet.

Gestion des fichiers sources par blocs

N'utilisez **editor** que pour des éditions de fichiers blocs dans le système de fichiers SPIFFS. Utilisez en priorité **RECORDFILE**. Voir le chapitre *RECORDFILE*.

Les blocs



```
Bk# 2 of 23 ; File=Forth Blocks
( Finger Painting Window Definition )

NEW.WINDOW SHEET
    " Finger Paint Window"  SHEET W.TITLE
    60 5 200 300      SHEET W.BOUNDS
    SIZE.BOX CLOSE.BOX +  SHEET W.ATTRIBUTES
    SYS.WINDOW          SHEET W.BEHIND

SHEET ADD.WINDOW
```

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, celà 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
0000 , 0175 , 0349 , 0523 , 0698 ,
0872 , 1045 , 1219 , 1392 , 1564 ,
1736 , 1908 , 2079 , 2250 , 2419 ,
2588 , 2756 , 2924 , 3090 , 3256 ,
3420 , 3584 , 3746 , 3907 , 4067 ,
4226 , 4384 , 4540 , 4695 , 4848 ,
5000 , 5150 , 5299 , 5446 , 5592 ,
5736 , 5878 , 6018 , 6157 , 6293 ,
| 1
| 2
| 3
| 4
| 5
| 6
| 7
| 8
| 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 celà:

- tapez **flush**, puis débranchez la carte ESP32;
- attendez quelques secondes et rebranchez la carte ESP32;
- tapez **editor** et **l**. 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 aisés 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

N'utilisez **visual edit** que pour des éditions de fichiers sources dans le système de fichiers SPIFFS. Utilisez en priorité **RECORDFILE**. Voir le chapitre *RECORDFILE*.

Editer un fichier source FORTH

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

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

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

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

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

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

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

Notez bien le nom du fichier que vous avez créé.

Choisissez comme extension de fichier **fs**, pour **Forth Source**.

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.

RECORDFILE et gestion de projets FORTH

Ce chapitre est consacré à un seul élément clé : **RECORDFILE**. Ce mot permet un enregistrement rapide de fichiers dans le système de fichiers SPIFFS.

Je vous conseille sa lecture attentive avant de chercher à manipuler des fichiers sources avec **visual** ou **editor**.

Voici étape par étape comment enregistrer la définition de **RECORDFILE**, puis l'utiliser de manière efficace.

Enregistrer RECORDFILE dans le fichier autoexec.fs

Au démarrage de ESP32forth, le système teste la présence du fichier **autoexec.fs**. Si ce fichier est présent, son contenu sera interprété.

Voici le code source de **RECORDFILE**. Cette définition a été mise au point par Bob EDWARDS. Copiez ce code et collez le dans la fenêtre du terminal pour le compiler. Cette manœuvre ne sera à exécuter qu'une seule fois :

```
\ These chars terminate all text lines in a file
create crlf 13 C, 10 C,

\ Records the input stream to a spiffs file until
\ an <EOF> marker is encountered, then close file
: RECORDFILE  ( "filename" "filecontents" "<EOF>" -- )
    bl parse          \ read the filename ( a n )
    W/O CREATE-FILE throw >R  \ create the file to record to -
                           \ put file id on R stack
BEGIN
    \ read a line of the file from the input stream
    tib #tib accept
    tib over
    S" <EOF>" startswith? \ does the line start with <EOF> ?
    DUP IF
        \ Yes, so drop the end line of the file containing <EOF>
        swap drop
    ELSE
        swap
        tib swap
        \ No, so write the line to the open file
        R@ WRITE-FILE throw
        \ and terminate line with cr-lf
        crlf 2 R@ WRITE-FILE throw
    THEN
UNTIL
R> CLOSE-FILE throw
\ repeat until <EOF> found
\ Close the file
;
```

Une fois ce mot compilé, on va voir comment procéder pour que ce mot soit disponible en permanence depuis **autoexec.fs**.

Sur votre PC, dans votre espace de développement dédié à ESP32Forth, créez un fichier **autoexec.fs**.

Copiez dans ce fichier **autoexec.fs** le code de **RECORDFILE** tel que donné ci-avant.

Rajoutez ces deux lignes de code :

```
RECORDFILE /spiffs/autoexec.fs
\ These chars terminate all text lines in a file
create crlf 13 C, 10 C,

\ Records the input stream to a spiffs file until
\ an <EOF> marker is encountered, then close file
: RECORDFILE  ( "filename" "filecontents" "<EOF>" -- )
    bl parse          \ read the filename ( a n )
    W/O CREATE-FILE throw >R  \ create the file to record to -
                           \ put file id on R stack
BEGIN
    \ read a line of the file from the input stream
    tib #tib accept
    tib over
    S" <EOF>" startswith? \ does the line start with <EOF> ?
    DUP IF
        \ Yes, so drop the end line of the file containing <EOF>
        swap drop
    ELSE
        swap
        tib swap
        \ No, so write the line to the open file
        R@ WRITE-FILE throw
        \ and terminate line with cr-lf
        crlf 2 R@ WRITE-FILE throw
    THEN
UNTIL
R> CLOSE-FILE throw
\ repeat until <EOF> found
\ Close the file
;
<EOF>
```

Copiez à nouveau ce code source, en incluant les lignes de code en rouge. Collez à nouveau ce code das la fenêtre du terminal. Transmettez ce code à la carte ESP32.

A la différence de la première manipulation qui consiste à compiler le code, cette fois-ci ce code est enregistré dans le fichier **/spiffs/autoexec.fs**.

Pour vérifier que le fichier **autoexec.fs** est bien enregistré, exécutez **ls** :

```
ls /spiffs/
```

Normalement, le fichier **autoexec.fs** doit apparaître dans la liste des fichiers. Pour vérifier le contenu de **autoexec.fs**, tapez :

```
cat /spiffs/autoexec.fs
```

Ceci doit afficher le contenu de **autoexec.fs**.

Utiliser le contenu modifié du fichier autoexec.fs

Relancez ESP32forth. Si tout s'est bien passé, **RECORDFILE** est maintenant disponible au démarrage de ESP32forth. Exécutez **words**. Vous devez retrouver **RECORDFILE** dans les premiers mots du dictionnaire FORTH :

```
RECORDFILE crlf FORTH spi oled telnetd registers webui login  web-interface
httpd ok LED OUTPUT INPUT HIGH LOW tone freq duty adc pin default-key?
default-key default-type visual set-title page at-xy normal bg fg ansi....
```

N'encombrez pas **autoexec.fs** avec d'autres définitions. Nous allons voir comment créer un projet.

Découpage d'un projet avec ESP32forth

Un projet de développement FORTH pour ESP32forth est élaboré sur votre PC :

- édition du code source avec l'éditeur texte de votre choix ou un IDE (Netbeans par exemple) ;
- avoir un terminal lié par USB à la carte ESP32 ;
- avoir ESP32forth activé sur la carte ESP32.

Sur le PC, travaillez de manière structurée. Les explications qui suivent ne sont que des préconisations.

Commencez par définir le répertoire général de travail pour tous les développements ESP32forth. Par exemple, un dossier nommé **ESP32forth developments**.

Ensuite, dans ce dossier, créer deux dossiers annexes :

- **_my Projects** qui est destiné à accueillir tous vos projets ;
- **_sandbox** qui est destiné à recevoir tous les petits programmes à tester et n'ayant pas d'utilisation précise ;
- **Tools** qui est destiné à accueillir tous les fichiers source ayant un intérêt général. Ce sont des fichiers testés et ne requérant pas d'adaptation ;
- **Documentation** qui est destiné aux documents de toute nature.

Exemple de projet

Je vais reprendre comme exemple de projet le code source de TEMPVS FVGIT. Les codes sources complets sont disponibles ici :

https://github.com/MPETREMANN11/ESP32forth/tree/main/_my%20projects/display/OLED%20SSD1306%20128x32/TEMPVS%20FVGIT

Le premier fichier à créer s'appelle **main.fs**. Ce fichier est à écrire dans un dossier TEMPVS FVGIT :

```
ESP32forth developments
    +-----> _my Projects
        +-----> TEMPVS FVGIT
            +-----> main.fs
                config.fs
                strings.fs
```

Encore une fois, ceci ne sont que des préconisations. L'intérêt principal est de regrouper tous les composants d'un seul projet. Contenu du fichier **main.fs** :

```
RECORDFILE /spiffs/main.fs
DEFINED? --tempusFugit [if] forget --tempusFugit [then]
create --tempusFugit

s" /spiffs/strings.fs"           included
s" /spiffs/RTClock.fs"          included

s" /spiffs/clepsydra.fs"        included

s" /spiffs/config.fs"           included
s" /spiffs/oledTools.fs"        included
( part of code removed here )
<EOF>
```

En rouge, on retrouve notre mot **RECORDFILE**. Pour enregistrer le code de main.fs dans le système de fichiers SPIFFS sur la carte ESP32, il suffit de copier ce code source et de le transmettre à ESP32forth avec le programme terminal.

En bleu, dans le code ci-avant, le contenu de **main.fs** fait un appel au fichier **strings.fs**. Le code source de ce fichier vient du dossier **Tools**. C'est une copie de **strings.fs** qui est ensuite modifié ainsi :

```
RECORDFILE /spiffs/strings.fs
structures
struct __STRING
    ptr field >maxLength      \ point to max length of string
    ptr field >realLength     \ real length of string
    ptr field >strContent     \ string content
forth
( ... removed part of file )
\ work only with strings. Don't use with other arrays
: input$ { addr len -- }
    addr len maxlen$ nip accept
    addr __STRING - cell+ >realLength !
;
<EOF>
```

La copie et la transmission de ce code source crée le fichier **strings.fs** dans le système de fichiers SPIFFS sur la carte ESP32.

A ce stade, on commence à avoir plusieurs fichiers sur la carte ESP32. Pour compiler l'ensemble des fichiers transférés, on exécutera simplement :

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

Il n'y a pas de limite aux fichiers pouvant être enregistrés sur la carte ESP32, hormis une limite d'espace physique. L'espace disponible dans le système de fichiers SPIFFS dépasse 1Mo d'espace d'enregistrement.

Si vous êtes amené à modifier le contenu d'un composant logiciel d'usage général, faites-le toujours sur une copie du fichier source de ce composant. Pensez à versionner et dater ces modifications.

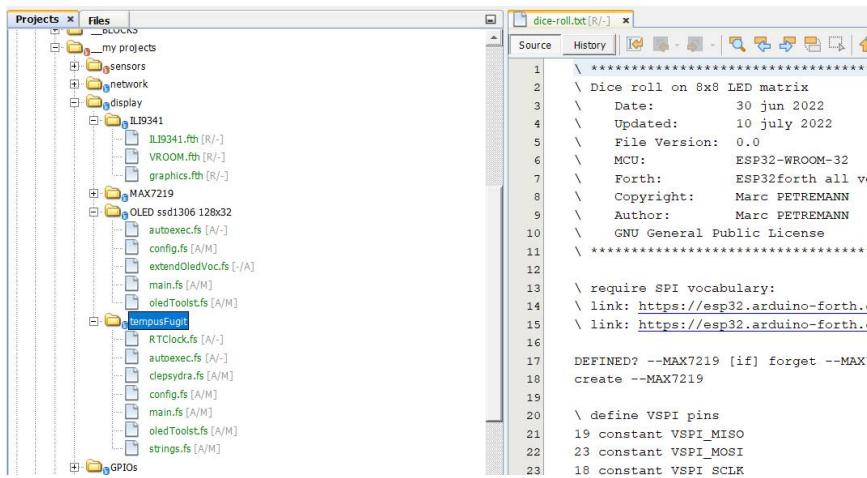


Figure 7: Structuration de projets avec l'IDE Netbeans

Pour chacun des fichiers de ce projet, on intègre **RECORDFILE** et son terminateur **<EOF>**.

Dans chaque projet, on retrouve les fichiers **main.fs** et **config.fs**. Mais leur contenu est adapté à chaque projet. Pour un projet spécifique, tous les fichiers d'extension **fs** sont chargés sur la carte ESP32 dans le système de fichiers SPIFFS. La compilation de leur contenu est incroyablement rapide. Mais surtout, le contenu de ces fichiers **est préservé** entre deux remises en route de la carte ESP32. Au moindre blocage de FORTH il est facile de redémarrer la carte et retrouver toutes les définitions de mots du projet sans nécessiter un nouveau transfert par l'intermédiaire du terminal.

La notion de boîte noire

C'est un concept ancien, qui date de l'époque où on développait surtout en assembleur sur des cartes à micro-contrôleur. On le retrouve également avec les classes en programmation objet. Dans le concept de « boîte noire », il faut considérer un sous-programme, une fonction, une méthode comme une boîte noire. On sait ce qu'on y met. On sait ce qui peut en sortir ou comment cette boîte fonctionne, mais on ne s'occupe pas de son fonctionnement interne. On fait confiance à ceux qui ont programmé la « boîte noire ».

En langage FORTH, un mot a une définition. Quand on passe des paramètres par la pile, au final, seul le concepteur de la définition doit s'assurer du bon fonctionnement de la définition. Et pour s'assurer du bon fonctionnement d'une définition, il est plus que vivement conseillé de ne pas faire des définitions trop longues.

Mon astuce, pour marquer un code vérifié, consiste tout simplement à mettre une ligne de commentaire juste avant la définition. Exemple de code non vérifié :

```
: fpi* ( fn - fn*pi )
    pi f*
;
```

Le code sera testé en *sandbox* ou dans un fichier du projet. Peu importe. Une fois testé avec différentes valeurs, je modifie le code source :

```
\ multiply fn by pi
: fpi* ( fn - fn*pi )
    pi f*
;
```

Si on doute de la fiabilité de son code, on peut définir un fichier **tests.fs**.

Ce fichier est uniquement destiné à réaliser des batteries de tests unitaires. Voir la définition du mot **assert**(qui réalise ces tests, définition visible ici :

<https://github.com/MPETREMANN11/ESP32forth/blob/main/tools/assert.fs>

Et voici un exemple de tests enregistrés dans notre fichier **tests.fs** :

```
assert( 0 >gray 0 = )
assert( 1 >gray 1 = )
assert( 2 >gray 3 = )
assert( 3 >gray 2 = )
assert( 4 >gray 6 = )
assert( 5 >gray 7 = )
assert( 6 >gray 5 = )
assert( 7 >gray 4 = )
```

assert(génère une alerte si le mot testé ne se comporte pas comme prévu.

Pour une définition aussi simple que celle de **fpi***, une batterie de tests peut être nécessaire si on n'a pas fiabilisé le mot **f***. On intègre ces tests dans le fichier **main.fs** :

```
RECORDFILE /spiffs/main.fs
DEFINED? --tempusFugit [if] forget --tempusFugit [then]
create --tempusFugit

s" /spiffs/strings.fs"           included
s" /spiffs/RTClock.fs"          included

s" /spiffs/clepsydra.fs"        included
s" /spiffs/config.fs"           included
s" /spiffs/oledTools.fs"        included
```

```
s" /spiffs/tests.fs"           included  
<EOF>
```

Évidemment, il faut aussi que le contenu du fichier **tests.fs** soit transféré sur la carte ESP32.

De cette manière, le cycle complet de compilation intègre également une batterie de tests. Les tests ne garantissent pas que le code soit fiable. Ils permettent seulement de détecter d'éventuels effets de bord si on est amené à modifier des parties de code de l'application.

En résumé, je conseille de fragmenter votre code en intégrant systématiquement ces fichiers :

- **main.fs** qui est le fichier principal. Avec l'habitude, quelque soit le nom du projet, vous le compilerez avec une simple exécution de **include /spiffs/main.fs**
- **config.fs** qui contient les paramètres de configuration globaux, mots de passe d'un accès WiFi par exemple ;
- **tests.fs** qui contient une batterie de tests. Si vous ne faites aucun test, la création de ce fichier n'est pas nécessaire.

Tous les autres fichiers auront l'extension **fs**, sauf pour les fichiers non traités par ESP32forth.

En suivant ces quelques prescriptions, vous aurez plus de facilité à gérer des applications complexes. Le gain de temps à chaque compilation des parties de code fiables et enregistrées dans des fichiers du système de fichiers SPIFFS est le principal argument pour adopter **RECORDFILE**.

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 réservé 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.

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.

Edition et gestion des fichiers sources pour ESP32forth

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

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

Les éditeurs de fichiers texte

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

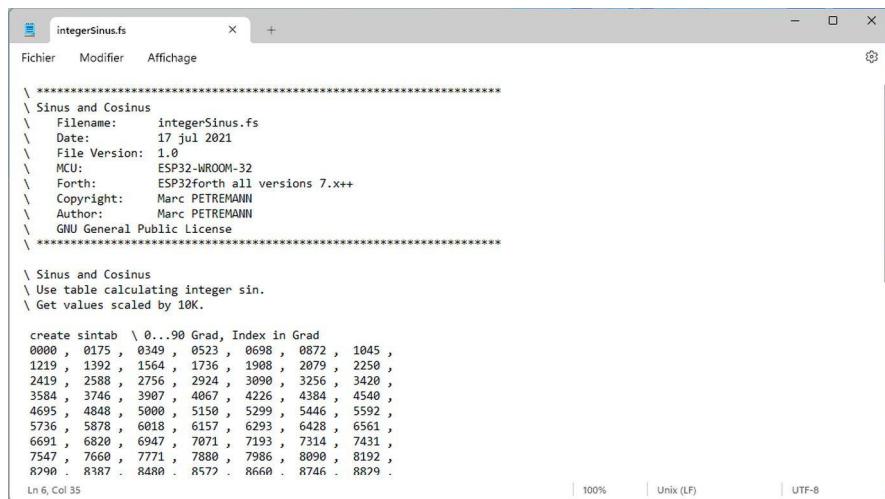


Figure 8: édition avec **edit** sous windows 11

Les autres éditeurs, comme **WordPad** sont déconseillés, car vous risquez de sauvegarder le code source en langage FORTH dans un format de fichier non compatible avec ESP32forth.

Sous Linux, l'équivalent s'appelle **gEdit**. MacOS dispose aussi d'un éditeur de texte simple.

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

Utiliser un IDE

Rien ne vous empêche d'utiliser un IDE⁴. Pour ma part, j'ai une préférence pour **Netbeans** que j'utilise aussi pour PHP, MySQL, Javascript, C, assembleur... C'est un IDE très puissant et aussi performant qu'**Eclipse** :

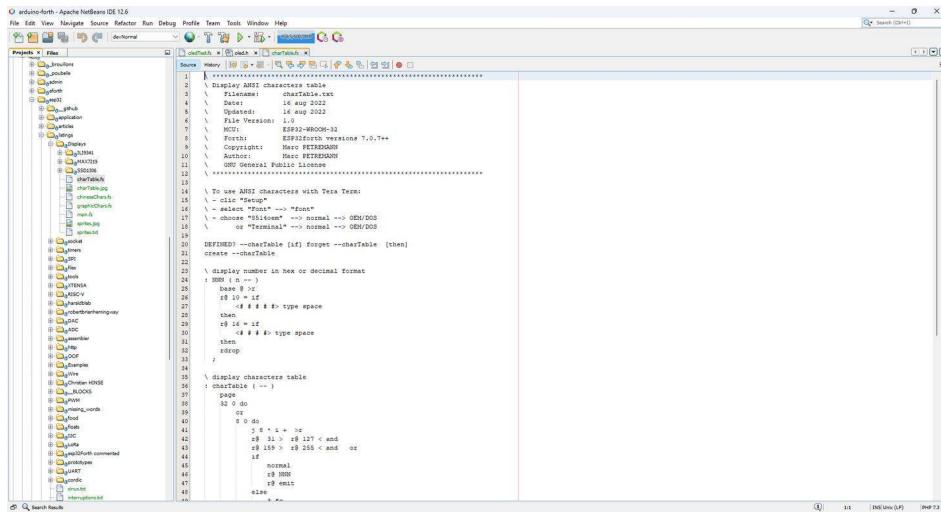


Figure 9: édition avec Netbeans

Netbeans offre plusieurs fonctionnalités intéressantes :

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

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

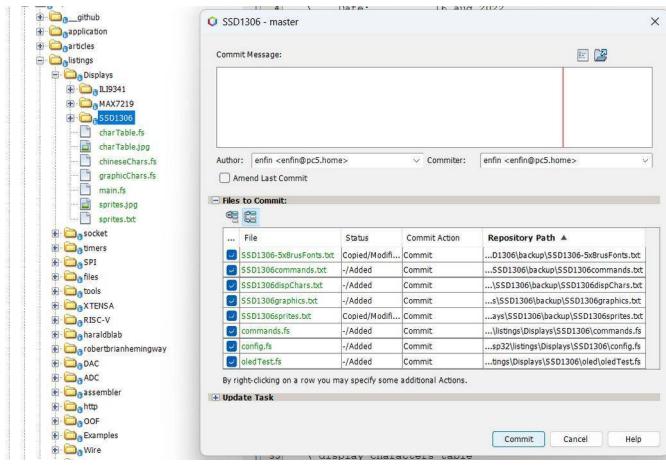
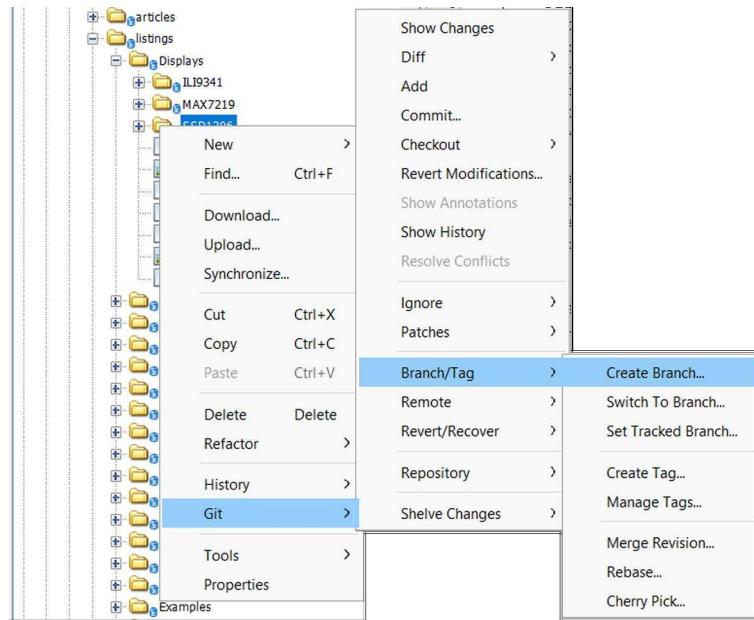


Figure 10: opération GIT dans Netbeans

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



création d'une branche sur un projet

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

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

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

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

Stockage sur GitHub

Le site web **GitHub**⁵ est, avec **SourceForge**⁶, un des meilleurs endroits pour stocker ses fichiers sources. Sur GitHub, vous pouvez mettre un dossier de travail en commun avec d'autres développeurs et gérer des projets complexes. L'éditeur Netbeans peut se connecter au projet et vous permet de transmettre ou récupérer des modifications de fichiers.

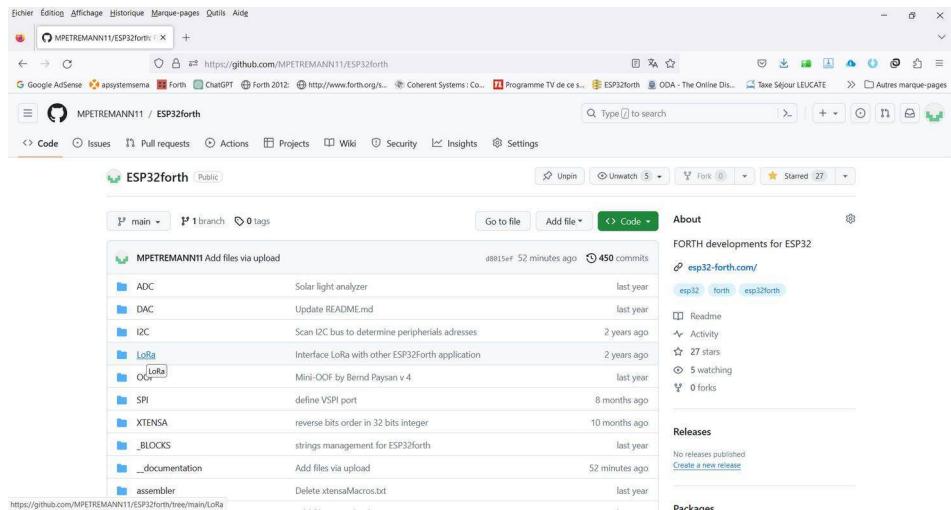


Figure 11: stockage des fichiers sur GitHub

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

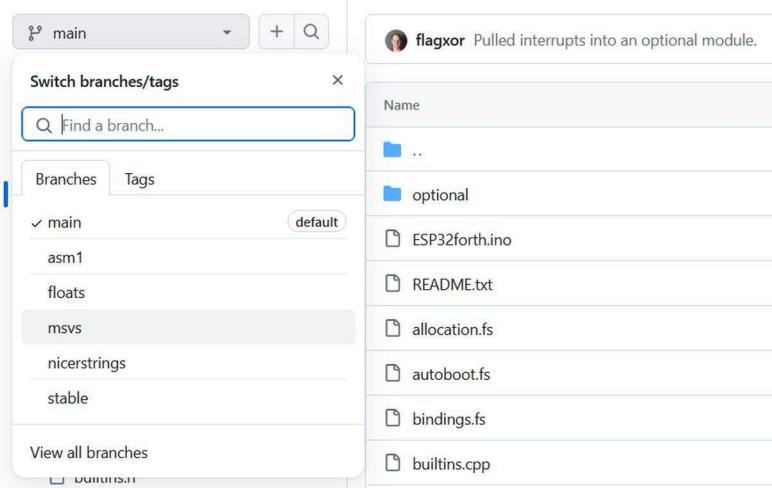


Figure 12: accès à une branche de projet

Quelques bonnes pratiques

La première bonne pratique consiste à bien nommer ses fichiers et dossiers de travail. Vous développez pour ESP32Forth, donc créez un dossier nommé **ESP32forth**.

5 <https://github.com/>

6 <https://sourceforge.net/>

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

Pour les projets bien construits, créez un dossier par projet. Par exemple, vous voulez piloter un robot, créez un sous-dossier **robot**.

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

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

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

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

Figure 13: exemple de nommage de fichiers source Forth

C'est le contenu de ces fichiers qui devra être transféré via le terminal vers la carte ESP32 pour que ESP32forth interprète et compile le code FORTH.

Le fichier **main.fs**

ESP32forth gère un système de fichiers SPIFFS⁷. Voir le chapitre *Le système de fichiers SPIFFS*.

Ces fichiers sont donc stockés dans la carte ESP32 et peuvent être lus par ESP32forth. Si vous avez écrit un fichier **config.fs** dans le système de fichiers SPIFFS, voici la ligne de code à écrire dans **main.fs** pour accéder au contenu de **config.fs** :

```
s" /spiffs/config.fs" included
```

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

⁷ Serial Peripheral Interface Flash File System

```
include /spiffs/config.fs
```

OU

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

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

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

Le traitement de nombreux fichiers prend moins d'une seconde. Cette stratégie évite la transmission répétée du code source par liaison série via le terminal.

Et quand vous gérez plusieurs projets sur plusieurs cartes ESP32, il est plus simple de tester chaque projet avec une simple commande **include /spiffs/main.fs**.

Le chargement des fichiers sera ensuite géré avec RECORDFILE. Voir le chapitre nommé *RECORDFILE*.

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:

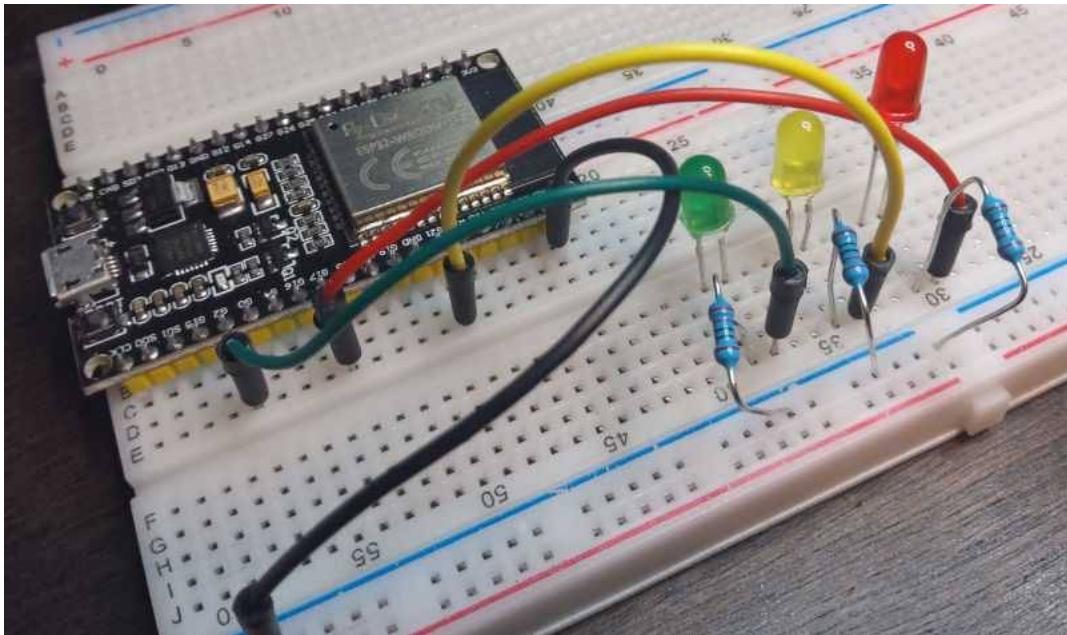
Chaque connecteur est identifié par une série de lettres et chiffres, ici de gauche à droite sur notre photo: G22 TXD RXD G21 GND G19 G18, etc...

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**:

```
\ Use:  
\ numGPIO defPIN: PD7  ( define portD pin #7)  
: defPIN: ( GPIOx --- <word> | <word> --- GPIOx )  
    value  
;  
  
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 :

```
\ traficLights 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 trafic 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, débogguer 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.

Accès direct aux registres GPIO

Dans certaines situations, il est bien plus intéressant d'avoir un accès direct aux registres GPIO sur la carte ESP32. Par exemple, pour gérer des séquences d'activation ou désactivation complexes.

Avec ESP32forth, l'accès à un registre ESP32 s'effectue au moyen des mots **m!** et **m@**. Ces mots permettent en particulier un accès direct aux registres GPIO.

Le registre GPIO gérant les 32 premières entrées/sorties est à l'adresse hexadécimale 3ff44004:

```
$3ff44004 defREG: GPIO_OUT_REG
```

Si on branche une LED au port GPIO2, on peut l'allumer et l'éteindre comme ceci :

```
0 GPIO_OUT_REG m!    \ turn LED on G2 off
4 GPIO_OUT_REG m!    \ turn LED on G2 on
```

L'inconvénient, dans la séquence **4 GPIO_OUT_REG m!**, c'est quelle n'active que le port G2. Si d'autres ports étaient actifs, ils seront désactivés. La solution venant à l'esprit serait donc de lire l'état du registre **GPIO_OUT_REG** à l'aide de **m@** et d'effectuer des opérations logiques sur la valeur avant de la réinjecter par **m!**.

Il se trouve que la carte ESP32 dispose de registres dédiés qui effectuent ces opérations sélectives d'activation et désactivation sans passer par ces opérations logiques sur ce registre **GPIO_OUT_REG**.

Utilisation des mots **m!** et **m@**

Ces deux mots sont définis dans le vocabulaire **registers**. Ce sont d'ailleurs les seuls mots définis dans ce vocabulaire :

- **m!** (val shift mask addr --)
modifie le contenu d'un registre pointé par **addr**, applique un masque logique avec **mask** et décale de **n** bits **val** en fonction de **shift** ;
- **m@** (shift mask addr -- val)
lit le contenu d'un registre pointé par **addr**, applique un masque logique avec **mask** et décale de **n** bits en fonction de **shift**.

Afin de bien comprendre le fonctionnement de ces deux mots, on va d'abord définir un mot qui affiche le contenu 32 bits d'une adresse quelconque :

```
\ display n in bbbbbbbb bbb..... format
: .binDisp ( n -- )
  base @ >r binary  \ select binary base
  <#                  \ start num formating
```

```

4 for
    aft
        8 for
            aft # then
        next
        bl hold      \ add 'space' in number formating
    then
next
#>
cr space ." 33222222 22221111 11111100 00000000"
cr space ." 10987654 32109876 54321098 76543210"
cr type          \ display n in binary format
r> base !       \ restore current numeric base
;

```

Définissons un espace mémoire quelconque, initialisé avec une valeur nulle :

```

create myReg
0 ,

```

Voici comment afficher le contenu de **myReg** avec **.binDisp** :

```

myReg @ .binDisp      \ display :
\ 33222222 22221111 11111100 00000000
\ 10987654 32109876 54321098 76543210
\ 00000000 00000000 00000000 00000000 ok

```

On a mis en évidence le bit b22. Ici il est à zéro. ATTENTION : les bits sont numérotés de 0 à 31 dans l'affichage restitué par **.binDisp**. Voici comment procéder pour mettre ce seul bit b22 à un :

```

registers
1 22 $fffffff myReg m!
forth
myReg @ .binDisp      \ display :
\ 33222222 22221111 11111100 00000000
\ 10987654 32109876 54321098 76543210
\ 00000000 01000000 00000000 00000000 ok

```

C'est fait. Mais on a pris comme masque la valeur 32 bits **\$fffffff**. Le choix de ce masque permet malheureusement une action sur tous les bits du contenu de **myReg**. Si on veut agir seulement sur un ou plusieurs bits, il faut choisir un masque qui limite l'action du mot **m!**. Par exemple, pour mettre à 1 le seul bit b07, il faudra utiliser un masque. Voici ce masque en binaire :

```

00000000 00000000 00000000 10000000

```

Qui se traduit en hexadécimal par **\$00000080**. Si vous avez un doute, on peut vérifier avec **.binDisp** :

```

$00000080 .binDisp      \ display :
\ 33222222 22221111 11111100 00000000
\ 10987654 32109876 54321098 76543210
\ 00000000 00000000 00000000 10000000 ok

```

Ce masque binaire correspond bien au bit b07. On va maintenant mettre ce bit à 1 dans **myReg** sans modifier l'autre bit b22 déjà à un :

```
registers
1 7 $00000080 myReg m!
forth
myReg @ .binDisp      \ display :
\ 33222222 22221111 11111100 00000000
\ 10987654 32109876 54321098 76543210
\ 00000000 01000000 00000000 10000000 ok
```

Parmi les paramètres nécessaires à **m!**, on a la paire shift addr. Pour simplifier le code ESP32forth, on va créer un mot **defMASK** :

```
\ define a mask for registers
: defMASK: ( comp: mask0 position -- <name> | exec: -- position mask1 )
    create
        dup ,
        lshift ,
    does>
        dup @
        swap cell + @
;
```

Pour définir un masque, il faut seulement deux paramètres :

- **mask0** : 1 est le masque minimal sur un bit, 3 sur 2 bits, 7 sur 3 bits, etc...
- **position** : indique la position, dans l'intervalle [0..31]

Pour modifier par exemple le seul bit b12, on peut définir notre masque ainsi :

```
1 12 defMASK: mB12
```

Pour mettre à 1 ce seul bit b12 :

```
registers
1 mB12 myREG m!
forth
myREG @ .binDisp      \ display :
\ 33222222 22221111 11111100 00000000
\ 10987654 32109876 54321098 76543210
\ 00000000 01000000 00010000 10000000 ok
```

Pour remettre à zéro ce bit b12 :

```
registers
0 mB12 myREG m!
forth
myREG @ .binDisp      \ display :
\ 33222222 22221111 11111100 00000000
\ 10987654 32109876 54321098 76543210
\ 00000000 01000000 00000000 10000000 ok
```

Les masques définis par **defMASK**: sont applicables à n'importe quel registre. C'est ce que vous verrez plus loin. Ces masques sont également très utiles pour déterminer l'état de un ou plusieurs bits spécifiques. Testons l'état de notre bit b12 avec **m@** :

```
registers
mB12 myREG m@ .           \ display : 0
forth
```

Remettons ce bit b12 à 1 et testons-le à nouveau :

```
registers
1 mB12 myREG m!
1 mb12 myREG m@ .           \ display : 1
forth
```

Nous avons maintenant les clés pour agir bit à bit sur le contenu de n'importe quel registre. Dans ce chapitre, on va particulièrement s'intéresser au registre **GPIO_OUT_REG** :

```
\ GPIO 0-31 output register R/W
$3FF44004 defREG: GPIO_OUT_REG
```

Le registre **GPIO_OUT_REG**

Ce registre permet de piloter les ports GPIO G0 à G31. On va câbler trois diodes de couleur sur les pins G25, G26 et G27. On définit donc trois constantes rattachées à ces pins :

```
\ definie LEDs GPIOs
25 constant ledRED
26 constant ledYELLOW
27 constant ledGREEN
```

Dans le chapitre *Gérer un feu tricolore avec ESP32*, on avait défini ces mêmes mots avec **defPIN**:. Ici on le fait avec **constant**, ce qui a le même comportement. On va utiliser ces constantes pour définir les masques :

```
\ define masks for red yellow and green LEDs
1 ledRED      defMASK: mLED_RED
1 ledYELLOW   defMASK: mLED_YELLOW
1 ledGREEN    defMASK: mLED_GREEN
```

Pour activer la LED rouge sur G25, on saisit :

```
registers
1 mLED_RED GPIO_OUT_REG m!
forth
```

Pour éviter la sélection récurrente du vocabulaire **registers**, on définit deux mots qui vont nous simplifier la programmation :

```
\ set mask in addr
: regSet ( val shift mask addr -- )
  [ registers ] m! [ forth ]
;
```

```
\ test mask in addr
: regTst ( shift mask addr -- val )
  [ registers ] m@ [ forth ]
;
```

Activation de la LED rouge sur G25 :

```
1 mLED_RED GPIO_OUT_REG regSet
```

Ça fonctionnera, sous réserve que les pins GPIO soient correctement initialisés. C'est ce que nous allons aborder.

Les registres d'activation et désactivation

Les noms des registres sont extraits de la documentation *ESP32 Technical Reference Manual* (pdf):

https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf

Avant d'allumer et éteindre nos LEDs connectées aux ports GPIO G25 G26 et G27, on va commencer par initialiser ces ports. Ceci est effectué en agissant sur le registre

GPIO_ENABLE_REG :

```
: GPIO.init ( -- )
  1 mLED_RED    GPIO_ENABLE_REG regSet
  1 mLED_YELLOW GPIO_ENABLE_REG regSet
  1 mLED_GREEN  GPIO_ENABLE_REG regSet
;
```

L'exécution du mot **GPIO.init** initialise les ports G25 G26 et G27 en sortie. Testons l'allumage des LEDs :

```
GPIO.init
1 mled_red    GPIO_OUT_REG regSet
1 mled_yellow GPIO_OUT_REG regSet
1 mled_green  GPIO_OUT_REG regSet
```

Si les LEDs sont correctement câblées, elles doivent s'allumer. Ici l'allumage des LEDs est effectué séquentiellement par le mot **regSet** répété trois fois. En agissant directement sur le contenu du registre **GPIO_OUT_REG**, on peut effectuer l'allumage des trois LEDs en une seule exécution de **regSet** :

```
GPIO.init
7 mled_red
  mled_yellow nip +
  mled_green  nip + GPIO_OUT_REG regSet
```

Voyons comment exploiter les deux registres **GPIO_OUT_W1TS_REG** et **GPIO_OUT_W1TC_REG** pour agir de manière indirecte sur l'état d'un ou plusieurs ports GPIO :

- **GPIO_OUT_W1TS_REG** : *GPIO Output Write to Set Register* est utilisé pour définir les bits correspondants aux broches GPIO en mode "Output" à l'état logique haut (1). Il permet d'activer les sorties GPIO spécifiées, en mettant les bits correspondants à 1.
- **GPIO_OUT_W1TC_REG** : *GPIO Output Write to Clear Register* est utilisé pour effacer (mettre à zéro) des bits spécifiques dans le registre de sortie GPIO. Chaque bit de ce registre est associé à une broche GPIO particulière, et **en écrivant un 1** dans un bit donné de ce registre, vous pouvez mettre à zéro (effacer) la sortie correspondante de la broche GPIO.

On peut donc allumer toutes nos LEDs en une seule séquence **regSet** comme ceci :

```
7 mled_red
  mled_yellow nip +
  mled_green  nip + GPIO_OUT_W1TS_REG regSet
```

Et pour éteindre toutes les LEDs en une seule séquence **regSet**, on exécutera ceci :

```
7 mled_red
  mled_yellow nip +
  mled_green  nip + GPIO_OUT_W1TS_REG regSet
```

Si on veut gérer un cycle d'allumage et extinction temporisée d'une LED, on va créer un mot qui gère un cycle d'allumage et extinction :

```
\ define a ON and OFF sequence for one LED
: GPIO.on.off.sequence { position mask delay -- }
  1 position mask GPIO_OUT_W1TS_REG regSet
  delay ms
  1 position mask GPIO_OUT_W1TC_REG regSet
;
```

On va tester notre mot **GPIO.on.off.sequence** :

```
mLED_RED 1000 GPIO.on.off.sequence
```

Cette séquence doit allumer la LED rouge pendant une seconde. Définissons maintenant un cycle complet simulant un feu à un carrefour routier :

```
: traffic-light ( -- )
  mLED_GREEN 3000 GPIO.on.off.sequence
  mLED_YELLOW 1000 GPIO.on.off.sequence
  mLED_RED    3000 GPIO.on.off.sequence
;
```

L'exécution de **traffic-light** va simuler un feu routier classique. Cependant, on ne peut pas simuler un feu routier où les feux rouge et orange sont allumés en même temps. On va donc écrire d'abord le mot **TRAFFIC.sequence** qui reprend le code de **GPIO.on.off.sequence**, à la différence qu'il faudra aussi utiliser une valeur en complément des autres paramètres :

```
\ define a ON and OFF sequence
```

```

: TRAFFIC.sequence { val position mask delay -- }
    val position mask GPIO_OUT_W1TS_REG regSet
    delay ms
    val position mask GPIO_OUT_W1TC_REG regSet
;

```

Puis on définit ces quatre mots qui vont permettre de gérer un cycle de feu routier complexe :

```

: TRAFFIC.red ( -- )
    1 mLED_RED    2500 TRAFFIC.sequence ;
: TRAFFIC.yellow ( -- )
    1 mLED_YELLOW 1000 TRAFFIC.sequence ;
: TRAFFIC.green ( -- )
    1 mLED_GREEN   3000 TRAFFIC.sequence ;
: TRAFFIC.red-yellow ( -- )
    3 mLED_RED
    mLED_YELLOW nip + 500 TRAFFIC.sequence ;

```

On définit maintenant un cycle de feu routier tel qu'on pourrait le rencontrer en Allemagne :

```

: TRAFFIC.german.cycle ( -- )
    TRAFFIC.red
    TRAFFIC.red-yellow
    TRAFFIC.green
    TRAFFIC.yellow
;

```

En Allemagne, les feux tricolores routiers ont quatre cycles. La particularité de ces feux est d'allumer simultanément les feux rouge et jaune avant de repasser au feu vert.

Et pour finir, on teste notre cycle de feu routier dans une boucle :

```

: TRAFFIC.loop ( -- )
begin
    TRAFFIC.german.cycle
key? until
;

```

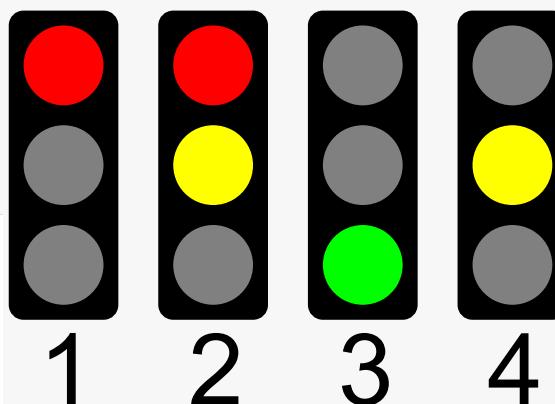


Figure 14: cycle de feux à quatre états

L'exécution de **TRAFFIC.loop** va simuler notre feu routier. Quand arrive la séquence **TRAFFIC.red-yellow**, on peut constater que les feux jaune et rouge sont allumés simultanément.

On a vu dans ce chapitre comment agir sur plusieurs sorties GPIO en même temps en agissant directement sur les registres GPIO.

Cependant, avec les cartes ESP32, il est déconseillé de gérer trop de LEDs simultanément. La carte ESP32 doit être réservée à la gestion des signaux vers des accessoires très peu gourmands en énergie. On évitera donc des montages du style 8 à 16 LEDs pour faire un

effet de type chenillard par exemple, sauf si on exploite des organes de commande utilisant une alimentation séparée.

Accès direct aux registres GPIO

Dans certaines situations, il est bien plus intéressant d'avoir un accès direct aux registres GPIO sur la carte ESP32. Par exemple, pour gérer des séquences d'activation ou désactivation complexes.

Avec ESP32forth, l'accès à un registre ESP32 s'effectue au moyen des mots **m!** et **m@**. Ces mots permettent en particulier un accès direct aux registres GPIO.

Le registre GPIO gérant les 32 premières entrées/sorties est à l'adresse hexadécimale 3ff44004:

```
$3ff44004 defREG: GPIO_OUT_REG
```

Si on branche une LED au port GPIO2, on peut l'allumer et l'éteindre comme ceci :

```
0 GPIO_OUT_REG m!    \ turn LED on G2 off
4 GPIO_OUT_REG m!    \ turn LED on G2 on
```

L'inconvénient, dans la séquence **4 GPIO_OUT_REG m!**, c'est quelle n'active que le port G2. Si d'autres ports étaient actifs, ils seront désactivés. La solution venant à l'esprit serait donc de lire l'état du registre **GPIO_OUT_REG** à l'aide de **m@** et d'effectuer des opérations logiques sur la valeur avant de la réinjecter par **m!**.

Il se trouve que la carte ESP32 dispose de registres dédiés qui effectuent ces opérations sélectives d'activation et désactivation sans passer par ces opérations logiques sur ce registre **GPIO_OUT_REG**.

Utilisation des mots **m!** et **m@**

Ces deux mots sont définis dans le vocabulaire **registers**. Ce sont d'ailleurs les seuls mots définis dans ce vocabulaire :

- **m!** (val shift mask addr --)
modifie le contenu d'un registre pointé par **addr**, applique un masque logique avec **mask** et décale de **n** bits **val** en fonction de **shift** ;
- **m@** (shift mask addr -- val)
lit le contenu d'un registre pointé par **addr**, applique un masque logique avec **mask** et décale de **n** bits en fonction de **shift**.

Afin de bien comprendre le fonctionnement de ces deux mots, on va d'abord définir un mot qui affiche le contenu 32 bits d'une adresse quelconque :

```
\ display n in bbbbbbbb bbb..... format
: .binDisp ( n -- )
  base @ >r binary  \ select binary base
```

```

<#                                \ start num formating
 4 for
    aft
      8 for
        aft  #  then
      next
      bl hold     \ add 'space' in number formating
    then
  next
#>
cr space ." 33222222 22221111 11111100 00000000"
cr space ." 10987654 32109876 54321098 76543210"
cr type          \ display n in binary format
r> base !       \ restore current numeric base
;

```

Définissons un espace mémoire quelconque, initialisé avec une valeur nulle :

```

create myReg
0 ,

```

Voici comment afficher le contenu de **myReg** avec **.binDisp** :

```

myReg @ .binDisp      \ display :
\ 33222222 22221111 11111100 00000000
\ 10987654 32109876 54321098 76543210
\ 00000000 00000000 00000000 00000000 ok

```

On a mis en évidence le bit b22. Ici il est à zéro. ATTENTION : les bits sont numérotés de 0 à 31 dans l'affichage restitué par **.binDisp**. Voici comment procéder pour mettre ce seul bit b22 à un :

```

registers
1 22 $fffffff myReg m!
forth
myReg @ .binDisp      \ display :
\ 33222222 22221111 11111100 00000000
\ 10987654 32109876 54321098 76543210
\ 00000000 01000000 00000000 00000000 ok

```

C'est fait. Mais on a pris comme masque la valeur 32 bits **\$fffffff**. Le choix de ce masque permet malheureusement une action sur tous les bits du contenu de **myReg**. Si on veut agir seulement sur un ou plusieurs bits, il faut choisir un masque qui limite l'action du mot **m!**. Par exemple, pour mettre à 1 le seul bit b07, il faudra utiliser un masque. Voici ce masque en binaire :

```

00000000 00000000 00000000 10000000

```

Qui se traduit en hexadécimal par **\$00000080**. Si vous avez un doute, on peut vérifier avec **.binDisp** :

```

$00000080 .binDisp      \ display :
\ 33222222 22221111 11111100 00000000
\ 10987654 32109876 54321098 76543210

```

```
\ 00000000 00000000 00000000 10000000 ok
```

Ce masque binaire correspond bien au bit b07. On va maintenant mettre ce bit à 1 dans **myReg** sans modifier l'autre bit b22 déjà à un :

```
registers
1 7 $00000080 myReg m!
forth
myReg @ .binDisp      \ display :
\ 33222222 22221111 11111100 00000000
\ 10987654 32109876 54321098 76543210
\ 00000000 01000000 00000000 10000000 ok
```

Parmi les paramètres nécessaires à **m!**, on a la paire shift addr. Pour simplifier le code ESP32forth, on va créer un mot **defMASK:** :

```
\ define a mask for registers
: defMASK: ( comp: mask0 position -- <name> | exec: -- position mask1 )
    create
        dup ,
        lshift ,
    does>
        dup @
        swap cell + @
;
```

Pour définir un masque, il faut seulement deux paramètres :

- **mask0** : 1 est le masque minimal sur un bit, 3 sur 2 bits, 7 sur 3 bits, etc...
- **position** : indique la position, dans l'intervalle [0..31]

Pour modifier par exemple le seul bit b12, on peut définir notre masque ainsi :

```
1 12 defMASK: mB12
```

Pour mettre à 1 ce seul bit b12 :

```
registers
1 mB12 myREG m!
forth
myREG @ .binDisp      \ display :
\ 33222222 22221111 11111100 00000000
\ 10987654 32109876 54321098 76543210
\ 00000000 01000000 00010000 10000000 ok
```

Pour remettre à zéro ce bit b12 :

```
registers
0 mB12 myREG m!
forth
myREG @ .binDisp      \ display :
\ 33222222 22221111 11111100 00000000
\ 10987654 32109876 54321098 76543210
\ 00000000 01000000 00000000 10000000 ok
```

Les masques définis par **defMASK**: sont applicables à n'importe quel registre. C'est ce que vous verrez plus loin. Ces masques sont également très utiles pour déterminer l'état de un ou plusieurs bits spécifiques. Testons l'état de notre bit b12 avec **m@** :

```
registers
mB12 myREG m@ .           \ display : 0
forth
```

Remettons ce bit b12 à 1 et testons-le à nouveau :

```
registers
1 mB12 myREG m!
1 mb12 myREG m@ .           \ display : 1
forth
```

Nous avons maintenant les clés pour agir bit à bit sur le contenu de n'importe quel registre. Dans ce chapitre, on va particulièrement s'intéresser au registre **GPIO_OUT_REG** :

```
\ GPIO 0-31 output register R/W
$3FF44004 defREG: GPIO_OUT_REG
```

Le registre **GPIO_OUT_REG**

Ce registre permet de piloter les ports GPIO G0 à G31. On va câbler trois diodes de couleur sur les pins G25, G26 et G27. On définit donc trois constantes rattachées à ces pins :

```
\ definie LEDs GPIOs
25 constant ledRED
26 constant ledYELLOW
27 constant ledGREEN
```

Dans le chapitre *Gérer un feu tricolore avec ESP32*, on avait défini ces mêmes mots avec **defPIN**:. Ici on le fait avec **constant**, ce qui a le même comportement. On va utiliser ces constantes pour définir les masques :

```
\ define masks for red yellow and green LEDs
1 ledRED      defMASK: mLED_RED
1 ledYELLOW   defMASK: mLED_YELLOW
1 ledGREEN    defMASK: mLED_GREEN
```

Pour activer la LED rouge sur G25, on saisit :

```
registers
1 mLED_RED GPIO_OUT_REG m!
forth
```

Pour éviter la sélection récurrente du vocabulaire **registers**, on définit deux mots qui vont nous simplifier la programmation :

```
\ set mask in addr
: regSet ( val shift mask addr -- )
  [ registers ] m! [ forth ]
```

```

;

\ test mask in addr
: regTst ( shift mask addr -- val )
    [ registers ] m@ [ forth ]
;

```

Activation de la LED rouge sur G25 :

```
1 mLED_RED GPIO_OUT_REG regSet
```

Ça fonctionnera, sous réserve que les pins GPIO soient correctement initialisés. C'est ce que nous allons aborder.

Les registres d'activation et désactivation

Les noms des registres sont extraits de la documentation *ESP32 Technical Reference Manual* (pdf) :

https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf

Avant d'allumer et éteindre nos LEDs connectées aux ports GPIO G25 G26 et G27, on va commencer par initialiser ces ports. Ceci est effectué en agissant sur le registre

GPIO_ENABLE_REG :

```

: GPIO.init ( -- )
  1 mLED_RED      GPIO_ENABLE_REG regSet
  1 mLED_YELLOW  GPIO_ENABLE_REG regSet
  1 mLED_GREEN   GPIO_ENABLE_REG regSet
;
```

L'exécution du mot **GPIO.init** initialise les ports G25 G26 et G27 en sortie. Testons l'allumage des LEDs :

```

GPIO.init
1 mled_red    GPIO_OUT_REG regSet
1 mled_yellow GPIO_OUT_REG regSet
1 mled_green  GPIO_OUT_REG regSet

```

Si les LEDs sont correctement câblées, elles doivent s'allumer. Ici l'allumage des LEDs est effectué séquentiellement par le mot **regSet** répété trois fois. En agissant directement sur le contenu du registre **GPIO_OUT_REG**, on peut effectuer l'allumage des trois LEDs en une seule exécution de **regSet** :

```

GPIO.init
7 mled_red
  mled_yellow nip +
  mled_green  nip + GPIO_OUT_REG regSet

```

Voyons comment exploiter les deux registres **GPIO_OUT_W1TS_REG** et **GPIO_OUT_W1TC_REG** pour agir de manière indirecte sur l'état d'un ou plusieurs ports GPIO :

- **GPIO_OUT_W1TS_REG** : *GPIO Output Write to Set Register* est utilisé pour définir les bits correspondants aux broches GPIO en mode "Output" à l'état logique haut (1). Il permet d'activer les sorties GPIO spécifiées, en mettant les bits correspondants à 1.
- **GPIO_OUT_W1TC_REG** : *GPIO Output Write to Clear Register* est utilisé pour effacer (mettre à zéro) des bits spécifiques dans le registre de sortie GPIO. Chaque bit de ce registre est associé à une broche GPIO particulière, et **en écrivant un 1** dans un bit donné de ce registre, vous pouvez mettre à zéro (effacer) la sortie correspondante de la broche GPIO.

On peut donc allumer toutes nos LEDs en une seule séquence **regSet** comme ceci :

```
7 mled_red
  mled_yellow nip +
  mled_green  nip + GPIO_OUT_W1TS_REG regSet
```

Et pour éteindre toutes les LEDs en une seule séquence **regSet**, on exécutera ceci :

```
7 mled_red
  mled_yellow nip +
  mled_green  nip + GPIO_OUT_W1TS_REG regSet
```

Si on veut gérer un cycle d'allumage et extinction temporisée d'une LED, on va créer un mot qui gère un cycle d'allumage et extinction :

```
\ define a ON and OFF sequence for one LED
: GPIO.on.off.sequence { position mask delay -- }
  1 position mask GPIO_OUT_W1TS_REG regSet
  delay ms
  1 position mask GPIO_OUT_W1TC_REG regSet
;
```

On va tester notre mot **GPIO.on.off.sequence** :

```
mLED_RED 1000 GPIO.on.off.sequence
```

Cette séquence doit allumer la LED rouge pendant une seconde. Définissons maintenant un cycle complet simulant un feu à un carrefour routier :

```
: traffic-light ( -- )
  mLED_GREEN 3000 GPIO.on.off.sequence
  mLED_YELLOW 1000 GPIO.on.off.sequence
  mLED_RED    3000 GPIO.on.off.sequence
;
```

L'exécution de **traffic-light** va simuler un feu routier classique. Cependant, on ne peut pas simuler un feu routier où les feux rouge et orange sont allumés en même temps. On va donc écrire d'abord le mot **TRAFFIC.sequence** qui reprend le code de **GPIO.on.off.sequence**, à la différence qu'il faudra aussi utiliser une valeur en complément des autres paramètres :

```
\ define a ON and OFF sequence
```

```

: TRAFFIC.sequence { val position mask delay -- }
    val position mask GPIO_OUT_W1TS_REG regSet
    delay ms
    val position mask GPIO_OUT_W1TC_REG regSet
;

```

Puis on définit ces quatre mots qui vont permettre de gérer un cycle de feu routier complexe :

```

: TRAFFIC.red ( -- )
    1 mLED_RED    2500 TRAFFIC.sequence ;
: TRAFFIC.yellow ( -- )
    1 mLED_YELLOW 1000 TRAFFIC.sequence ;
: TRAFFIC.green ( -- )
    1 mLED_GREEN   3000 TRAFFIC.sequence ;
: TRAFFIC.red-yellow ( -- )
    3 mLED_RED
    mLED_YELLOW nip + 500 TRAFFIC.sequence ;

```

On définit maintenant un cycle de feu routier tel qu'on pourrait le rencontrer en Allemagne :

```

: TRAFFIC.german.cycle ( -- )
    TRAFFIC.red
    TRAFFIC.red-yellow
    TRAFFIC.green
    TRAFFIC.yellow
;

```

En Allemagne, les feux tricolores routiers ont quatre cycles. La particularité de ces feux est d'allumer simultanément les feux rouge et jaune avant de repasser au feu vert.

Et pour finir, on teste notre cycle de feu routier dans une boucle :

```

: TRAFFIC.loop ( -- )
begin
    TRAFFIC.german.cycle
key? until
;

```

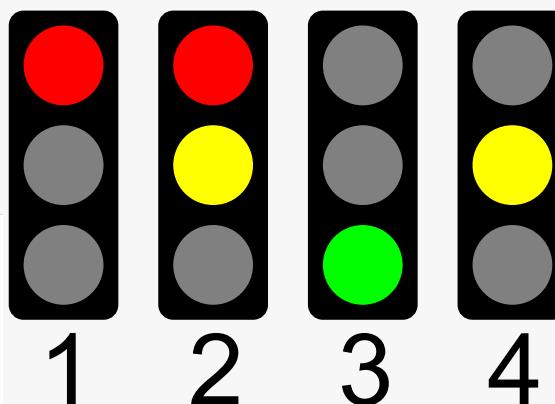


Figure 15: cycle de feux à quatre états

L'exécution de **TRAFFIC.loop** va simuler notre feu routier. Quand arrive la séquence **TRAFFIC.red-yellow**, on peut constater que les feux jaune et rouge sont allumés simultanément.

On a vu dans ce chapitre comment agir sur plusieurs sorties GPIO en même temps en agissant directement sur les registres GPIO.

Cependant, avec les cartes ESP32, il est déconseillé de gérer trop de LEDs simultanément. La carte ESP32 doit être réservée à la gestion des signaux vers des accessoires très peu gourmands en énergie. On évitera donc des montages du style 8 à 16 LEDs pour faire un

effet de type chenillard par exemple, sauf si on exploite des organes de commande utilisant une alimentation séparée.

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 paramétriser 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

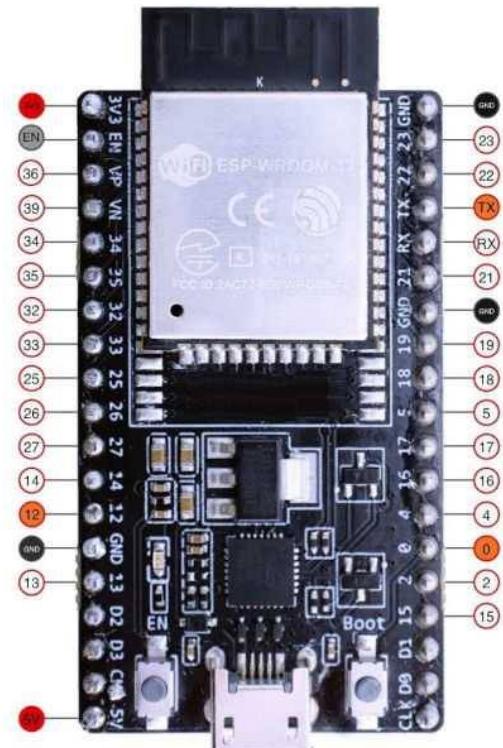
Code FORTH complet avec détection du front montant :

```
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

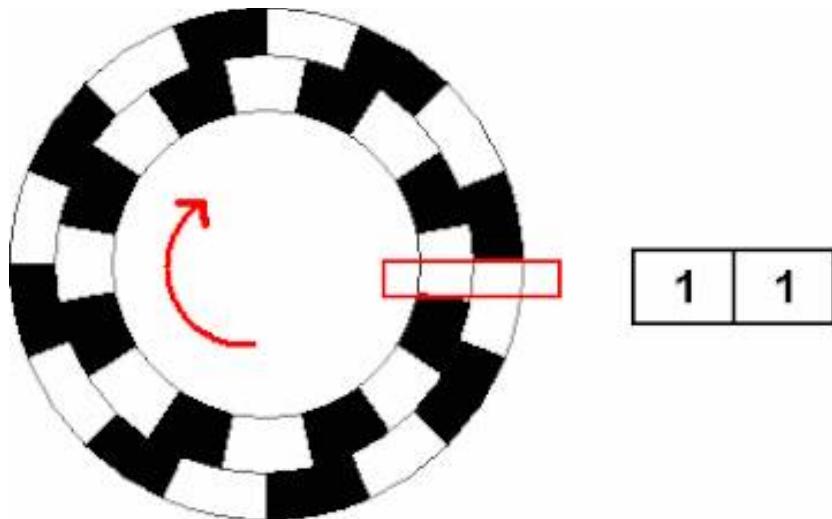
Présentation de l'encodeur

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:

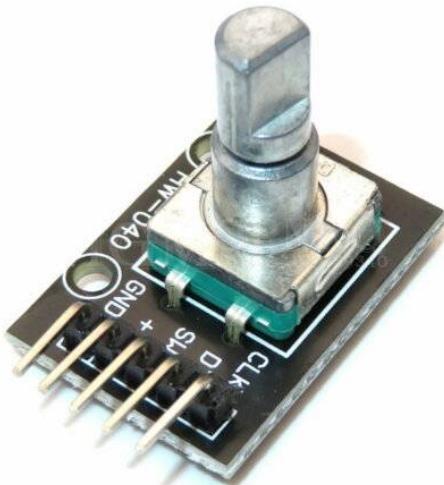


Schéma du fonctionnement interne:

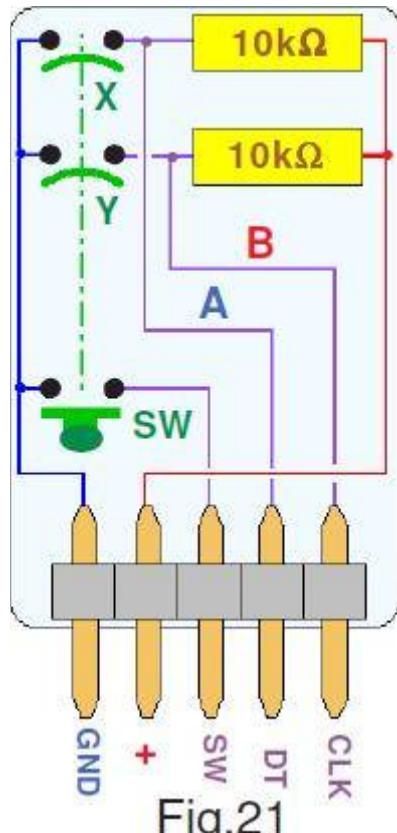


Fig.21

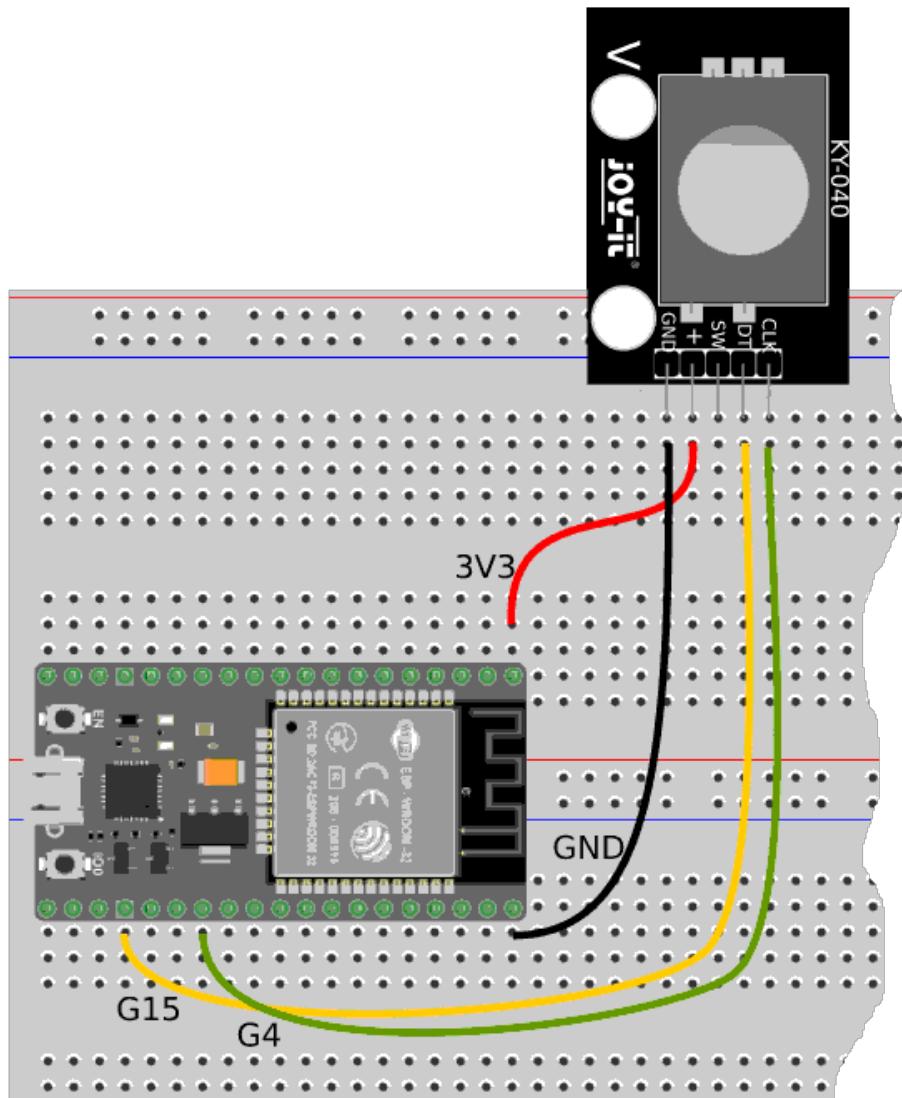
Selon ce schéma, deux bornes nous intéressent:

- 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:

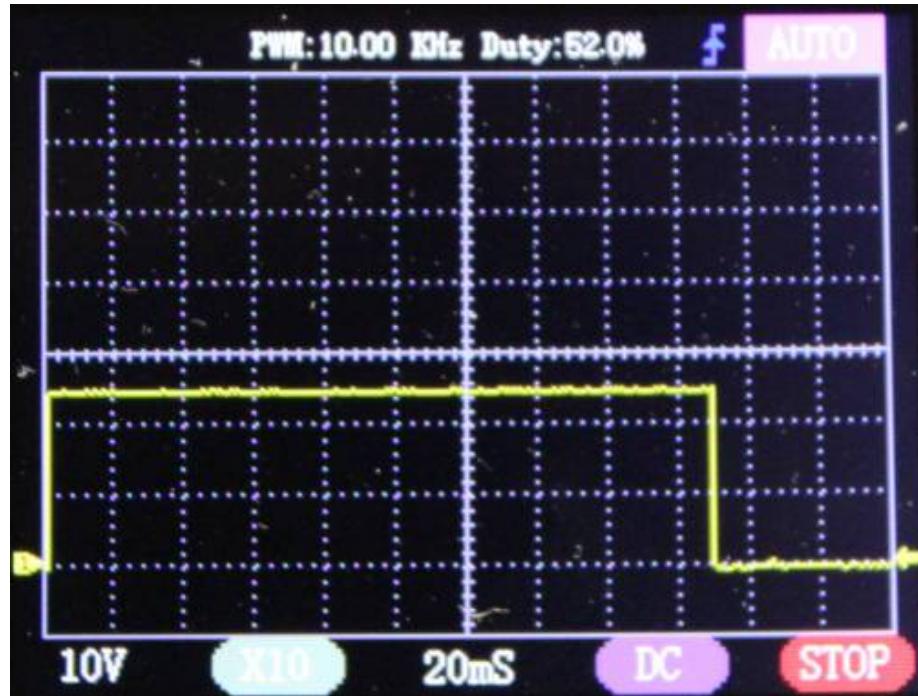


ATTENTION: la position des pins G4 et G15 peut varier selon la version de votre carte ESP32.

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 apparait 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\n\n\\ enable interrupt on GPIO G15\n: intG15enable ( -- )\n    15 GPIO_INTR_POSEDGE gpio_set_intr_type drop\n;\n\n\\ disable interrupt on GPIO G15\n: intG15disable ( -- )\n    15 GPIO_INTR_DISABLE gpio_set_intr_type drop\n;\n\n: pinsInit ( -- )\n    04 input pinmode\n                                \\ GO4 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_PPOSEDGE 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 GIOP18 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 débuter 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 les 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]

Reprendons 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**.

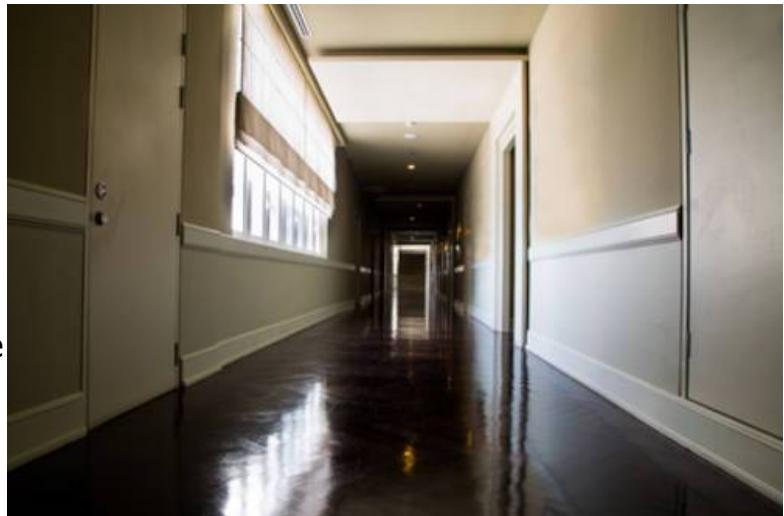
L'interpréteur FORTH de ESP32forth reste accessible pendant le clignotement de la LED, chose impossible en langage C!

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'un 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_EXTENDED_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 délai < 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
```

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 aisément 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** initie 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 dépilage 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 apparait 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

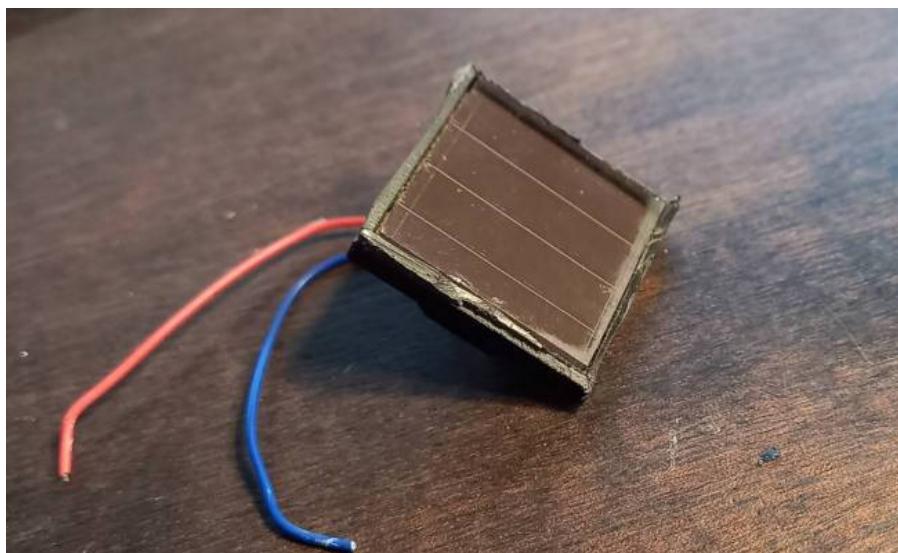
Pour réaliser notre détecteur de nuages, on va utiliser un panneau solaire de très petite taille, ici un panneau de 25mm x 25mm.

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

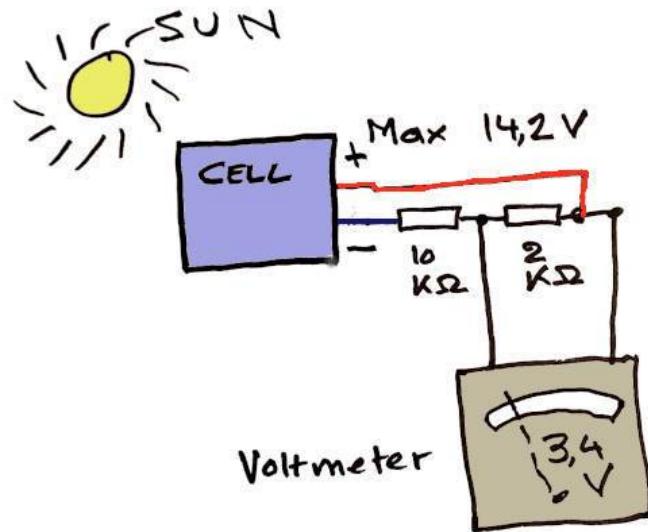
Le courant, c'est à dire l'intensité, doit être relevée à l'aide d'un ampèremètre. La fonction ampèremètre d'un contrôleur universel conviendra. La mise en court-circuit du mini panneau solaire en pleine lumière permet de mesurer un courant de 10 mA.

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 :



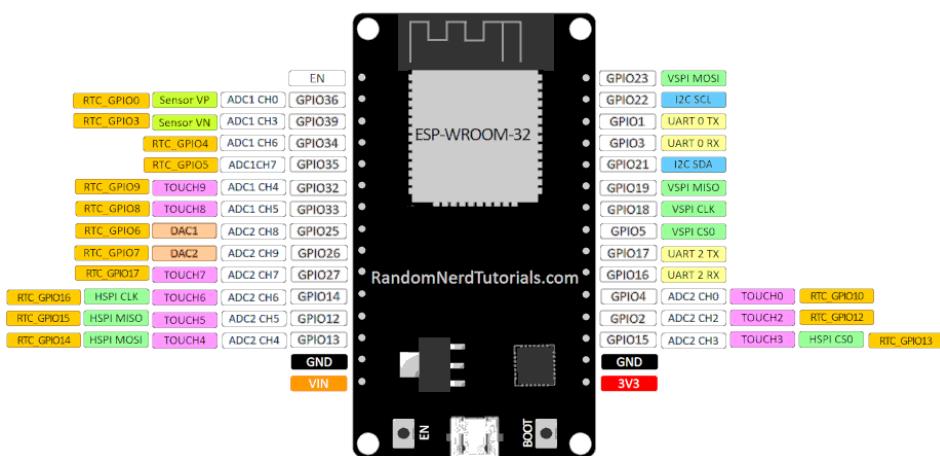
La mesure de tension est relevée entre les deux résistances et la borne positive du panneau solaire.

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/lAjeev2u9fc>

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

Le fichier script FORTH complet **solarLightAnalyzer.fs** est disponible dans le fichier **ESP32forth-book.zip**.

caractéristiques des ports GPIO de la carte ESP32.

Gestion des sorties N/A (Numériques/Analogiques)

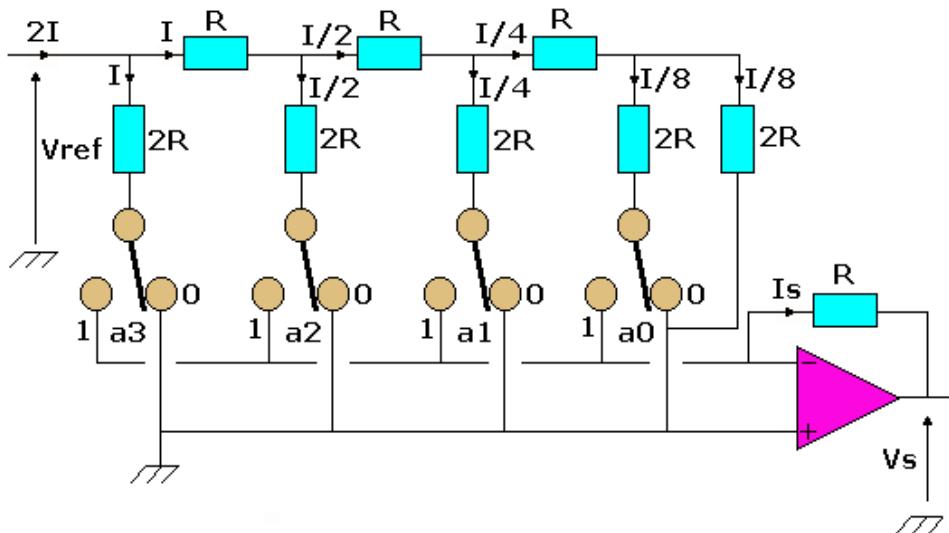
La conversion numérique / analogique

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 a0 à a3. 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 a2 et a0 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:

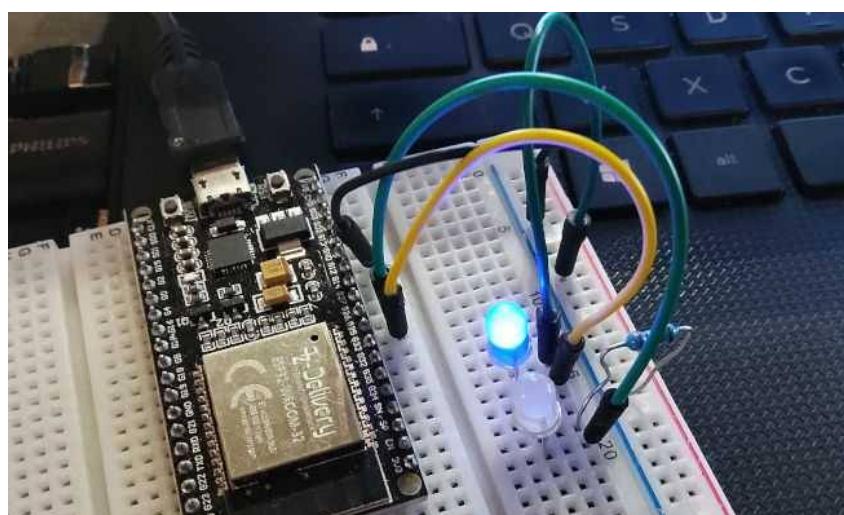
```
\ init Gx as output
: initLeds ( -- )
    ledBLUE  output pinMode
    ledWHITE output pinMode
;
```

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** :

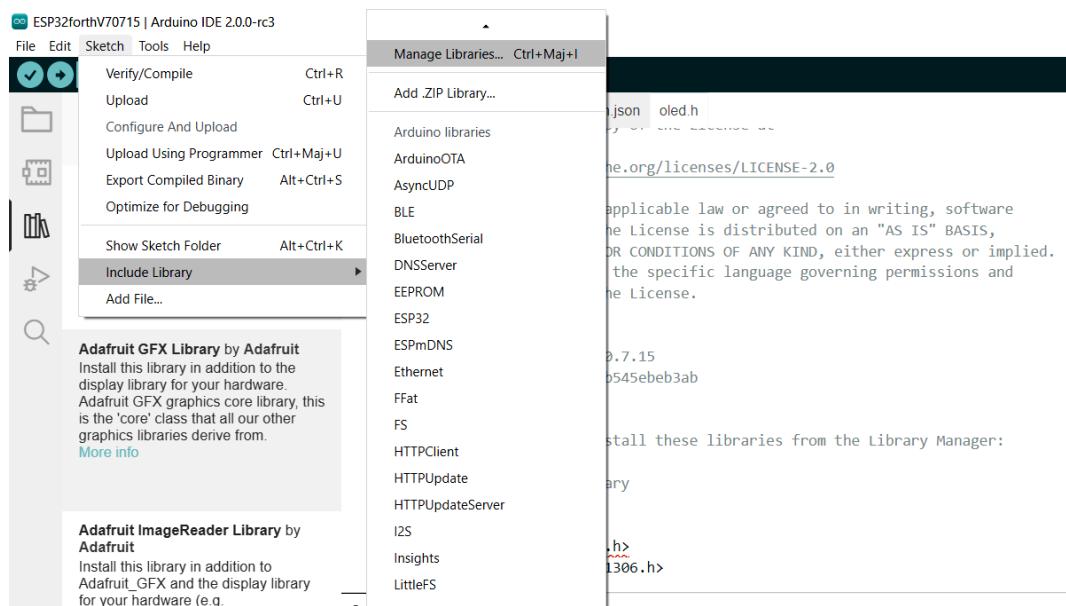
Téléchargements > ESP32forth-7.0.7.15(1).zip > ESP32forth > optional

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 téléchargé 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

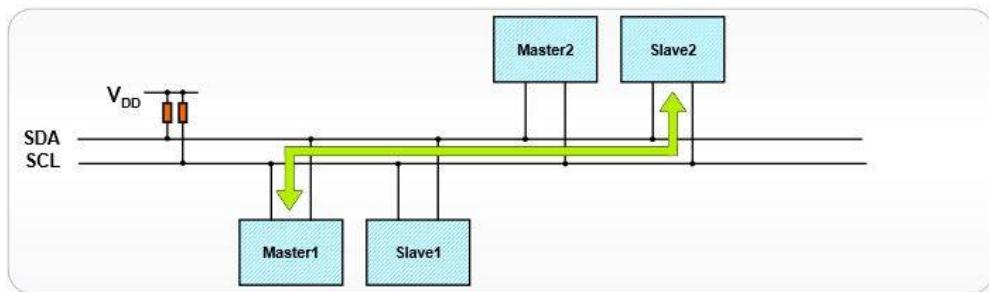
Introduction

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 périphérique, etc.

Il existe d'innombrables périphériques exploitant ce bus, il est même implantable 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.



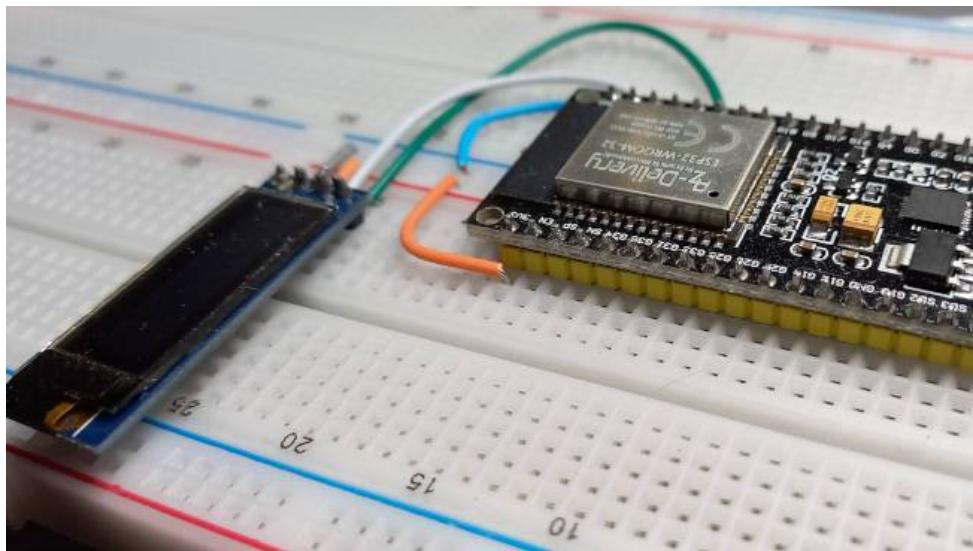
principe d'un bus I2C

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).



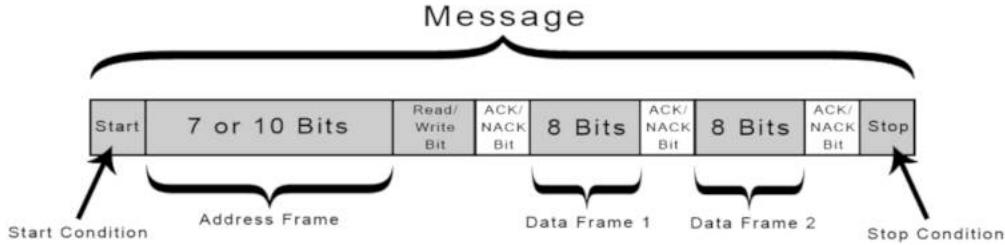
Afficheur OLED connecté au bus I2C

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.beginTransmission
        -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.

Le module ne peut pas être détecté s'il n'est pas physiquement branché et sous tension.

L'afficheur OLED SSD1306

L'afficheur OLED existe en deux définitions :

- 128 x 64 pixels, écran monochrome ou coloré. Si l'écran est coloré, les pixels restent monochrome.
- 128 x 32 pixels, écran monochrome.



Ces afficheurs sont disponibles en interface SPI ou I2C.

Privilégiez l'interface I2C qui permet le branchement de plusieurs interfaces I2C sur le même périphérique I2C. Le vocabulaire **oled** est prévu pour gérer la transmission via I2C vers ces afficheurs OLED.

Choix d'un interface d'affichage

Le choix d'un interface d'affichage est soumis à plusieurs conditions :

- son prix;
- sa consommation électrique;
- sa robustesse;
- sa facilité de programmation et d'utilisation.

Un interface d'affichage s'avère très utile sur un montage autonome pour fournir des informations textuelles ou graphiques très claires.

Après plusieurs recherches, le choix s'est arrêté sur un afficheur OLED de ce type

Il ne coûte que quelques €uros.

Cet afficheur utilise une technologie OLED, donc sans rétro éclairage.

Sa résolution d'affichage est de 128x32 pixels. Il peut afficher du texte et des images, mais seulement en monochrome.



En mode **DISPLAYOFF**, la consommation électrique est quasi nulle.

C'est un produit très répandu et plutôt bien documenté.

Documentation en ligne

- Adafruit: documentation technique et commandes de l'écran OLED
<https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf>
- Adafruit: SSD1306 C library
https://adafruit.github.io/Adafruit_SSD1306/html/files.html
- Adafruit: SSD1306 microPython
<https://github.com/adafruit/micropython-adafruit-ssd1306>
- Punyforth: SSD1306 SPI Forth
<https://github.com/zeroflag/punyforth/blob/master/arch/esp8266/forth/ssd1306-spi.forth>
- TG9541: Forth Oled Display
<https://github.com/TG9541/forth-oled-display/blob/master/ssd1306.fs>
- Yunfan: SSD1306 128x32 i2c forth
<https://gist.github.com/yunfan/2d3ee14697f3ebd3cb43ae411216d9aa>

Branchement de l'afficheur OLED SSD1306

L'afficheur OLED SSD1306 128x32 doit être utilisé sur le bus I2C de la carte ESP32.

Ce bus I2C est présent sur toutes les cartes ESP32.

Branchement sur une carte ESP32:

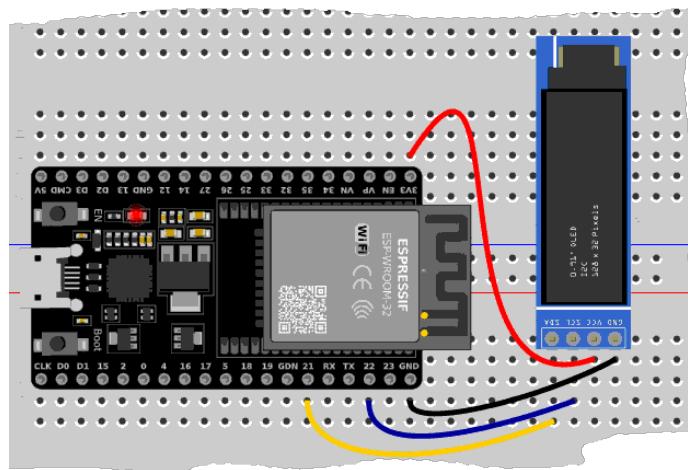


Figure 16: branchement de l'afficheur OLED SSD1306 I2C

Comme on le voit, 4 fils suffisent: 2 pour l'alimentation de l'afficheur OLED SSD1306 (fils noir et rouge), 2 pour la liaison sur le bus I2C (fils bleu et jaune).

L'alimentation de l'afficheur est reprise sur la carte ESP32. Il n'est pas nécessaire d'utiliser une alimentation auxiliaire. La très faible consommation électrique de cet afficheur le permet. L'afficheur OLED SSD1306 dispose d'un circuit intégré ramenant à la tension 5V nécessaire à son fonctionnement.

Organisation de la mémoire

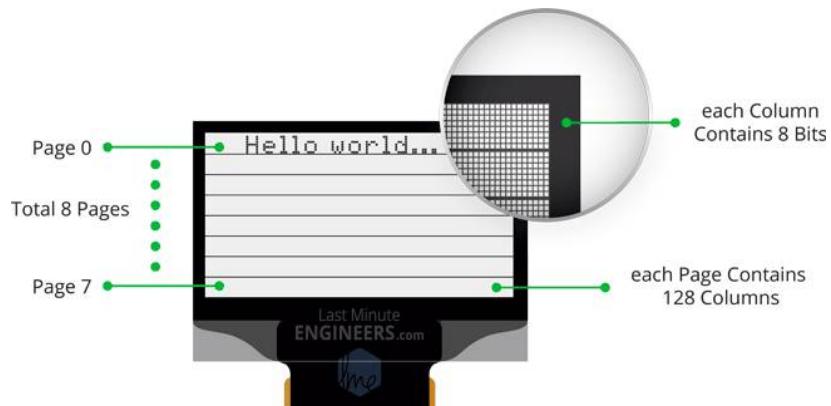
L'écran de l'afficheur SSD1306 128x32 utilise le même composant interne que l'afficheur SSD1306 128x64. La mémoire interne est commune aux deux modèles, l'écran de l'afficheur 128x32 n'utilise qu'une partie de cette mémoire.

La mémoire interne de l'afficheur dispose de 1Ko RAM.

Sur ce schéma, voici l'organisation de l'écran pour une définition de 128x64 pixels :

Chaque colonne contient 8 bits. Une ligne est désignée par page:

- l'afficheur 128x64: contient 8 pages, numérotées de 0 à 7
- l'afficheur 128x32: contient 4 pages, numérotées de 0 à 3



Chaque page est divisée en segments :

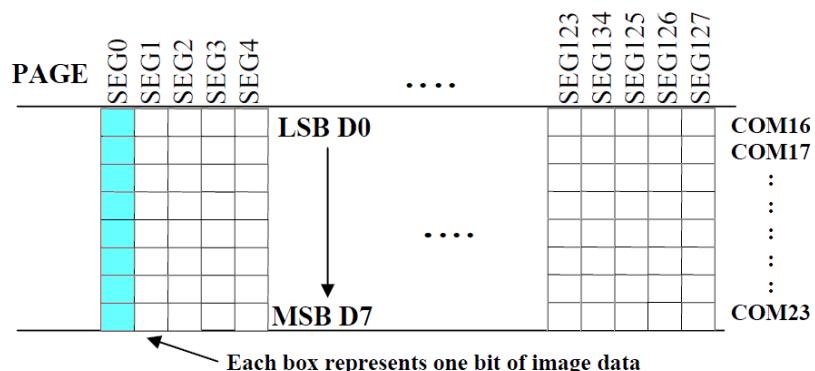


Figure 17: segmentation de l'espace mémoire de l'afficheur

Ici, en bleu sur la figure, un segment représente un octet. Le bit de poids faible est en haut.

Nous n'avons pas besoin d'aller plus loin pour gérer cet afficheur avec le vocabulaire **oled**.

Organiser le projet SSD1306

Avant de rentrer dans le vif du sujet, voyons comment nous allons organiser notre projet. Sur votre ordinateur, créez un dossier de travail nommé **display**. Dans ce dossier, créez un sous-dossier **SSD1306**.

Nous allons pleinement exploiter la gestion des fichiers SPIFFS et la construction de notre code FORTH dans un vrai projet.

Comme pré-requis, vous devez avoir la définition de RECORDFILE enregistrée dans le fichier **/spiffs/autoexec.fs**.

Création du fichier main.fs

Sur votre ordinateur, dans le sous-dossier **SSD1306**, créez le fichier **main.fs** et copiez-y ce code FORTH :

```
RECORDFILE /spiffs/main.fs
DEFINED? --oledTest [if] forget --oledTest [then]
create --oledTest
s" /spiffs/config.fs"      included
s" /spiffs/oledTools.fs"   included
<EOF>
```

Copiez ce code à nouveau, lancez le terminal qui communique avec la carte ESP32 et ESP32forth. Copiez ce code dans le terminal. Exécutez-le. En fin d'exécution, vous devriez retrouver votre fichier main.fs sur la carte ESP32 :

```
--> ls /spiffs/
autoexec.fs
main.fs
```

Pour vérifier que le contenu de **main.fs** a bien été enregistré dans le système de fichiers SPIFFS :

```
cat /spiffs/main.fs
```

doit afficher le contenu du fichier **/spiffs/main.fs**.

Création du fichier config.fs

Sur votre ordinateur, dans le sous-dossier **SSD1306**, créez le fichier **config.fs** et copiez-y ce code FORTH :

```
RECORDFILE /spiffs/config.fs
\ set oled SSD1306 dimensions
oled
128 to WIDTH
32 to HEIGHT
forth
\ set address of OLED SSD1306 display 128x32 pixels
$3c constant I2C_SSD1306_ADDRESS
```

```
<EOF>
```

Comme pour **main.fs**, transmettez ce contenu via le terminal pour enregistrer le nouveau fichier **config.fs**.

Création du fichier **oledTools.fs**

Pour le moment, voici notre dernier fichier **oledTools.fs** à créer sur l'ordinateur et à transmettre vers la carte ESP32 :

```
RECORDFILE /spiffs/oledTools.fs
oled
: Oled128x32Init
    OledAddr @ 0=
    if
        WIDTH HEIGHT OledReset OledNew
        SSD1306_SWITCHCAPVCC I2C_SSD1306_ADDRESS OledBegin drop
    then
        OledCLS
        1 OledTextsize      \ Draw 2x Scale Text
        WHITE OledTextc     \ Draw white text
        0 0 OledSetCursor   \ Start at top-left corner
        z" *Esp32forth*" OledPrintln OledDisplay
    ;
forth
<EOF>
```

Tester notre projet **SSD1306**

Voici la structure de notre projet sur le disque de notre ordinateur.

Tous les fichiers d'extension fs ont été transmis à la carte ESP32 pour être enregistrés sur le système de fichiers SPIFFS.

Pour compiler le contenu du fichier /SPIFFS/config.fs, on peut tester ceci depuis la fenêtre du terminal qui communique avec ESP32forth :

```
include /spiffs/config.fs
```

Si vous ne modifiez jamais le contenu du fichier **config.fs**, il sera toujours disponible pour ESP32forth dès la mise sous tension de la carte ESP32.

Vous rappelez-vous que nous avons également transmis un fichier **main.fs** ? Le contenu de ce fichier doit être réservé à l'appel des différents fichiers du projet. Rappel du contenu de notre fichier main.fs tel qu'il est enregistré sur la carte ESP32 dans le système de fichiers SPIFFS :

```
DEFINED? --oledTest [if] forget --oledTest [then]
```

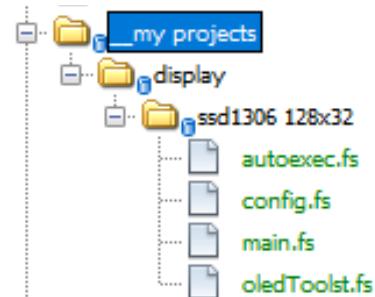


Figure 18: fichiers du projet

```
create --oledTest

s" /spiffs/config.fs"      included
s" /spiffs/oledTools.fs"   included
```

Les deux premières lignes permettent de gérer un marqueur. A chaque interprétation du contenu de **main.fs**, ESP32forth va tester s'il existe un mot **--oledTest**. Si ce mot existe, il sera supprimé du dictionnaire. Tous les mots compilés après **--oledTest** seront supprimés du dictionnaire.

A la seconde ligne, on recrée le mot **--oledTest**. Ce n'est pas surprenant de supprimer ce mot et de le recréer. De cette manière, à chaque interprétation du contenu de **main.fs**, le dictionnaire de ESP32forth redémarre avec un contenu qui ne perturbera pas notre projet.

Enfin, dans les deux dernières lignes de **main.fs**, on demande à ESP32forth de traiter le contenu des fichiers **config.fs** et **oledTools.fs**. Donc, pour lancer ce traitement global, on peut taper :

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

Quand on a un projet complexe, on peut être amené à taper ce traitement des dizaines, voire des centaines de fois. Vous rappelez-vous que dans le fichier **autoexec.fs** on a défini le mot **MAIN** ? Rappel de la définition de ce mot :

```
: MAIN
  s" /spiffs/main.fs"      included
;
```

Trop bien ! Donc, on peut simplement taper **MAIN** au lieu de faire **include /spiffs/main.fs...**

On fait le test ? OK. Mettez hors tension la carte ESP32. Remettez-la sous tension. Ouvrez le terminal qui communique avec ESP32forth et tapez **MAIN**. Tout le contenu du projet est exécuté et compilé quasi instantanément ! Tapez **words**. Tous les mots de notre projet sont dans le vocabulaire **forth**.

On vérifie que ça fonctionne en tapant maintenant :

```
oled128x32Init
```

Si l'afficheur OLED SSD1306 128x32 pixels est bien branché, il doit afficher ***Esp32forth*** :

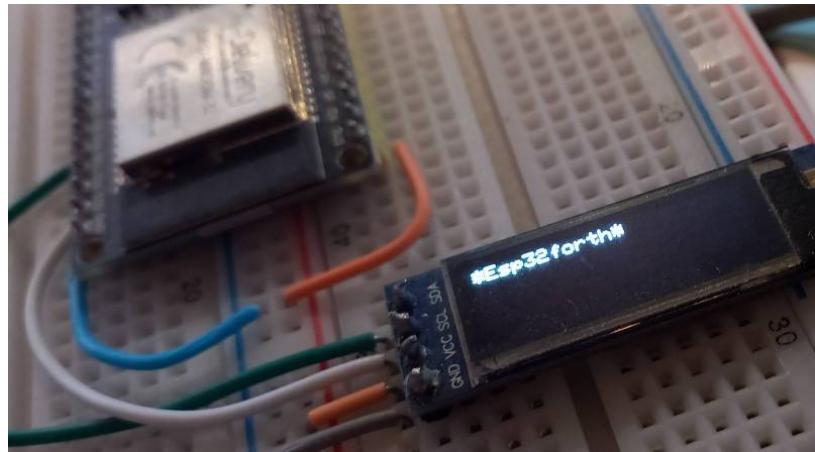


Figure 19: résultat de l'exécution du mot Oled128x32Init

A partir de ce moment, nous pouvons écrire et réaliser tous les autres mots de notre projet.

Exploiter le vocabulaire oled

Les mots que nous allons exploiter pour utiliser notre afficheur OLED SSD1306 128x32 sont disponibles dans le vocabulaire **oled** :

```
--> oled vlist
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 OledFastHLine OledFastVLine OledCirc
OledCircF OledRect OledRectF OledRectR OledRectRF oled-builtins
```

Initialisation du bus I2C pour l'afficheur OLED SSD1306

Notre afficheur OLED est câblé sur le bus I2C. Il est donc disponible à l'adresse hexadécimale 3c sur ce bus I2C. La première chose à faire est donc de définir une constante :

```
\ set adress of OLED SSD1306 display 128x32 pixels
$3c constant I2C_SSD1306_ADDRESS
```

Dans le vocabulaire oled, il y a une variable **OledAddr** chargée de mémoriser cette adresse 3c. Si cette adresse contient une valeur nulle, c'est que le bus I2C n'a pas établi de connexion avec notre afficheur SSD1306. C'est cette valeur nulle qui conditionne l'initialisation de cette connexion :

```
OledAddr @ 0=
if
    \ init I2C communication with SSD1306
then
```

Voici la partie initialisation de notre communication via le bus I2C vers l'afficheur OLED SSD1306 :

```

WIDTH HEIGHT OledReset OledNew
SSD1306_SWITCHCAPVCC I2C_SSD1306_ADDRESS OledBegin drop

```

- la séquence **WIDTH HEIGHT OledReset OledNew** instancie une nouvelle session oled pour notre afficheur SSD1306 ;
- le mot OledBegin sera précédé de deux paramètres :
 - **SSD1306_SWITCHCAPVCC** qui confirme l'alimentation en 3,3V depuis la carte ESP32, ce qui est le cas de notre montage. Si on avait utilisé une alimentation externe, on remplace ce mot par **SSD1306_EXTERNALVCC**.
 - **I2C_SSD1306_ADDRESS** qui indique l'adresse de l'afficheur OLED sur le bus I2C.

Cette initialisation du bus I2C n'est réalisée qu'une seule fois pour notre afficheur OLED SSD1306.

Initialisation de l'affichage pour SSD1306

Pour démarrer un affichage, on va initialiser l'afficheur :

- **OledCLS** qui demande un effacement du contenu de l'écran ;
- **1 OledTextsize** qui indique la taille du texte à afficher ;
- **WHITE OledTextc** qui indique la couleur du texte à afficher. Pour l'afficheur SSD1306, il n'existe que les couleurs **WHITE** et **BLACK**.

Voici le mot **Oled128x32Init** permettant une initialisation dans les règles :

```

oled
: Oled128x32Init
  OledAddr @ 0=
  if
    WIDTH HEIGHT OledReset OledNew
    SSD1306_SWITCHCAPVCC I2C_SSD1306_ADDRESS OledBegin drop
  then
    OledCLS
    1 OledTextsize      \ Draw 1x Scale Text
    WHITE OledTextc     \ Draw white text
    0 0 OledSetCursor   \ Start at top-left corner
    z" *Esp32forth*" OledPrintln OledDisplay
  ;
forth

```

L'exécution de **Oled128x32Init** doit afficher le texte ***Esp32forth*** sur l'écran OLED.

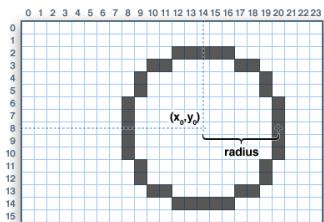
Voici un résumé des commandes de gestion d'affichage texte pour l'afficheur OLED :

- **OledCLS (--)**
Efface le contenu de l'écran OLED

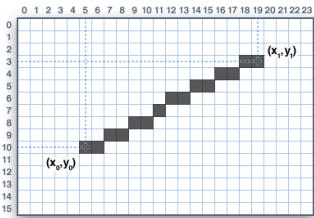
- **OledDisplay (--)**
transmet à l'afficheur OLED les commandes en attente d'affichage
- **OledHOME (--)**
Positionne le curseur en ligne 0, colonne 0 sur l'afficheur OLED. Cette position est au pixel près.
- **OledInvert (--)**
Inverse l'affichage de l'écran OLED
- **OledNum (n --)**
Affiche le nombre n sous forme de chaîne sur l'écran OLED
- **OledNumLn (n --)**
Affiche un nombre entier sur l'afficheur OLED et passe à la ligne suivante
- **OledPrint (z-string --)**
Affiche un texte z-string sur l'écran OLED
- **OledPrintln(z-string --)**
Affiche un texte z-string sur l'écran OLED et passe à la ligne suivante
- **OledTextc (WHITE|BLACK --)**
Définit la couleur du texte à afficher
- **OledSetCursor (x y --)**
Définit la position du curseur
- **OledTextsize (size --)**
Définit la taille du texte à afficher sur l'écran OLED. La valeur de n doit être dans l'intervalle [1..3]. Pour un texte de taille normale, size=1. Si vous dépassiez la valeur 4, le texte sera tronqué sur un afficheur 4 lignes.

Et voici un résumé des commandes de gestion d'affichage graphique :

- **OledCirc (x y radius color --)**
Trace un cercle centré à x y, de rayon radius et de couleur color (0|1)



- **OledCircF (x y radius color --)**
Trace un cercle plein centré à x y, de rayon radius et de couleur color (0|1)
- **OledDrawL (x0 y0 x1 y1 color --)**
Trace un trait depuis x0 y0 jusqu'à x1 y1 de couleur color.



- **OledFastHLine** (x y lenght color --)
Trace un trait horizontal depuis x y de dimension length et de couleur color.
- **OledFastVLine** (x y lenght color --)
Trace un trait vertical depuis x y de dimension length et de couleur color.
- **OledPixel** (x y color)
Active un pixel à la position x y. Le paramètre color détermine la couleur du pixel.
- **OledRect** (x y width height color --)
Trace un rectangle vide depuis la position x y de taille width height et de couleur color.
- **OledRectF** (x y width height color --)
Trace un rectangle plein depuis la position x y de taille width height et de couleur color.
- **OledRectR** (x y width height radius color --)
Trace un rectangle vide à coins arrondis, depuis la position x y, de dimension width heigh, dans la couleur color, avec un rayon radius.
- **OledRectRF** (x y width height radius color --)
Trace un rectangle plein à coins arrondis, depuis la position x y, de dimension width heigh, dans la couleur color, avec un rayon radius.

L'afficheur OLED permet d'exécuter les mots de gestion texte et graphiques dans le même mode d'affichage. En clair, on peut mélanger texte et graphismes.

Étendre le vocabulaire oled

Après quelques plantages, j'ai découvert qu'on ne peut pas définir une extension **OledTriangle** comme ceci en langage C pour étendre les définitions dans le fichier **oled.h** :

```
VV(oled, OledTriangle, oled_display->drawTriangle(n5, n4, n3, n2, n1, n0); DROFn(6)) \
```

mais c'est sans compter que nous programmons en langage FORTH et qu'avec ce langage on peut étendre notre vocabulaire oled. On va donc créer un nouveau fichier sur notre ordinateur, dans le répertoire de notre projet, avec le nom de fichier **extendOledVoc.fs**. Contenu :

```
RECORDFILE /spiffs/extendOledVoc.fs
oled definitions
: OledTriangle { x0 y0 x1 y1 x2 y2 color -- }
```

```

x0 y0 x1 y1 color OledDrawL
x1 y1 x2 y2 color OledDrawL
x2 y2 x0 y0 color OledDrawL
;
forth definitions
<EOF>

```

Puis on copie et colle ce code et on le transmet à ESP32forth via le terminal qui communique avec la carte ESP32. On doit se retrouver avec un fichier **extendOledVoc.fs** dans l'espace mémoire SPIFFS. On modifie maintenant le contenu du fichier **main.fs** :

```

s" /spiffs/config.fs"           included
s" /spiffs/extendOledVoc.fs"    included
s" /spiffs/oledTools.fs"        included

```

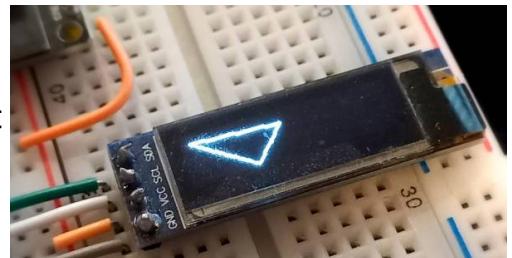
On débranche et rebranche la carte ESP32. On tape **MAIN** dans le terminal. Si tout s'est bien passé, on doit retrouver le mot **OledTriangle** en tapant simplement **oled vlist**.

Test de notre nouveau mot **OledTriangle** :

```

oled
OledCLS OledDisplay
5 5 60 8 40 30 WHITE OledTriangle OledDisplay

```



TEMPVS FVGIT⁸

Et si les romains avaient pu programmer l'affichage de l'heure sous forme numérique ?

Voilà un projet intéressant qui combine plusieurs fichiers. Dans ce chapitre, nous n'allons pas donner l'intégralité du code utilisé ici. Ce serait trop long.

Les codes sources de ce chapitre sont dans ce fichier :

- **ESP32forth-book.zip** → projects → tempusFugit

Lien : https://github.com/MPETREMANN11/ESP32forth/blob/main/_documentation/ESP32forth-book.zip

L'ensemble du projet contient ces fichiers :

- **autoexec.fs** contenu chargé au démarrage de ESP32forth
- **clepsydra.fs** conversion nombres en chiffres romains
- **config.fs** paramètres de configuration globale
- **main.fs** fichier principal chargeant les autres fichiers du projet
- **oledTools.fs** complète le vocabulaire **oled**
- **RTClock.fs** gère l'horloge temps réel
- **strings.fs** gère le traitement des chaînes alphanumériques

L'enchaînement du chargement des fichiers du projet est écrit dans **main.fs** :

```
s" /spiffs/strings.fs"           included
s" /spiffs/RTClock.fs"          included
s" /spiffs/clepsydra.fs"        included
s" /spiffs/config.fs"          included
s" /spiffs/oledTools.fs"        included
```

La plupart des fichiers de ce projet sont indépendants, à l'exception de **clepsydra.fs** qui est dépendant de **strings.fs**.

Romani non ustulo nulla⁹

Les romains ne connaissaient pas le chiffre 0. Alors comment peut-on afficher **13:00** ou **00:15** en chiffres romains ?

Pour résoudre le problème des heures après minuit, par exemple 00:15, les japonais (habitants du JAPON) vont nous

⁸ Tempus fugit = le temps passe

⁹ Romani non ustulo nulla = Les romains ne connaissaient pas le zéro



être d'une grande aide. Si un jour vous allez dans ce pays, vous serez étonné de voir des boutiques ouvertes jusqu'à **25:00**!

Cette boutique est ouverte de 09:00 à 25:00! Hé oui. Pourtant, les horloges du JAPON font aussi 24 heures. On savait les japonais travailleurs, mais au point de faire des journées de 25 heures, on a le droit d'avoir quelques doutes...

En fait, il y a une explication très logique. Après 12:00, il est 12:01, etc... Et donc, après 23:59, il est 24:00, puis 24:01. Donc, si une boutique ferme à 25:00, il faut comprendre qu'elle ferme à 01:00 pour nous.

Si on transpose ceci sur notre horloge romaine, quand il sera **00:00**, on pourra afficher **XXIV** ou mieux **XXIII:LX** (23:60).

Romani horas et minuta¹⁰

Pour résoudre le cas des heures comme 01:00, 02:00... 23:00, en toute logique, après 12:59, on peut très bien afficher 12:60, puis la minute d'après 13:01... 12:60 en chiffres romains: **XXII:LX**.

C'est ce qui est réalisé dans le mot **tempusTo\$** :

```
: tempusTo$ { HH MM -- }
    HH 0 = MM 0= AND if
        60 to MM
        23 to HH
    THEN
    HH 0 > MM 0= AND if
        60 to MM
        -1 +to HH
    then
    HH 0 <= if
        24 to HH
    then
    HH roman tempus $!
    [char] : tempus c+$!           \ add char :
    MM roman tempus append$
    tempus
;
```

Dans le premier test if..then, on teste si on est à **00:00**. Dans ce cas, on force l'heure à **23** et les minutes à **60**.

Dans le second test, si l'heure est supérieure à 00 et les minutes à 00, on décrémente l'heure et on force les minutes à **60**. L'inconvénient est que si l'heure est à 00, on la fait passer à -1.

Dans le dernier test, si l'heure est nulle ou négative, on la force à **24**.

10 Romani horas et minuta = Heures et minutes romaines

On peut utiliser le mot **.tempus** qui a servi à la mise au point pour vérifier ce bon fonctionnement :

```
--> 23 59 .tempus
XXIII:LIX ok
--> 0 0 .tempus
XXIII:LX ok
--> 0 1 .tempus
XXIV:I ok
--> 1 0 .tempus
XXIV:LX ok
--> 1 1 .tempus
I:I ok
```

Haec omnia integramus pro ESP32forth¹¹

En l'état du projet, on doit entrer manuellement l'heure initiale :

```
22 19 start
```

Signifie qu'on initialise l'heure à **22:19**. Cette initialisation s'effectue très simplement :

```
0 RTC.setTime
```

Ensuite on initialise l'afficheur OLED 128x32 :

```
Oled128x32Init
1 OledTextsize
WHITE OledTextc
```

Et pour finir, on va récupérer l'heure courante et l'afficher :

```
oledCLS OledDisplay
16 20 OledSetCursor
RTC.getTime drop tempusTo$ s>z OledPrintln OledDisplay
```

J'ai fait des tests avec un affichage plus grand. Le problème, pour la chaîne **XXIII:XXVIII**, il n'y a pas assez de place pour afficher cette chaîne.

Voici la boucle finale à exécuter pour lancer l'affichage de l'heure en chiffres romains :

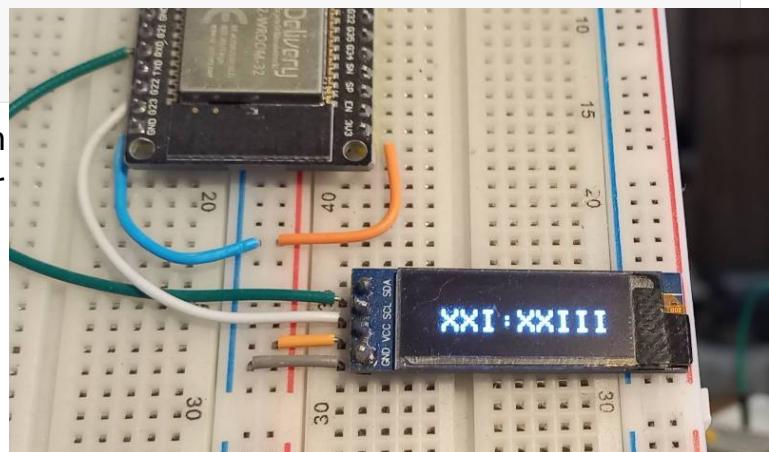
```
oled
: start ( HH MM -- )
  0 RTC.setTime      \ define current time
  Oled128x32Init
  1 OledTextsize
  WHITE OledTextc
begin
  oledCLS OledDisplay
  16 20 OledSetCursor
  RTC.getTime drop tempusTo$ s>z OledPrintln OledDisplay
  1000 ms
```

¹¹ Haec omnia integramus pro ESP32forth = On intègre tout ceci pour ESP32forth

```
key? until  
;  
forth
```

Le programme peut être amélioré en récupérant l'heure depuis un serveur de temps. Voir chapitre *Récupérer l'heure depuis un serveur WEB*.

Il est également possible d'utiliser les fonctions de **timers** pour libérer l'interpréteur. Voir chapitre *Clignotement d'une LED par timer*.



Pour terminer, l'assemblage des différents fichiers de ce projet et les quelques tests et aménagements m'ont occupé une après-midi.

Je tiens cependant à insister sur quelques points :

- faites toujours une copie dans le dossier de votre projet d'un fichier à usage général. Par exemple, pour le fichier **strings.fs** ;
- si vous copiez un composant général, par exemple **strings.fs** ou **RTClock.fs**, ne faites des modifications que sur ces fichiers copiés dans le dossier de votre projet. Versionnez ces modifications et indiquez la date de modification dans le commentaire de tête du fichier modifié.

Au fil de vos projets, il est possible que vous vous retrouviez avec un même fichier copié dans différents dossiers et modifié. Cette solution est préférable à celle d'un fichier unique rempli de paramètres d'ajustement.

Ajouter la librairie SPI

La librairie SPI n'est pas nativement implémentée dans ESP32forth. Pour l'installer, vous devez préalablement créer le fichier **spi.h** qui devra être installé dans le même dossier que celui contenant le fichier **ESP32forth.ino**.

Contenu du fichier **spi.h** (en langage C) :

```
# include <SPI.h>

#define OPTIONAL_SPI_VOCABULARY V(spi)
#define OPTIONAL_SPI_SUPPORT \
    XV(internals, "spi-source", SPI_SOURCE, \
        PUSH spi_source; PUSH sizeof(spi_source) - 1) \
    XV(spi, "SPI.begin", SPI_BEGIN, SPI.begin((int8_t) n3, (int8_t) n2, (int8_t) n1, (int8_t) n0); DROFn(4)) \
    XV(spi, "SPI.end", SPI_END, SPI.end()); \
    XV(spi, "SPI.setHwCs", SPI_SETHWCS, SPI.setHwCs((boolean) n0); DROP) \
    XV(spi, "SPI.setBitOrder", SPI_SETBITORDER, SPI.setBitOrder((uint8_t) n0); DROP) \
    XV(spi, "SPI.setDataMode", SPI_SETDATAMODE, SPI.setDataMode((uint8_t) n0); DROP) \
    XV(spi, "SPI.setFrequency", SPI_SETFREQUENCY, SPI.setFrequency((uint32_t) n0); DROP) \
    XV(spi, "SPI.setClockDivider", SPI_SETCLOCKDIVIDER, SPI.setClockDivider((uint32_t) n0); DROP) \
    XV(spi, "SPI.getClockDivider", SPI_GETCLOCKDIVIDER, PUSH SPI.getClockdivider()); \
    XV(spi, "SPI.transfer", SPI_TRANSFER, SPI.transfer((uint8_t *) n1, (uint32_t) n0); DROFn(2)) \
    XV(spi, "SPI.transfer8", SPI_TRANSFER_8, PUSH (uint8_t) SPI.transfer((uint8_t) n0); NIP) \
    XV(spi, "SPI.transfer16", SPI_TRANSFER_16, PUSH (uint16_t) SPI.transfer16((uint16_t) n0); NIP) \
    XV(spi, "SPI.transfer32", SPI_TRANSFER_32, PUSH (uint32_t) SPI.transfer32((uint32_t) n0); NIP) \
    XV(spi, "SPI.transferBytes", SPI_TRANSFER_BYTES, SPI.transferBytes((const uint8_t *) n2, (uint8_t *) n1, (uint32_t) n0); \
DROFn(3)) \
    XV(spi, "SPI.transferBits", SPI_TRANSFER_BITES, SPI.transferBits((uint32_t) n2, (uint32_t *) n1, (uint8_t) n0); DROFn(3)) \
    XV(spi, "SPI.write", SPI_WRITE, SPI.write((uint8_t) n0); DROP) \
    XV(spi, "SPI.write16", SPI_WRITE16, SPI.write16((uint16_t) n0); DROP) \
    XV(spi, "SPI.write32", SPI_WRITE32, SPI.write32((uint32_t) n0); DROP) \
    XV(spi, "SPI.writeBytes", SPI_WRITE_BYTES, SPI.writeBytes((const uint8_t *) n1, (uint32_t) n0); DROFn(2)) \
    XV(spi, "SPI.writePixels", SPI_WRITE_PIXELS, SPI.writePixels((const void *) n1, (uint32_t) n0); DROFn(2)) \
    XV(spi, "SPI.writePattern", SPI_WRITE_PATTERN, SPI.writePattern((const uint8_t *) n2, (uint8_t) n1, (uint32_t) n0); \
DROFn(3))

const char spi_source[] = R"""" \
vocabulary spi  spi definitions \
transfer spi-builtins \
forth definitions \
) """;
```

Le fichier complet est également disponible ici :

<https://github.com/MPETREMANN11/ESP32forth/blob/main/optional/spi.h>

Modifications du fichier **ESP32forth.ino**

Le contenu du fichier **spi.h** ne peut pas être intégré à ESP32forth sans apporter quelques modifications au fichier **ESP32forth.ino**. Voici les quelques modifications à apporter à ce fichier. Ces modifications ont été réalisées sur la version 7.0.7.15, mais devraient être applicables aux autres versions récentes ou à venir.

Première modification

Code rajouté en rouge :

```
#define VOCABULARY_LIST \
    V(forth) V(internals) \
    V(rtos) V(SPIFFS) V(serial) V(SD) V(SD_MMC) V(ESP) \
    V(ledc) V(Wire) V(WiFi) V(sockets) \
    OPTIONAL_CAMERA_VOCABULARY \
```

```

OPTIONAL_BLUETOOTH_VOCABULARY \
OPTIONAL_INTERRUPTS_VOCABULARIES \
OPTIONAL_OLED_VOCABULARY \
OPTIONAL_SPI_VOCABULARY \
OPTIONAL_RMT_VOCABULARY \
OPTIONAL_SPI_FLASH_VOCABULARY \
USER_VOCABULARIES

```

Seconde modification

Rajout en rouge après ce code :

```

// Hook to pull in optional Oled support.
# if __has_include("oled.h")
# include "oled.h"
# else
# define OPTIONAL_OLED_VOCABULARY
# define OPTIONAL_OLED_SUPPORT
# endif

// Hook to pull in optional SPI support.
# if __has_include("spi.h")
# include "spi.h"
# else
# define OPTIONAL_SPI_VOCABULARY
# define OPTIONAL_SPI_SUPPORT
# endif

```

Troisième modification

Rajout en rouge :

```

#define EXTERNAL_OPTIONAL_MODULE_SUPPORT \
OPTIONAL_ASSEMBLERS_SUPPORT \
OPTIONAL_CAMERA_SUPPORT \
OPTIONAL_INTERRUPTS_SUPPORT \
OPTIONAL_OLED_SUPPORT \
OPTIONAL_SPI_SUPPORT \
OPTIONAL_RMT_SUPPORT \
OPTIONAL_SERIAL_BLUETOOTH_SUPPORT \
OPTIONAL_SPI_FLASH_SUPPORT

```

Quatrième modification

Rajout en rouge :

```

internals DEFINED? oled-source [IF]
  oled-source evaluate
[THEN] forth

internals DEFINED? spi-source [IF]
  spi-source evaluate
[THEN] forth

```

Si vous suivez bien ces consignes, vous pourrez compiler ESP32forth avec ARDUINO IDE et le téléverser sur la carte ESP32. Une fois ces opérations effectuées, lancez le terminal. Vous devez retrouver le prompt d'accueil ESP32forth. Tapez :

```
spi vlist
```

Vous devez retrouver les mots définis dans ce vocabulaire **spi** :

```
SPI.begin SPI.end SPI.setHwCs SPI.setBitOrder SPI.setDataMode SPI.setFrequency  
SPI.setClockDivider SPI.getClockDivider SPI.transfer SPI.transfer8 SPI.transfer16  
SPI.transfer32 SPI.transferBytes SPI.transferBits SPI.write SPI.write16  
SPI.write32 SPI.writeBytes SPI.writePixels SPI.writePattern spi-builtins
```

Vous pouvez maintenant piloter des extensions via le port SPI, comme les afficheurs LED MAX7219.

Communiquer avec le module d'affichage MAX7219

Dans une communication SPI, il y a toujours un *maître* qui contrôle les périphériques (également appelés *esclaves*). Les données peuvent être envoyées et reçues simultanément. Cela signifie que le maître peut envoyer des données à un esclave et un esclave peut envoyer des données au maître en même temps.

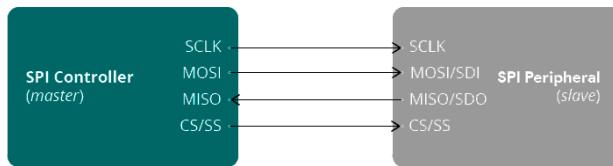


Figure 20: commande d'un périphérique SPI

Vous pouvez avoir plusieurs esclaves. Un esclave peut être un capteur, un écran, une carte microSD, etc., ou un autre microcontrôleur. Cela signifie que vous pouvez avoir votre ESP32 connecté à **plusieurs périphériques**.

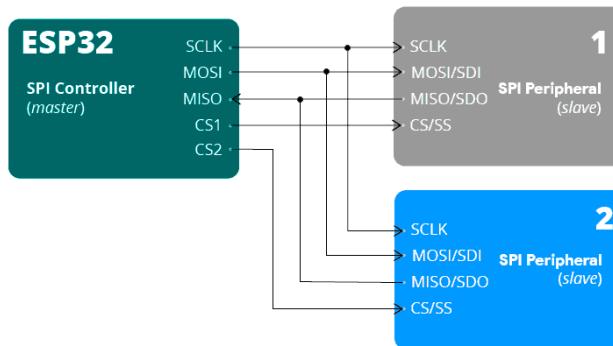


Figure 21: commande de deux périphériques SPI

La sélection d'un esclave s'effectue en mettant au niveau bas le sélecteur CS1 ou CS2. Il faudra autant de sélecteurs CS qu'il y a d'esclaves à gérer.

Répérage du port SPI sur la carte ESP32

Il y a deux ports SPI sur une carte ESP32: HSPI et VSPI. Le port SPI que nous allons gérer est celui dont les pins sont préfixés VSPI:

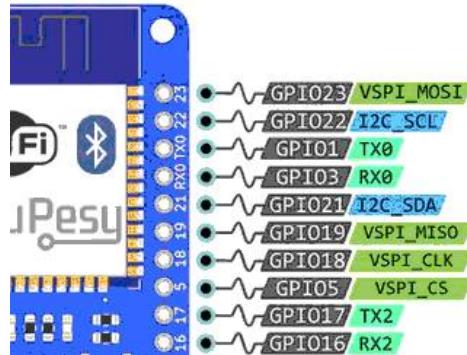


Figure 22: les deux ports SPI sur la carte ESP32

Avec ESP32forth, on peut donc définir les constantes pointant vers ces pins VSPI :

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

Pour communiquer vers le module d'affichage MAX7219, nous n'aurons besoin de câbler que les pins VSPI_MOSI, VSPI_SCLK et VSPI_CS.

Les connecteurs SPI sur le module d'affichage MAX7219

Voici la carte des connecteurs du port SPI sur le module MAX7219 :

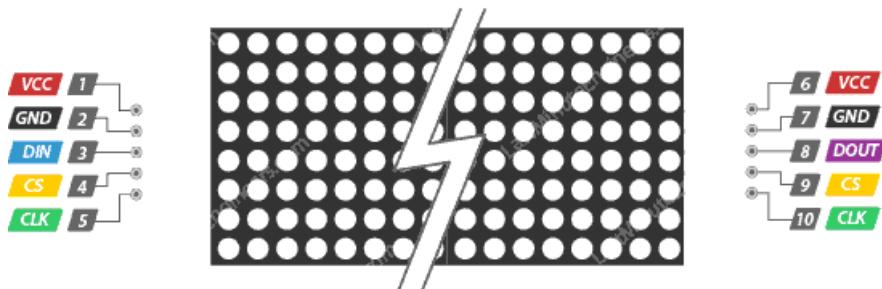


Figure 23: connecteurs SPI sur le module MAX7219

Connexion entre le module MAX7219 et la carte ESP32 :

MAX7219		ESP32
DIN	<---->	VSPI_MOSI
CS	<---->	VSPI_CS
CLK	<---->	VSPI_SCLK

Les connecteurs VCC et GND sont raccordés à une alimentation extérieure :

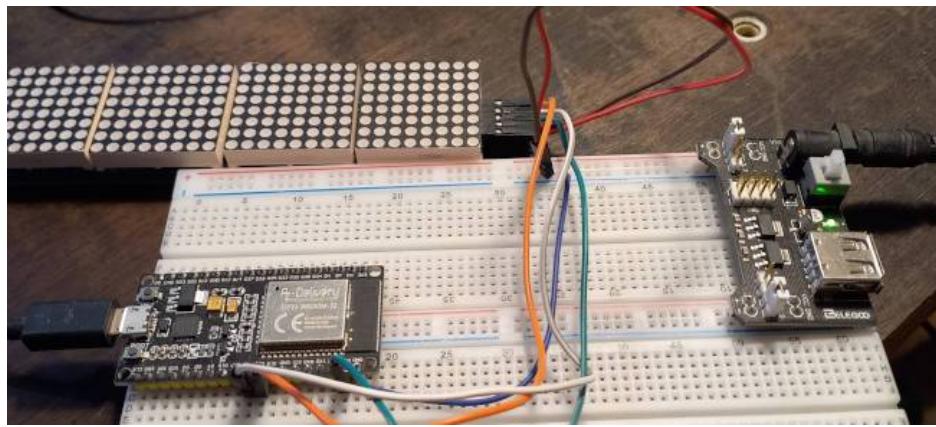


Figure 24: utilisation d'une alimentation extérieure

La partie GND de cette alimentation extérieure est mise en commun avec le pin GND de la carte ESP32.

Couche logicielle du port SPI

Tous les mots permettant de gérer le port SPI sont déjà disponibles dans le vocabulaire **spi**.

La seule chose à définir concerne l'initialisation du port SPI :

```
\ define SPI port frequency
4000000 constant SPI_FREQ

\ select SPI vocabulary
only FORTH SPI also

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

Nous sommes maintenant prêts à utiliser notre module d'affichage MAX7219.

Installation du client HTTP

Modification du fichier ESP32forth.ino

ESP32Forth est fourni sous forme de fichier source, écrit en langage C. Ce fichier doit être compilé à l'aide de ARDUINO IDE ou tout autre compilateur C compatible avec l'environnement de développement ARDUINO.

Voici les portions de code à modifier. Première portion à modifier :

```
#define ENABLE_SD_SUPPORT
#define ENABLE_SPI_FLASH_SUPPORT
#define ENABLE_HTTP_SUPPORT
// #define ENABLE_HTTPS_SUPPORT
```

Seconde portion à modifier :

```
// .....
#define VOCABULARY_LIST \
  V(forth) V(internal) \
  V(rtos) V(SPIFFS) V(serial) V(SD) V(SD_MMC) V(ESP) \
  V(ledc) V(http) V(Wire) V(WiFi) V(bluetooth) V(sockets) V(oled) \
  V(rmt) V(interrupts) V(spi_flash) V(camera) V(timers)
```

Troisième portion à modifier :

```
OPTIONAL_RMT_SUPPORT \
OPTIONAL_OLED_SUPPORT \
OPTIONAL_SPI_FLASH_SUPPORT \
OPTIONAL_HTTP_SUPPORT \
FLOATING_POINT_LIST

#ifndef ENABLE_HTTP_SUPPORT
# define OPTIONAL_HTTP_SUPPORT
#else

# include <HTTPClient.h>
HTTPClient http;

# define OPTIONAL_HTTP_SUPPORT \
  XV(http, "HTTP.begin", HTTP_BEGIN, tos = http.begin(c0)) \
  XV(http, "HTTP.doGet", HTTP_DOGET, PUSH http.GET()) \
  XV(http, "HTTP.getPayload", HTTP_GETPL, String s = http.getString(); \
      memcpy((void *) n1, (void *) s.c_str(), n0); DROPN(2)) \
  XV(http, "HTTP.end", HTTP_END, http.end())
#endif
```

Quatrième portion à modifier :

```
vocabulary ledc ledc definitions
transfer ledc-builtins
```

```

forth definitions

vocabulary http  http definitions
transfer http-builtins
forth definitions

vocabulary Serial    Serial definitions
transfer Serial-builtins
forth definitions

```

Une fois le fichier **ESP32forth.ino** modifié, vous le compilez et le téléversez sur la carte ESP32. Si tout s'est correctement déroulé, vous devez disposer d'un nouveau vocabulaire **http**:

```

http
vlist  \ displays :
HTTP.begin HTTP.doGet HTTP.getPayload HTTP.end http-builtins

```

Test du client HTTP

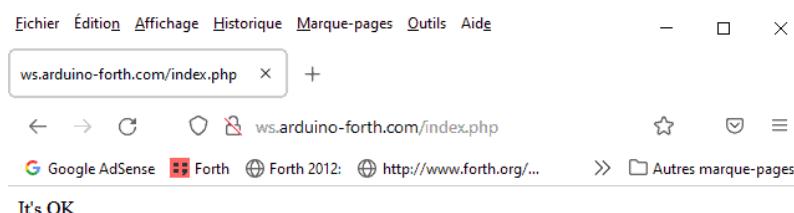
Pour tester notre client HTTP, on peut le faire en interrogeant n'importe quel serveur web. Mais pour ce que nous envisageons ultérieurement, il faut disposer d'un serveur web personnel. Sur ce serveur, on crée un sous domaine :

- notre serveur est arduino-forth.com
- on crée un sous-domaine **ws**
- on accède à ce sous domaine avec l'URL <http://ws.arduino-forth.com>

Ce sous-domaine étant créé, il ne contient aucun script à exécuter. On crée la page **index.php** et on y met ce code :

```
It's OK
```

Pour vérifier que notre sous-domaine est fonctionnel, il suffit de l'interroger depuis notre navigateur web préféré :



Si tout se passe comme prévu, on doit avoir le texte **It's OK** qui s'affiche dans notre navigateur web préféré. Voyons maintenant comment réaliser cette même interrogation de serveur depuis ESP32Forth...

Voici le code FORTH écrit rapidement pour effectuer le test du client HTTP :

```
WiFi

\ connection to local WiFi LAN
: myWiFiConnect
    z" mySSID"
    z" myWiFiCode"
    login
;

Forth

create httpBuffer 700 allot
    httpBuffer 700 erase

HTTP

: run
    cr
    z" http://ws.arduino-forth.com/" HTTP.begin
    if
        HTTP.doGet dup ." Get results: " . cr 0 >
        if
            httpBuffer 700 HTTP.getPayload
            httpBuffer z>s dup . cr type
        then
    then
    HTTP.end
;
```

On active la liaison Wifi en exécutant **myWiFiConnect** puis **run**:

```
--> myWiFiConnect
192.168.1.23
MDNS started
ok
--> run

Get results: 200
8
It's OK
ok
```

Notre client HTTP a parfaitement interrogé le serveur web en affichant le même texte que celui récupéré à partir de notre navigateur web.

Ce petit test réussi ouvre la voie à d'énormes possibilités.

Récupérer l'heure depuis un serveur WEB

Dans le chapitre *Horloge temps réel logicielle*, on s'est intéressé à la manière de gérer une horloge temps réel grâce aux propriétés du **timer**.

Cependant, l'initialisation de cette horloge temps réel doit s'effectuer manuellement. Maintenant que nous disposons d'un moyen de communiquer avec un serveur web, nous allons voir comment effectuer cette initialisation au travers d'un serveur web.

Transmission et réception du temps depuis un serveur web

Pour la partie serveur, on crée un nouveau script **gettime.php** dont voici le contenu :

```
<?php  
echo date('H i s')." RTC.set-time";
```

Si on exécute ce script <http://ws.arduino-forth.com/gettime.php>, sur un navigateur web, voici ce qui s'affiche :

```
15 25 30 RTC.set-time
```

On a préparé le travail pour que l'interpréteur ESP32Forth n'aie que cette ligne à exécuter. Voici le code FORTH permettant de récupérer l'heure :

```
WiFi  
\ connection to local WiFi LAN  
: myWiFiConnect  
  z" mySSID"  
  z" myWiFiCode"  
  login  
;  
  
Forth  
  
0 value currentTime  
  
\ store current time  
: RTC.set-time { hh mm ss -- }  
  hh 3600 *  
  mm 60 *  
  ss + + 1000 *  
  MS-TICKS - to currentTime  
;  
  
\ 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
;

700 constant bufferSize
create httpBuffer
    bufferSize allot

0 buffer 700 erase

HTTP

: getTime
    cr
    z" http://ws.arduino-forth.com/gettime.php" HTTP.begin
    if
        HTTP.doGet
        if
            httpBuffer bufferSize HTTP.getPayload
            httpBuffer z>s evaluate
        then
    then
    HTTP.end
;

myWiFiConnect
getTime
RTC.display-time

```

Dans le mot **getTime**, cette séquence **httpBuffer z>s evaluate** récupère le contenu du buffer de transaction web et évalue son contenu. Ceci est possible, car le serveur web a transmis une séquence compatible avec notre interpréteur FORTH. L'exécution des trois dernières lignes de ce code affiche ceci :

```

--> myWiFiConnect
192.168.1.23
MDNS started
ok
--> getTime
ok
--> RTC.display-time
15:33:09 ok

```

Cette initialisation peut être exécutée une seule fois, en général au démarrage de ESP32Forth. Cette technique consistant à interroger notre propre serveur web évite de négocier avec un serveur de temps.

La plupart des serveurs de temps délivrent l'information dans des formats difficiles à traiter par FORTH: csv, JSON, XML...

Comprendre la transmission par GET vers un serveur WEB

Transmission de données vers un serveur par GET

Il existe deux méthodes de transmission de données depuis une page web vers un serveur web :

POST qui est la méthode généralement utilisée depuis les formulaires

GET qui est la méthode que nous allons étudier

Il existe d'autres méthodes, mais celles-ci sont généralement réservées aux transactions entre machines au travers de services web.

Les paramètres dans un URL

Commençons par expliquer ce qu'est un URL : <http://my-website.com/> (URL pour l'exemple).

On analyse un URL en commençant par la fin :

- **.com** est le TLD (Top-Level Domain)
- **my-website** est le nom de domaine
- **http://** est le protocole de communication.

Nous n'allons pas faire un cours exhaustif sur ces éléments. La seule chose qu'il y a à savoir vient maintenant.

Cet URL peut être suivi par le script ou la page HTML, exemple :

<http://my-website.com/index.php>

On peut compléter cet URL avec un passage de paramètre :

`http://my-website.com/index.php?temp=32.7`

Ici on passe un paramètre **temp** dont la valeur est **32.7**.

Le passage de paramètres par la méthode GET est marqué par le signe **?**.

Passage de plusieurs paramètres

On peut transmettre plusieurs paramètres en les séparant par le signe **&**:

`http://my-website.com/index.php?log=myLog&pwd=myPasswd&temp=32.7`

Ici, nous transmettons trois paramètres :

log avec la valeur myLog

pwd avec la valeur myPassWd

temp avec la valeur 32.7

Pour comprendre comment le serveur va recevoir ces données, on crée un script **record.php** qui va provisoirement contenir simplement ceci:

```
<?php  
var_dump($_GET);
```

et qui affichera ceci si on interroge ce script avec notre navigateur web préféré :

```
array(3) {  
    ["log"]=>  
    string(7) "myLogin"  
    ["pwd"]=>  
    string(10) "mypassword"  
    ["temp"]=>  
    string(4) "32.7"  
}
```

C'est quasiment tout ce dont nous avons besoin pour récupérer les données et les enregistrer sur le serveur. C'est ce que nous allons découvrir...

Gestion du passage de paramètres avec ESP32forth

Pour commencer, il est nécessaire de disposer de mots permettant de gérer des chaînes de caractères. Vous trouverez ces mots dans le chapitre *Affichage des nombres et chaînes de caractères*, partie *Code des mots de gestion de variables texte*.

On commence par créer une chaîne de caractères :

```
256 string myUrl  
s" http://ws.arduino-forth.com/record.php?log=myLog&pwd=myPassWd&temp="  
myUrl $!
```

On vient de définir une variable alphanumérique **myUrl**. Cette variable est quasiment complète. Il ne manque que la valeur du paramètre **temp**. Pour rajouter cette valeur, on exécutera **append\$**:

```
s" 32.5" myUrl append$  
myUrl type  
\ display: http://ws.arduino-forth.com/record.php?  
log=myLog&pwd=myPassWd&temp=32.5
```

C'est cet URL que nous allons utiliser dans cette définition :

```
: sendData ( str -- )  
  s" http://ws.arduino-forth.com/record.php?log=myLog&pwd=myPassWd&temp="  
  myUrl $!  
  myUrl append$  
  \ cr myUrl type
```

```

myUrl s>z HTTP.begin
if
    HTTP.doGet dup 200 =
    if drop
        httpBuffer bufferSize HTTP.getPayload
        httpBuffer z>s type
    else
        cr ." CNX ERR: " .
    then
then
HTTP.end
;

myWiFiConnect
s" 32.65" sendData

```

Le mot **sendData** récupère le contenu de la chaîne, ici **32.65**, concatène ce contenu à **myUrl**, puis engage une transaction client Web vers le serveur mentionné dans **myUrl**.

Vous noterez que dans l'URL il y a un paramètre log. Ce paramètre peut être différent pour chaque carte ESP32 engageant une transaction vers le serveur web. Il est possible à dix, vingt, ou même mille cartes ESP32 d'enregistrer leurs données vers un seul serveur web.

Transmission de données vers un serveur WEB

Enregistrement des données côté serveur web

Dans le précédent chapitre *Comprendre la transmission par GET vers un serveur WEB*, on a expliqué comment ESP32Forth la transmission d'information vers un serveur web.

Voyons maintenant comment, coté serveur, on va enregistrer les données. Voici un premier script, en PHP, qui effectue cet enregistrement :

```
<?php
// echo "<pre>"; var_dump($_GET);
$handle = fopen("datasRecords.csv", "a");
$myDatas = array(
    'currentDateTime' => date("Y-m-d H:i:s"),
    'currentLogin'     => $_GET['log'],
    'currentTemp'      => $_GET['temp'],
);
fwrite($handle, implode(';', $myDatas) . "
");
fclose($handle);
echo "DATAs recorded";
```

Ce script est très simple :

- on ouvre un fichier **dataRecords.csv** avec **fopen**.
- on prépare les données à enregistrer dans un tableau **myDatas**
- on enregistre ces données avec **fwrite**
- les données sont mises au format csv grâce à **implode**
- on ferme le fichier avec **fclose**

Le fichier au format **csv** est facile à récupérer avec un tableur ou à lire avec un simple éditeur de texte.

Protection de l'accès

Si vous avez bien suivi nos explications, vous avez noté qu'on transmet deux paramètres **log** et **pwd**. Ces deux paramètres servent d'abord de clés d'accès à notre script d'enregistrement des données.

C'est cette protection que nous mettons en place pour empêcher tout accès au script à un transmetteur non autorisé. Ici, on accepte deux transmetteurs :

```

<?php
// echo "<pre>"; var_dump($_GET);
$myAuths = array(
    'pooltemp' => 'pool2022',
    'housetemp' => 'house2022',
);

/**
 * Test authorization access
 * @param array $auths
 * @return boolean
 */
function testAuths($auths) {
    if(array_key_exists($_GET['log'], $auths) &&
$auths[$_GET['log']]==$_GET['pwd']) {
        return true;
    }
    return false;
}

// Recording datas in CSV file format
if (testAuths($myAuths)) {
    $handle = fopen("datasRecords.csv", "a");
    $myDatas = array(
        'currentDateTime' => date("Y-m-d H:i:s"),
        'currentLogin'     => $_GET['log'],
        'currentTemp'      => $_GET['temp'],
    );
    fwrite($handle, implode(';', $myDatas) . "
");
    fclose($handle);
    echo "DATAs recorded";
} else {
    echo "AUTH failed";
}

```

Ce script sert d'exemple. Il est volontairement simple. Sur une application professionnelle, les clés et mots de passe seraient enregistrés en base de données.

Ici une transaction qui sera exécutée avec succès :

```
http://ws.arduino-fourth.com/record.php?log=pooltemp&pwd=pool2022&temp=27.5
```

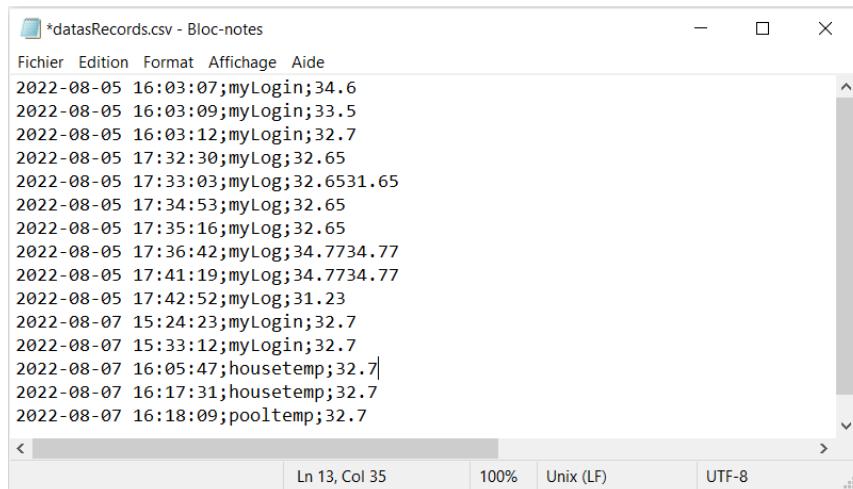
Cette transaction transmet un couple **log pwd** dont les valeurs sont testées et approuvées par le script d'enregistrement de données.

Consulter les données enregistrées

Pour accéder aux données enregistrées, on utilise un client FTP (Filezilla) :

Nom de fichier	Taille d...	Type de ...	Dernière modification
..			
datasRecords.csv	672	Fichier C...	07/08/2022 16:18:09
record.php	927	Fichier P...	07/08/2022 16:17:22
gettime.php	41	Fichier P...	04/08/2022 15:25:24
index.php	8	Fichier P...	03/08/2022 20:14:10

On y retrouve notre fichier **datasRecords.csv**. Il suffit de le télécharger pour en voir le contenu avec n'importe quel éditeur de texte :



```

*datasRecords.csv - Bloc-notes
Fichier Edition Format Affichage Aide
2022-08-05 16:03:07;myLogin;34.6
2022-08-05 16:03:09;myLogin;33.5
2022-08-05 16:03:12;myLogin;32.7
2022-08-05 17:32:30;myLog;32.65
2022-08-05 17:33:03;myLog;32.6531.65
2022-08-05 17:34:53;myLog;32.65
2022-08-05 17:35:16;myLog;32.65
2022-08-05 17:36:42;myLog;34.7734.77
2022-08-05 17:41:19;myLog;34.7734.77
2022-08-05 17:42:52;myLog;31.23
2022-08-07 15:24:23;myLogin;32.7
2022-08-07 15:33:12;myLogin;32.7
2022-08-07 16:05:47;housetemp;32.7
2022-08-07 16:17:31;housetemp;32.7
2022-08-07 16:18:09;pooltemp;32.7

```

On retrouve, dans les dernières lignes, nos tests de transmission avec deux login différents. Le script **record.php** peut traiter des transactions avec des centaines de cartes ESP32 différentes, chacune avec un login différent.

Rajouter des données à transmettre

Si vous gérer un capteur de type DHT11 ou DHT22 (capteur de température et humidité), vous seriez tenté d'enregistrer les valeurs de température et d'hygrométrie en une seule transaction. Pour ce faire, rien de plus facile. Voici l'aspect de la transaction permettant ceci :

```
http://ws.arduino-forth.com/record.php?log=pol1temp&pwd=pool2022&temp=27.5&hygr=62.2
```

Mais pour que ça fonctionne, il faut agir sur le script PHP record.php :

```

<?php
// Recording datas in CSV file format
if (testAuths($myAuths)) {
    $handle = fopen("datasRecords.csv", "a");
    $myDatas = array(
        'currentDateTime' => date("Y-m-d H:i:s"),
        'currentLogin'     => $_GET['log'],
        'currentTemp'      => $_GET['temp'],
        'currentHygr'      => $_GET['hygr'],
    );
    fwrite($handle, implode(';', $myDatas) . "

```

```

");
fclose($handle);
echo "DATAAs recorded";
} else {
    echo "AUTH failed";
}

```

Ici, on rajoute simplement une ligne dans la table **\$myDatas**.

Coté FORTH, on va améliorer la gestion de l'URL :

```

256 string myUrl      \ declare string variable

: addTemp ( strAddrLen -- )
    s" &temp=" myUrl append$
    myUrl append$
;

: addHygr ( strAddrLen -- )
    s" &hygr=" myUrl append$
    myUrl append$
;

: sendData ( strHygr strTemp -- )
    s" http://ws.arduino-forth.com/record.php?log=myLog&pwd=myPassWd" myUrl
$!
    addTemp
    addHygr
    cr myUrl type
    myUrl s>z HTTP.begin
    if
        HTTP.doGet dup 200 =
        if drop
            httpBuffer bufferSize HTTP.getPayload
            httpBuffer z>s type
        else
            cr ." CNX ERR: " .
        then
    then
    HTTP.end
;

\ for test:
myWiFiConnect
s" 64.2"   \ hygrometry
s" 31.23"  \ temperature
sendData

```

On a rajouté deux mots, **addTemp** et **addHygr**. Chacun de ces mots concatène un paramètre et sa valeur à l'URL qui sera utilisé pour la transaction web entre votre carte ESP32 et le serveur web.

Il n'y a que deux limitations au nombre de paramètres transmissibles par la méthode GET :

- la longueur de notre URL tel que défini en FORTH, ici 256 caractères. Si vous souhaitez augmenter cette limite, il suffit de définir notre URL avec une longueur initiale plus longue: **512 string myUrl**
- la longueur maximale des URLs acceptés par le protocole HTTP. Cette longueur peut atteindre 8000 caractères selon les normes récentes.

Pour ce qui concerne FORTH, on a d'autres limitations. En particulier, si on souhaite transmettre des données textuelles, certains caractères, "&" par exemple, devront être encodés. Vous devrez gérer cet encodage en FORTH.

Conclusion

QUESTION: à quoi ça peut servir tout ça?

Une carte ESP32 coûte moins de 10€/\$ pièce. Même plutôt 5€/\$ si vous en achetez en quantité. Si vous intégrez un capteur de tméparture et un relais, vous pouvez par exemple effectuer des relevés de température et transmettre des commandes depuis le serveur pour activer/désactiver un relais. Gérer la température de plusieurs pièces devient très facile. Idem pour gérer un arrosage intelligent dans une serre.

Vous pouvez également surveiller des accès et déclencher des éclairages ou alarmes très facilement. Prenons le cas d'un portail. Vous autorisez les passages entre certaines heures et vous verrouillez ce même portail (ventouse magnétique) depuis la carte ESP32.

Nous faisons confiance à votre imagination pour trouver des montages pratiques exploitant cette transmission de données entre des cartes ESP32 et un serveur web.

ET POURQUOI UN SERVEUR WEB?

Avec'un serveur web, il est facile de l'interroger depuis n'importe où, avec un navigateur web installé sur votre PC, une tablette numérique, un smartphone. Et un seul serveur web peut intégrer un nombre indéterminé de scripts différents.

Synthèse sonore avec ESP32Forth

Pour vos premières expérimentations sonores, il faut disposer d'un haut-parleur que vous connectez à une sortie GPIO. Mais l'impédance des haut-parleurs étant très basse, il faudra passer par un transistor. Voici le schéma conseillé pour un petit haut-parleur.

Sur ce schéma, il est mentionné le pin GPIO4. En fait, ce montage est utilisable sur n'importe quelle sortie GPIO de la carte ESP32. Les deux sorties qui vont plus particulièrement nous intéresser sont GPIO25 et GPIO26 qui sont réservées aux sorties DAC (Digital to Analog Conversion).

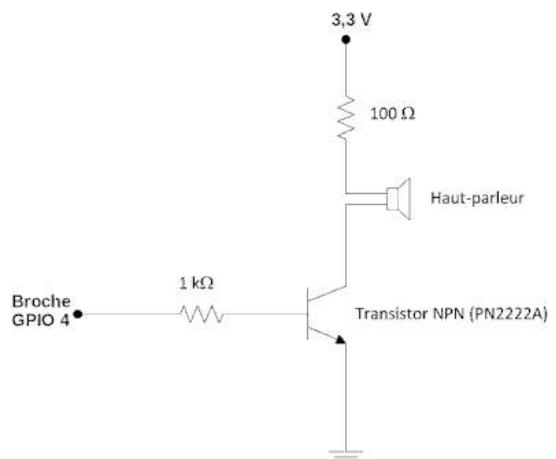


Figure 25: branchement d'un haut parleur

Synthèse sonore simple

Nous allons utiliser la génération de signaux PWM, mais sur les sorties DAC.

Notre haut-parleur est connecté à la sortie GPIO25, via le transistor PN2222A qui sert d'adaptateur d'impédance.

```
0 constant CHANNEL0      \ define PWM channel 0
25 constant BUZZER        \ buzzer connected to GPIO25

ledc                      \ select ledc vocabulary
: initTones ( -- )
    BUZZER CHANNEL0 ledcAttachPin
;
```

Le mot **initTones** connecte la sortie GPIO25 au canal PWM 0. La génération d'un son s'effectue comme ceci :

```
CHANNEL0 freq ledcWriteTone drop
```

où freq est la fréquence souhaitée, multipliée par 1000. Ainsi, pour générer la note LA (A en notation anglaise), dont la fréquence est de 440 Hz, il faudra utiliser la valeur $440 * 1000$:

```
CHANNEL0 440000 ledcWriteTone drop
```

Définition du tableau des fréquences sonores

Pour retrouver les fréquences sonores des notes de musique, nous sommes allés sur Wikipedia. On construit un tableau des fréquences, où chaque fréquence sera enregistrée sous sa forme utilisable par **ledcWriteTone**:

```

\ frequency notes
\ source: https://fr.wikipedia.org/wiki>Note_de_musique
\ frequency is multiplied by 1000
create NOTES
\ octave -1
15350 , 17330 , 18360 , 19450 , 20600 , 21830 ,
23130 , 24500 , 25960 , 27500 , 29140 , 30870 ,
\ octave 0
32700 , 34650 , 36710 , 38890 , 41200 , 43650 ,
46250 , 49000 , 51910 , 55000 , 58270 , 61740 ,
\ octave 1
65410 , 69300 , 73420 , 77780 , 82410 , 87310 ,
92500 , 98000 , 103830 , 110000 , 116540 , 123470 ,
\ octave 2
130810 , 138590 , 146830 , 155560 , 164810 , 174610 ,
185000 , 196000 , 207650 , 220000 , 233080 , 246940 ,
\ octave 3
261630 , 277180 , 293660 , 311130 , 329630 , 349230 ,
369990 , 392000 , 415300 , 440000 , 466160 , 493880 ,
\ octave 4
523250 , 554370 , 587330 , 622250 , 659260 , 698460 ,
739990 , 783990 , 830610 , 880000 , 932330 , 987770 ,
\ octave 5
1046500 , 1108730 , 1174660 , 1244510 , 1318510 , 1396910 ,
1479980 , 1567980 , 1661220 , 1760000 , 1864660 , 1975530 ,
\ octave 6
2093000 , 2217460 , 2349320 , 2489020 , 2637020 , 2793830 ,
2959960 , 3135960 , 3322440 , 3520000 , 3729310 , 3951070 ,
\ octave 7
4186010 , 4434920 , 4698640 , 4978030 , 5274040 , 5587650 ,
5919910 , 6271930 , 6644880 , 7040000 , 7458620 , 7902130 ,
\ octave 8
8372020 , 8869840 , 9397280 , 9956060 , 10548080 , 11175300 ,
11839820 , 12543860 , 13289760 , 14080000 , 14917240 , 15804260 ,

```

Il y a douze notes par octave, d'où la définition de 12 valeurs par octave. Ici, on enregistre seulement 10 lignes, soit 10 octaves. Car passé 15Khz, les sons ne seraient plus audibles.

Pour retrouver une note, il suffit de connaître sa position dans une octave. Par exemple, notre note LA en octave 3 sera: ((octave+1)*12)+position. LA étant en 10ème position en octave 3, l'adresse à déterminer sera NOTES+4*((OCTAVE+1*12)+position)

Récupération de la fréquence d'une note de musique

On crée d'abord un mot **set.octave** qui nous permettra de sélectionner l'octave souhaitée. Ensuite, on définit **get.note** qui récupère la fréquence de la note souhaitée :

```

3 value OCTAVE
\ select octave in interval -1..8
: set.octave ( n[-1..8] )
    to OCTAVE
;

```

```

\ select note in interval 1..12
: get.note ( n[1..12] -- )
    1- OCTAVE 1+ 12 * +  cell *      \ calc. offset in NOTES array
    NOTES + @                          \ fetch frequency of selected note
;

3 value OCTAVE
\ select octave in interval -1..8
: set.octave ( n[-1..8] )
    to OCTAVE
;

: OCT6 ( -- )      6 set.octave ;
: OCT5 ( -- )      5 set.octave ;
: OCT4 ( -- )      4 set.octave ;
: OCT3 ( -- )      3 set.octave ;
: OCT2 ( -- )      2 set.octave ;
: OCT1 ( -- )      1 set.octave ;

```

Nous verrons plus loin comment gérer les notes en les appelant depuis leur notation.

Gestion de la durée des notes

La durée d'une note, c'est l'intervalle de temps qui sépare le déclenchement de deux notes consécutives.

Un délai de base est défini par la constante **WHOLE-NOTE-DURATION**.

Les durées sont définies dans un nouveau vocabulaire **music** :

```

1600 constant WHOLE-NOTE-DURATION
WHOLE-NOTE-DURATION value duration

vocabulary music
music definitions
music also

\ set duration of a whole note
: o ( -- )
    WHOLE-NOTE-DURATION to duration
;

\ set duration of a white note
: o| ( -- )
    WHOLE-NOTE-DURATION 2/ to duration
;

\ set duration of a black note
: .| ( -- )
    WHOLE-NOTE-DURATION 2/ 2/ to duration
;

```

```

\ set duration of a half black note
: .|' ( -- )
    WHOLE-NOTE-DURATION 2/ 2/ 2/ to duration
;

\ set duration of a quarter black note
: .|" ( -- )
    WHOLE-NOTE-DURATION 2/ 2/ 2/ 2/ to duration
;

```

On définit des mots qui symbolisent les durées souhaitées: **o** pour une pleine note, **\o** pour une note blanche, **\.** pour une note noire, etc...

Soutien d'une note

Le soutien d'une note correspond au délai pendant lequel la note est audible pendant son temps d'exécution. On définit une valeur **sustain** qui exprime le pourcentage de soutien d'émission de la note pendant sa durée totale. Si cette valeur est à 100, les notes s'enchainent sans aucun silence entre les notes.

```

\ sustain of note, in interval [0..100]
90 value SUSTAIN

ledc
\ sustain note in interval [0..100]
: sustain.note ( -- )
    duration SUSTAIN 100 */ ms
    CHANNEL0 0 ledcWriteTone drop
    duration 100 SUSTAIN - 100 */ ms
;

```

Le mot **sustain.note** génère deux délais. Le premier délai correspond à la durée du maintien de la note. Le second délai correspond à un délai de maintien du silence. La somme de ces deux délais correspond toujours au délai défini dans duration.

Création des notes musicales

On arrive à la partie la plus intéressante, définir les notes par leur nom :

```

: create-note
    \ compile position in octave
    create      ( position -- )

    '
    \ get note frequency in current octave
does>
    @ 1- get.note
    CHANNEL0 swap ledcWriteTone drop
    sustain.note
;

\ notes in english notation
1 create-note C

```

```

2 create-note C#
3 create-note D
4 create-note D#
5 create-note E
6 create-note F
7 create-note F#
8 create-note G
9 create-note G#
10 create-note A
11 create-note A#
12 create-note B

\ notes in french notation
1 create-note DO
2 create-note DO#
3 create-note RE
4 create-note RE#
5 create-note MI
6 create-note FA
7 create-note FA#
8 create-note SOL
9 create-note SOL#
10 create-note LA
11 create-note LA#
12 create-note SI

: SIL ( -- )
    CHANNEL0 0 ledcWriteTone drop
    duration ms
;

forth definitions

```

En plus des douzes notes, de **DO** à **SI**, on définit une note **SIL** qui est un silence.

Test des notes

On teste toutes les notes, gamme par gamme:

```

forth definitions
: music-scale ( -- )
    C C# D D# E F F# G G# A A# B
;

initTones
forth also music also
.|_
80 to SUSTAIN
OCT1 music-scale
OCT2 music-scale
OCT3 music-scale
OCT4 music-scale
OCT5 music-scale

```

Si tout se passe bien, on doit dérouler toutes les notes de musique, par demi-tons, de l'octave 1 à l'octave la plus élevée, ici 6. On ne définit pas d'octave supplémentaire. C'est faisable. Mais les sons émis entrent dans une zone limite pour être audibles.

Le vol du bourdon

Ceci est un premier test de transposition d'une partition musicale. Pour ce faire, on récupère un morceau musical particulièrement difficile, le **VOL DU BOURDON** de **Rimski KORSAKOV**.



Figure 26: première mesure - Le Vol du Bourdon - Rimski KORSAKOV

KORSAKOV. Voici la première mesure de la première ligne :

Voici comment on code cette première mesure, en notation française :

```
OCT5 MI RE# RE DO#      RE DO# DO OCT4 SI
```

Ou en notation anglaise :

```
OCT5 E D# D C#      D C# C OCT4 B
```

Voici le code la première ligne de cette partition :

```
: 1stLine ( -- )
    . |" ( duration of a quarter black note )
    OCT5 MI RE# RE DO#      RE DO# DO OCT4 SI
    OCT5 DO OCT4 SI LA# LA      SOL# SOL FA# FA
    MI RE# RE DO#      RE DO# DO OCT3 SI
;
```

Toutes mes excuses si j'ai fait des erreurs de traduction de la partition. A ce stade, il est facile de tester cette ligne musicale :

```
: flightBumbleBee ( -- )
    initTones
    1stLine
    ;
flightBumbleBee
```

On code deux autres lignes:

```
: 2ndLine ( -- )
    . |" ( duration of a quarter black note )
    OCT4 DO OCT3 SI FA# FA      SOL# SOL FA# FA
    MI RE# RE DO#      RE DO DO# OCT2 SI OCT3
    MI RE# RE DO#      RE DO DO OCT2 SI OCT3
```

```
MI RE# RE DO#      DO FA FA RE#
;

: 3rdLine ( -- )
    .|"  ( duration of a quarter black note )
MI RE# RE DO#      DO DO# RE RE#
MI RE# RE DO#      DO FA FA RE#
MI RE# RE DO#      DO DO# RE RE#
MI RE# RE DO#      RE DO DO# OCT2 SI OCT3
;

: flightBumbleBee ( -- )
  initTones
  1stLine
  2ndLine
  3rdLine
;
flightBumbleBee
```

On vous laisse coder les trois autres lignes de la partition.

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 aisément 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

```
CALLO, CALLX0, RET, J, JX,
```

Branchements 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 aisée 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 - équivalent à SP@
code mySP@
    a1 32      ENTRY,
    a8 a2      MOV.N,  \ copie contenu de a2 dans a8
    a2 a2 4    ADDI,   \ incrémente 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

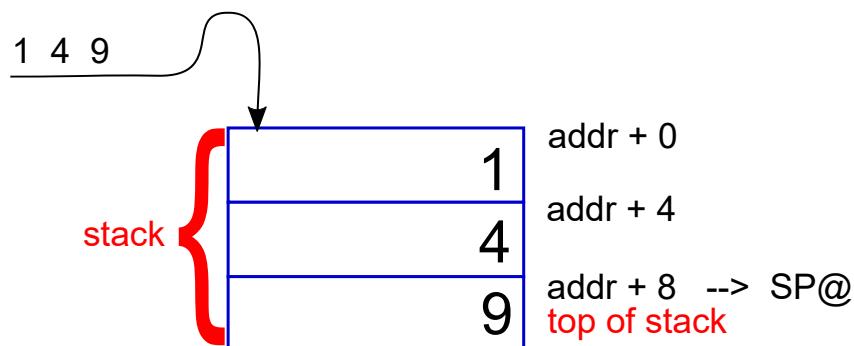
code mySWAP
    a1 32      ENTRY,
    a9  arPOP,
    a8  arPOP,
    a9  arPUSH,
    a8  arPUSH,
                RETW.N,
end-code

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'incrémentation 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

```

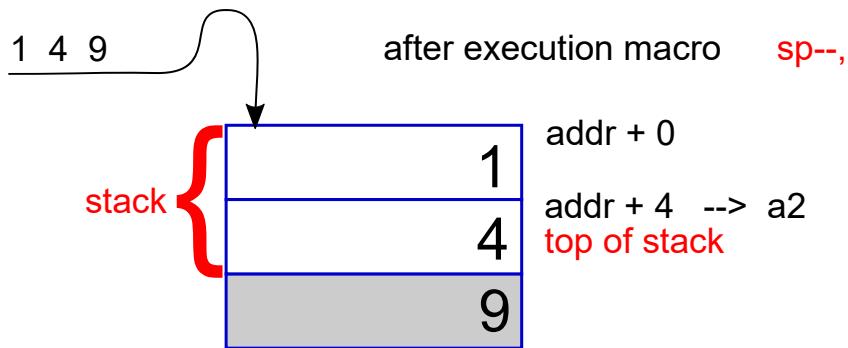
Il y a un autre moyen pour récupérer les données depuis la pile en utilisant l'instruction **L32I.N**,. Cette instruction utilise un index immédiat :

```

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 ← 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 ← 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 ← 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 désassemblier 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 -- f1 )      \ f1=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]	<code><></code>
BGE	Branch if Greater Than or Equal	AR[s] ≥ AR[t]	<code><</code>
BLT	Branch if Less Than	AR[s] < AR[t]	<code>>=</code>
BNE	Branch if Not Equal	AR[s] ≠ AR[t]	<code>=</code>

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 -- f1 )      \ f1=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
```

Définition et manipulation de registres

Dans le document technique de ESP32, on retrouve une très grande quantité de registres. Ces registres permettent de contrôler tous les périphériques et ports GPIO de la carte ESP32.

SAR ADC2 control registers			
SENS_SAR_READ_CTRL2_REG	SAR ADC2 data and sampling control	0x3FF48890	R/W
SENS_SAR_MEAS_START2_REG	SAR ADC2 conversion control and status	0x3FF48894	RO
ULP coprocessor configuration register			
SENS_ULP_OP_SLEEP_CYCLE_REG	Sleep cycles for ULP coprocessor	0x3FF48818	R/W
Pad attenuation configuration registers			
SENS_SAR_ATTEN1_REG	2-bit attenuation for each pad	0x3FF48834	R/W
SENS_SAR_ATTEN2_REG	2-bit attenuation for each pad	0x3FF48838	R/W
DAC control registers			
SENS_SAR_DAC_CTRL1_REG	DAC control	0x3FF48898	R/W
SENS_SAR_DAC_CTRL2_REG	DAC output control	0x3FF4889C	R/W

Figure 27: extrait de Technical Reference manual

En général, la manipulation de ces registres est effectuée par la couche applicative proposée par ESP32forth. Il n'est donc pas nécessaire d'y accéder directement.

Dans certains cas, il peut être intéressant de gérer des registres directement:

- pour accéder à des fonctionnalités non proposées par ESP32forth
- pour exécuter plus rapidement du code FORTH

Définition de registres

Définir un registre est très simple :

```
$3FF48898 constant SENS_SAR_DAC_CTRL1_REG \ DAC control
```

Le premier inconvénient, en définissant un registre comme constante, c'est qu'à la lecture du code source, on ne pourra pas distinguer le registre des autres constantes. On va donc définir un mot de création de registres comme ceci :

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

$3FF48898 defREG: SENS_SAR_DAC_CTRL1_REG \ DAC control
```

De cette manière, en relisant notre code, nous savons que le mot créé est un registre. L'autre intérêt est qu'on peut modifier **defREG:** pour en changer le comportement: rajout de tests de contrôles, initialisation de paramètres, etc...

Accès au contenu des registres

Exemple, valeurs des bits dans le registre **SENS_SAR_DAC_CTRL1_REG**:

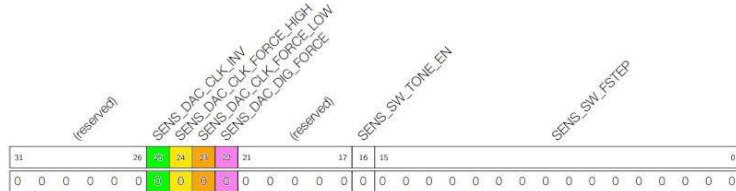


Figure 28: bits dans le registre
ENS_SAR_DAC_CTRL1_REG

Ce registre contient des bits ou blocs de bits ayant des fonctions définies.

Nous allons d'abord créer un mot permettant de visualiser le contenu d'un registre :

```
\ display reg content
: .reg ( reg -- )
  base @ >r
  binary
  @ <#
  4 for
    aft
    8 for
      aft  #  then
    next
    bl hold
  then
next
#>
cr space ." 33222222 22221111 11111100 00000000"
cr space ." 10987654 32109876 54321098 76543210"
cr type
r> base !
;
```

Voyons ce que donne le contenu de notre registre **SENS_SAR_DAC_CTRL1_REG** :

```
SENS_SAR_DAC_CTRL1_REG .reg
\ display:
33222222 22221111 11111100 00000000
10987654 32109876 54321098 76543210
00000000 00000000 00000000 00000000 ok
```

Les deux premières lignes permettent de lire verticalement le rang d'un bit dans ce registre, ici, en rouge, 25, dont le contenu est 0. Pour lire ce bit, on procède comme suit :

```
SENS_SAR_DAC_CTRL1_REG @
1 25 lshift and
```

Pour modifier ce bit et le mettre à 1 :

```
1 25 lshift
SENS_SAR_DAC_CTRL1_REG @
```

```
or
SENS_SAR_DAC_CTRL1_REG !
```

Vérifions avec **.reg** :

```
SENS_SAR_DAC_CTRL1_REG .reg
\ display:
33222222 22221111 11111100 00000000
10987654 32109876 54321098 76543210
00000010 00000000 00000000 00000000 ok
```

Si c'est pour une fois, ça dépanne. Voyons comment faire de manière plus efficace...

Manipulation des bits des registres

Reprendons la modification du bit 25 de notre registre **SENS_SAR_DAC_CTRL1_REG**, voici comment mettre à 1 le bit b25 :

```
SENS_SAR_DAC_CTRL1_REG .reg      \ display:
\ 33222222 22221111 11111100 00000000
\ 10987654 32109876 54321098 76543210
\ 00000000 10000000 00000000 00000000 ok

registers
1 25 $02000000 SENS_SAR_DAC_CTRL1_REG m!
SENS_SAR_DAC_CTRL1_REG .reg      \ display:
\ 33222222 22221111 11111100 00000000
\ 10987654 32109876 54321098 76543210
\ 00000010 00000000 00000000 00000000 ok
```

On utilise le mot **m!** (val shift mask addr --) qui accepte quatre paramètres :

- **val** qui est la valeur à modifier, ici 1
- **shift** qui correspond au décalage à appliquer à cette valeur, ici 25
- **mask** qui correspond au masque logique de la partie de registre à modifier, ici \$02000000
- **addr** qui est l'adresse du registre, ici SENS_SAR_DAC_CTRL1_REG

Définition de masques

Un masque sert à indiquer quels sont les bits modifiables. Dans le précédent exemple, nous avons modifié le bit b25. Dans la documentation Espressif, le bit b25 est marqué avec le label **SENS_DAC_CLK_INV**. La solution la plus simple consisterait à créer une constante comme ceci:

```
1 25 lshift constant SENS_DAC_CLK_INV
```

Mais ça ne règle pas le décalage de valeur qui doit être le même que la valeur du masque binaire.

Voyons une manière plus élégante de définir les masques :

```

: defMASK:
    create ( mask0 position -- )
        dup ,
        lshift ,
    does> ( -- position mask1 )
        dup @
        swap cell + @
;

1 25 defMASK: mSENS_DAC_CLK_INV

```

Au passage, notez que le nom du masque est préfixé avec la lettre '**m**' (pour mask). Ce n'est nullement obligatoire. Mais quand vous aurez compilé de nombreux registres et masques, le préfixe '**m**' permettra de vous y retrouver entre registres et masques :

```

--> words
mSENS_SW_FSTEP mSENS_SW_TONE_EN mSENS_DAC_DIG_FORCE mSENS_DAC_CLK_FORCE_LOW
mSENS_DAC_CLK_FORCE_HIGH mSENS_DAC_CLK_INV defMask: SENS_SAR_DAC_CTRL2_REG
SENS_SAR_DAC_CTRL1_REG GPIO_ENABLE_W1TC_REG GPIO_ENABLE_W1TS_REG GPIO_ENABLE_REG
GPIO_OUT_W1TC_REG GPIO_OUT_W1TS_REG GPIO_OUT_REG DR_REG GPIO_BASE PIN_DAC2
PIN_DAC1 CONFIG_IDF_TARGET_ESP32S3 CONFIG_IDF_TARGET_ESP32S2 .reg AdcREG:
mtst mset mclr --DAdirect SENS_DAC_CLK_INV defMASK: input$ c+$! mid$ left$
right$ 0$! $! maxlen$ string $= FORTH camera-server camera telnetd bterm
.....

```

Le mot défini avec **defMASK:** dépose sur la pile le décalage du masque, ici 25 pour **mSENS_DAC_CLK_INV** et la valeur du masque binaire à appliquer.

Reprendons la modification du bit b25 avec cette définition de masque :

```

1 mSENS_DAC_CLK_INV SENS_SAR_DAC_CTRL1_REG m!
SENS_SAR_DAC_CTRL1_REG .reg \ display:
\ 33222222 22221111 11111100 00000000
\ 10987654 32109876 54321098 76543210
\ 00000010 00000000 00000000 00000000

0 mSENS_DAC_CLK_INV SENS_SAR_DAC_CTRL1_REG m!
SENS_SAR_DAC_CTRL1_REG .reg \ display:
\ 33222222 22221111 11111100 00000000
\ 10987654 32109876 54321098 76543210
\ 00000000 00000000 00000000 00000000

```

Passer du langage C au langage FORTH

Avec ESP32Forth, il y a deux solutions pour rajouter des primitives au dictionnaire :

- réécrire le code source de ESP32Forth en rajoutant les primitives souhaitées;
- réécrire ces mots du langage C en FORTH

La première solution, en langage C est écrite ici :

```
#ifndef ENABLE_DAC_SUPPORT
```

```

#define OPTIONAL_DAC_SUPPORT
#else
#include <driver/dac.h>
#include <driver/dac_common.h>
#include <soc/rtc_io_reg.h>
#include <soc/rtc_ctrl_reg.h>
#include <soc/sens_reg.h>
#include <soc/rtc.h>
#define OPTIONAL_DAC_SUPPORT \
Y(dac_output_enable, n0 = dac_output_enable( (dac_channel_t) n0 ) ) \
Y(dac_output_disable, n0 = dac_output_disable( (dac_channel_t) n0 ) ) \
Y(dac_output_voltage, n0 = dac_output_voltage((dac_channel_t) n1, (gpio_num_t) n0); NIP ) \
Y(dac_cw_generator_enable, PUSH dac_cw_generator_enable () ) \
Y(dac_cw_generator_disable, PUSH dac_cw_generator_disable () ) \
Y(dac_i2s_enable, PUSH dac_i2s_enable() ) \
Y(dac_i2s_disable, PUSH dac_i2s_disable() ) \
Y rtc_freq_div_set, REG_SET_FIELD(RTC_CNTL_CLK_CONF_REG, RTC_CNTL_CK8M_DIV_SEL, n0 ); DROP ) \
Y(dac_freq_step_set, SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL1_REG, SENS_SW_FSTEP, n0, SENS_SW_FSTEP_S); \
DROP ) \
Y(dac1_scale_set, SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_SCALE1, 0, SENS_DAC_SCALE1_S); ) \
Y(dac2_scale_set, SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_SCALE2, 0, SENS_DAC_SCALE2_S); ) \
Y(dac1_offset_set, SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_DC1, n0, SENS_DAC_DC1_S); DROP ) \
Y(dac2_offset_set, SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_DC2, n0, SENS_DAC_DC2_S); DROP ) \
Y(dac1_invert_set, SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_INV1, n0, SENS_DAC_INV1_S); DROP ) \
Y(dac2_invert_set, SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_INV2, n0, SENS_DAC_INV2_S); DROP ) \
Y(dac1_cosine_enable, SET_PERI_REG_MASK(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_CW_EN1_M); ) \
Y(dac2_cosine_enable, SET_PERI_REG_MASK(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_CW_EN2_M); ) \
Y(dacWrite, dacWrite(n1, n0); DROPN(2))
#endif

```

La seconde solution consiste à regarder le code source d'un fichier écrit en langage C et essayons de comprendre comment les registres sont manipulés dans ce langage.

Extrait du fichier **dac-cosine.c**:

```

/*
 * Enable cosine waveform generator on a DAC channel
 */
void dac_cosine_enable(dac_channel_t channel)
{
    // Enable tone generator common to both channels
    SET_PERI_REG_MASK(SENS_SAR_DAC_CTRL1_REG, SENS_SW_TONE_EN);
    switch(channel) {
        case DAC_CHANNEL_1:
            // Enable / connect tone generator on / to this channel
            SET_PERI_REG_MASK(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_CW_EN1_M);
            // Invert MSB, otherwise part of waveform will have inverted
            SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_INV1, 2, SENS_DAC_INV1_S);
            break;
        case DAC_CHANNEL_2:
            SET_PERI_REG_MASK(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_CW_EN2_M);
            SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_INV2, 2, SENS_DAC_INV2_S);
            break;
        default :
            printf("Channel %d\n", channel);
    }
}

```

Une des fonctions C qui revient souvent est **SET_PERI_REG_MASK**. Cette fonction met à 1 les bits désignés par un masque dans un registre. Exemple:

```
SET_PERI_REG_MASK(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_CW_EN1_M);
```

La fonction C qui met à 0 les bits désignés par un masque dans un registre est **CLEAR_PERI_REG_MASK**.

On va s'intéresser à la manière dont on va réécrire **dac_cosine_enable(dac_channel_t channel)** en langage FORTH. On voit que le registre SENS_SAR_DAC_CTRL2_REG est mentionné. On va définir ce registre :

```
$3FF4889c defREG: SENS_SAR_DAC_CTRL2_REG \ DAC output control
```

Dans ce registre **SENS_SAR_DAC_CTRL2_REG**, les deux bits qui nous intéressent sont b24 et b25. Définissons les masques correspondants :

```
1 24 defMASK: mSENS_DAC_CW_EN1 \ selects CW generator as source for PDAC1  
1 25 defMASK: mSENS_DAC_CW_EN2 \ selects CW generator as source for PDAC2
```

La programmation 'bare-metal' agissant directement sur les registres de ESP32 ne nécessite pas de définir en FORTH tous les registres et masques de registres comme le fait le langage C. Limitez-vous aux registres et masques essentiels pour votre application.

Il est conseillé d'utiliser les noms des registres et masques de registres tels que figurant dans la documentation Espressif, ou à défaut les noms de registres utilisés dans les codes sources en langage C.

La gestion de certains périphériques est très complexe. La documentation Espressif est avare d'exemples sur l'utilisation directe des registres.

Le générateur de nombres aléatoires

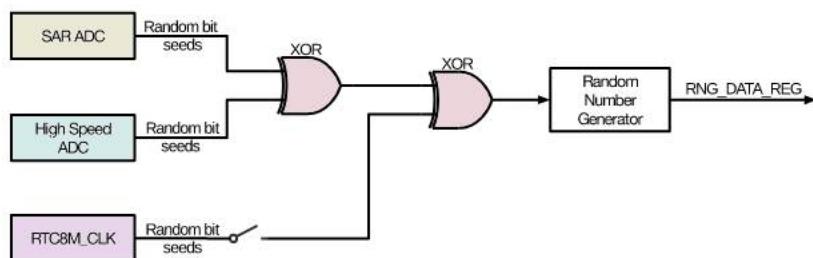
Caractéristique

Le générateur de nombres aléatoires génère de vrais nombres aléatoires, ce qui signifie un nombre aléatoire généré à partir d'un processus physique, plutôt qu'au moyen d'un algorithme. Aucun nombre généré dans la plage spécifiée n'est plus ou moins susceptibles d'apparaître que tout autre nombre.

Chaque valeur de 32 bits que le système lit dans le registre RNG_DATA_REG du générateur de nombres aléatoires est un vrai nombre aléatoire. Ces vrais nombres aléatoires sont générés en fonction du bruit thermique dans le système et le décalage d'horloge asynchrone.

Le bruit thermique provient de l'ADC haute vitesse ou de l'ADC SAR ou des deux. Chaque fois que l'ADC haute vitesse ou l'ADC SAR est activé, les flux de bits seront générés et introduits dans le générateur de nombres aléatoires via une porte logique XOR comme graines aléatoires.

Lorsque l'horloge RTC8M_CLK est activée pour le noyau numérique, le générateur de nombres aléatoires échantillonnera également RTC8M_CLK (8 MHz) en tant que graine binaire aléatoire. RTC8M_CLK est une source d'horloge asynchrone et elle augmente l'entropie RNG en introduisant la métastabilité du circuit. Cependant, pour assurer une entropie maximale, il est également recommandé de toujours activer une source ADC.



Lorsqu'il y a du bruit provenant du SAR ADC, le générateur de nombres aléatoires est alimenté avec une entropie de 2 bits dans un cycle d'horloge de RTC8M_CLK (8 MHz), qui est généré à partir d'un oscillateur RC interne (voir le chapitre Réinitialisation et Horloge pour plus de détails). Ainsi, il est conseillé de lire le registre **RNG_DATA_REG** à une cadence maximum de 500 kHz pour obtenir l'entropie maximale.

Lorsqu'il y a du bruit provenant du CAN haute vitesse, le générateur de nombres aléatoires est alimenté par une entropie de 2 bits dans un cycle d'horloge APB, qui est normalement de 80 MHz. Ainsi, il est conseillé de lire le registre **RNG_DATA_REG** à un débit maximal de 5 MHz pour obtenir l'entropie maximale.

Un échantillon de données de 2 Go, qui est lu à partir du générateur de nombres aléatoires à une fréquence de 5 MHz avec seulement la haute speed ADC étant activé, a été testé à l'aide de la suite de tests Dieharder Random Number (version 3.31.1). le l'échantillon a réussi tous les tests.

Procédure de programmation

Lors de l'utilisation du générateur de nombres aléatoires, assurez-vous qu'au moins l'ADC SAR, l'ADC haute vitesse ou le RTC8M_CLK est autorisé. Sinon, des nombres pseudo-aléatoires seront renvoyés.

- SAR ADC peut être activé à l'aide du contrôleur DIG ADC.
- L'ADC haut débit est activé automatiquement lorsque les modules Wi-Fi ou Bluetooth sont activés.
- RTC8M_CLK est activé en définissant le bit RTC_CNTL_DIG_CLK8M_EN dans le registre RTC_CNTL_CLK_CONF_REG.

Lorsque vous utilisez le générateur de nombres aléatoires, lisez le registre **RNG_DATA_REG** plusieurs fois jusqu'à ce qu'il y ait suffisamment d'aléas des nombres générés.

Name	Description	Address	Access
RNG_DATA_REG	Random number data	\$3FF75144	RO

```
\ Random number data
$3FF75144 constant RNG_DATA_REG

\ get 32 bits random b=number
: rnd  ( -- x )
    RNG_DATA_REG L@
;

\ get random number in interval [0..n-1]
: random ( n -- 0..n-1 )
    rnd swap mod
;
```

Fonction RND en assembleur XTENSA

Depuis la version 7.0.7.4, ESP32forth dispose d'un assembleur XTENSA. Il est possible de réécrire notre mot **rnd** en assembleur XTENSA:

```
forth definitions
asm xtensa
$3FF75144 constant RNG_DATA_REG

code myRND ( -- [addr] )
    a1 32          ENTRY,
    a8 RNG_DATA_REG L32R,      \ a8 = RNG_DATA_REG
    a9 a8 0        L32I.N,    \ a9 = [a8]
```

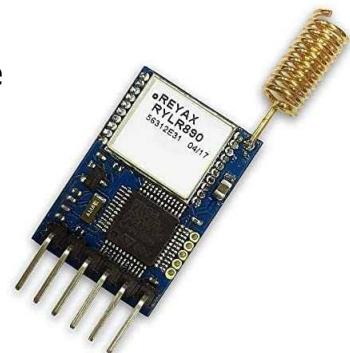
```
a9          arPUSH,      \ push a9 on stack
            RETW.N,
end-code
```

Le système de transmission LoRa

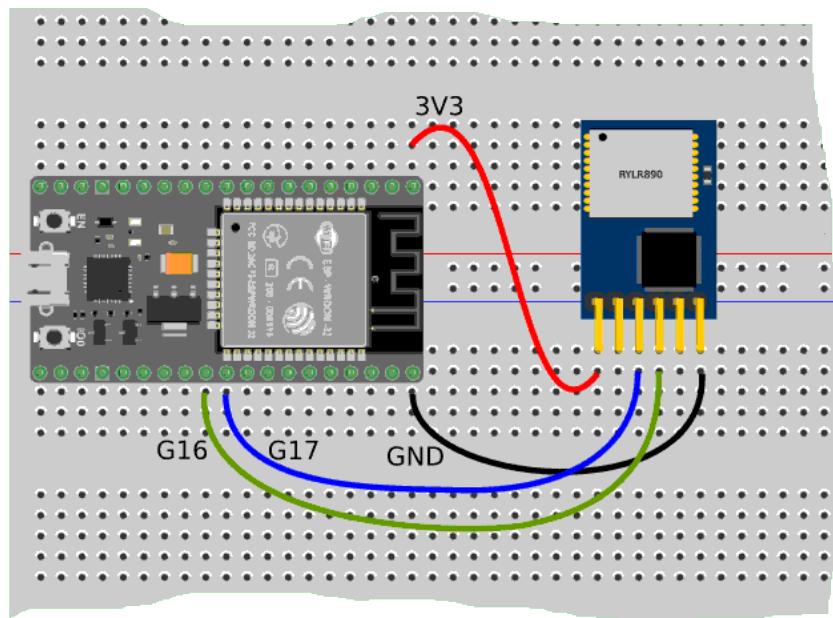
LoRa est une technologie de communication qui utilise un réseau étendu à faible consommation. LoRa permet de connecter sans fil des appareils et des passerelles.

Ce standard ne nécessite encore aucun abonnement. Il offre la communication **de pair à pair**.

LoRa, WiFi et Bluetooth sont complémentaires et ne se recoupent pas. Comparé au Wi-Fi et au Bluetooth qui offrent une très courte portée, LoRa bénéficie d'une bande passante très étroite. Les passerelles ou les concentrateurs sont utilisés à peine 1% du temps par les dispositifs connectés. Ce qui réduit considérablement la bande passante. Le trafic est lent et unidirectionnel entre les capteurs et la passerelle. LoRa est le meilleur moyen pour communiquer sur plusieurs kilomètres, avec très peu de puissance et de manière très simple !



Câblage du transmetteur LoRa REYAX LR890



Le transmetteur est raccordé à la carte ESP32 comme ceci :

ATTENTION: vérifiez la position des pins G16 et G17 sur votre carte ESP32 qui peut être différente selon votre version de carte ESP32.

Le transmetteur LoRa pour ESP32

Le module REYAX LR890 coûte environ 15€. Il pèse 7 grammes.

Sa consommation, en émission, est de 43 mA (3,3V). En réception, elle est de 16,5 mA et peut descendre à 0,5 mA en mode SLEEP.

Pour assurer une transmission point à point, il faut deux modules LoRa. Chaque module est émetteur et récepteur.

La carte ESP32 communique via son port série avec le module LoRa. Toutes les transmissions entre la carte ESP32 et le transmetteur LoRa sont traitées au travers des commandes AT. Exemple:

```
AT+SEND=50,5,HELLO
```

Cette chaîne est transmise par la carte ESP32 au module de transmission LoRa :

- le module LoRa passe en mode émission et transmet cette chaîne de caractères
- immédiatement après la transmission, le module LoRa repasse en mode réception
- le module LoRa distant reçoit la chaîne de caractères.
- le module LoRa distant peut acquiter cette réception par **+OK**

Un module LoRa peut communiquer avec une passerelle LoRaWan. C'est en général un boîtier connecté à un routeur par liaison Ethernet. Il est donc possible d'avoir une application web qui communique avec un ou plusieurs modules LoRa.

Sécurité des transmission LoRa

Un même module LoRa peut communiquer avec plusieurs modules LoRa distants.

Ces modules LoRa doivent être différenciés par leur **NETWORKID**. L'émetteur et le récepteur doivent avoir le même **NETWORKID**.

Ensuite, chaque module reçoit une **ADDRESS**, par défaut 0. Cette adresse est comprise entre 0..65535.

La transmission peut être cryptée par **clé AES** de 32 caractères. Les modules LoRa émetteur et récepteur doivent avoir la même **clé AES**. Si un module reçoit un message crypté avec une **clé AES** inconnue, il ignorera le message.

Et pour terminer, chaque module se voit attribuer une fréquence de transmission. Les modules émetteurs et récepteurs doivent travailler sur la même fréquence.

Exemple de sélection de fréquence 868.5 Mhz.

```
\ select frequency 868.5 Mhz for LoRa transmission
32 string AT_BAND
s" AT+BAND=868500000" AT_BAND $!  \ set frequency at 868.5 Mhz
$0a AT_BAND c+$!
$0d AT_BAND c+$!      \ add CR LF code at end of command
AT_BAND Serial2.write drop
```

Sur une même fréquence, on peut gérer un parc de 65535 modules LoRa, chaque module ayant son adresse. Si on transmet avec l'adresse 0, on s'adressera à tous les modules LoRa.

Si on rajoute la clé de cryptage AES, ce seront des centaines de milliers de modules LoRa qui peuvent cohabiter dans un rayon de quelques kilomètres!

La portée des modules peut être augmentée en modifiant la puissance d'émission. On peut aussi agir sur l'antenne de réception. Avec une antenne directionnelle, on peut atteindre **20 à 30 kilomètres** de portée...

Test du transmetteur LoRa REYAX RYLR890

Environnement de test requis

Pour tester notre transmetteur LoRa REYAX RYLR890, il faut:

- utiliser les mots de gestion de chaînes.... @todo : référence dans fichier
- utiliser une version ESP32Forth avec accès au port UART2
- câbler le transmetteur LoRa REYAX RYLR890 comme suit:

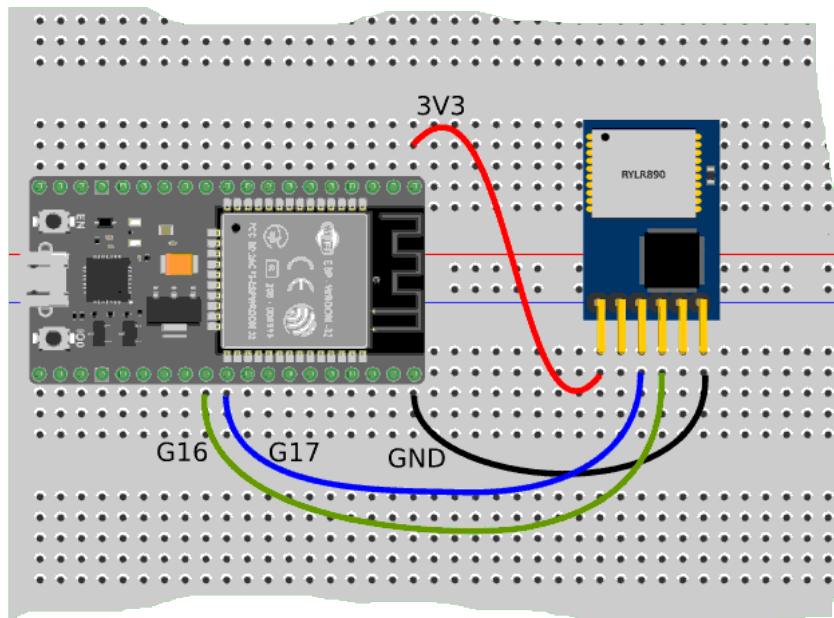


Figure 29: branchement du transmetteur LoRa

Préparer la communication avec le transmetteur LoRa

Tous les programmeurs amenés à gérer le port UART2 définissent une zone mémoire qui va servir de tampon. Pour notre part, nous allons créer directement une variable alphanumérique:

```
128 string LoRaTX \ buffer ESP32 -> LoRa transmitter
```

Ici, nous allons seulement procéder à des tests d'envoi de commandes vers le transmetteur LoRa REYAX RYLR890 et voir comment récupérer ce que ce même transmetteur renvoie. Il nous faut donc une autre variable alphanumérique pour la réception :

```
128 string LoRaRX \ buffer LoRa transmitter -> ESP32
```

Commençons par initialiser la transmission série vers le transmetteur LoRa. Ici, la vitesse est de 115200 bauds. C'est la vitesse de transmission par défaut du transmetteur LoRa :

```

Serial \ Select Serial vocabulary

\ initialise Serial2
: Serial2.init ( -- )
    #SERIAL2_RATE Serial2.begin
;

```

Pour notre commande de test vers le transmetteur LoRa, on sélectionne la fréquence de travail du transmetteur, ici 868.5 Mhz :

```

\ Setup LoRa Frequency
: .band8685 ( -- )
    s" AT+BAND=868500000" LoRaTX $!
    $0d LoRaTX c+$!
    $0a LoRaTX c+$!      \ add CR LF code at end of command
    LoRaTX Serial2.write drop
;

```

Enfin, on définit un mot permettant de récupérer la réponse du transmetteur LoRa :

```

\ input from LoRa transmitter
: LoRaInput ( -- n )
    Serial2.available dup if
        LoRaRX maxlen$ nip
        Serial2.readBytes
        LoRaRX drop cell - !
    then
;

```

Le mot **LoRaInput** teste si une transmission par liaison série a été reçue depuis le port série UART2 :

- s'il n'y a pas de réception, renvoie 0
- s'il a des caractères, stocke ces caractères dans la chaîne alphanumérique LoRaRX et met à jour la taille de cette chaîne.

Exemple de transmission et réception :

```

Serial2.init
.band8685
LoRaInput

```

Voici ce que donne un dump mémoire de **LoRaRX**:

```

LoRaRX dump
--addr--- 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  -----
chars-----
3FFF-87A0 05 00 00 00 2B 4F 4B 0D 0A 31 0D 0A 2B 4F 4B 0D  ....
+OK..1..+OK.

```

Si on exécute **LoRaRX**, on récupère l'adresse et surtout la longueur de tous les caractères reçus, incluant les caractères CR+LF (\$0d \$0a). Pour ne gérer que les caractères

strictement compris dans l'intervalle [0..0A..Za..z], il faut diminuer la taille de la chaîne de deux unités:

```
: LoRaType ( -- )
    LoRaRX dup 0 > if
        2 - type
    else
        2drop
    then
;
LoRaType \ display: +OK
```

Ici, l'exécution de **LoRaType** affiche +OK qui est la réponse à notre commande de test **AT+BAND=868500000**.

Paramétrage du transmetteur LoRa REYAX RYLR890

Avant de définir des commandes pour notre transmetteur LoRa REYAX RYLR896, on définit le mot **crlf**:

```
: crlf ( -- )          \ same action as cr, but adapted for LoRa
    $0d emit
    $0a emit
;
```

Le but de ce mot crlf est de terminer la transmission sur le port UART2 depuis la carte ESP32 vers le transmetteur LoRa. La définition de ce mot utilise emit. Ne soyez pas surpris. On verra plus loin comment exploiter l'exécution vectorisée de mots en langage FORTH pour faire l'action souhaitée à emit. Cette solution surprendra les débutants programmeurs en langage FORTH. Elle montrera aussi à quel point FORTH est bien plus souple que beaucoup d'autres langages de programmation.

Paramètres essentiels

Voici la liste des paramètres essentiels pour paramétrer votre module LoRa.

La séquence d'utilisation de la commande **AT**:

1. Utilisez **AT+ADDRESS** pour régler ADRESSE. L'ADRESSE est considérée comme l'identification de l'émetteur ou du récepteur spécifié.
2. Utilisez **AT+NETWORKID** pour définir l'ID du réseau Lora. Ceci est une fonction de groupe. Ce n'est qu'en définissant le même NETWORKID que les modules peuvent communiquer entre eux. Si l'ADRESSE du destinataire spécifié appartient à un groupe différent, elle n'est pas capables de communiquer entre eux. La valeur recommandée: 1 ~ 15
3. Utilisez **AT+BAND** pour régler la fréquence centrale de la bande sans fil. L'émetteur et le récepteur doit utiliser la même fréquence pour communiquer entre eux.
4. Utilisez **AT+PARAMETER** pour régler les paramètres sans fil RF. L'émetteur et le récepteur doit définir les mêmes paramètres pour communiquer entre eux. Les paramètres sont à définir comme suit:
 1. **<Spreading Factor>**: Plus le SF est grand, meilleure est la sensibilité. Mais le temps de transmission prendra plus de temps.
 2. **<Bandwidth>**: Plus la bande passante est petite, meilleure est la sensibilité. Mais le temps de transmission prendra plus de temps.

2. **<Coding Rate>**: Le taux de codage sera le plus rapide si vous le définissez sur 1.
 3. **<Programmed Preamble>**: Code préambule. Si le code du préambule est plus gros, il se traduira par moins de chances de perdre des données. Code de préambule généralement peut être réglé au-dessus de 10 si sous l'autorisation de l'heure de transmission.
 - * Communication jusqu'à 3 km: réglage recommandé "AT+PARAMETER=10,7,1,7"
 - * Plus de 3 km: réglage recommandé "AT+PARAMETER=12,4,1,7"
2. Utilisez **AT+SEND** pour envoyer des données à l'ADRESSE spécifiée. En raison du programme utilisé par le module, la partie charge utile augmentera de plus de 8 octets par rapport à la longueur réelle des données.

Il est nécessaire de transmettre **crlf** à la fin de toutes les commandes **AT**.

Il faut attendre que le module réponde **+OK** pour que vous puissiez exécuter la prochaine commande **AT**.

ADDRESS Définit l'adresse du module

Chaque module de transmission LoRa doit avoir une adresse personnelle.

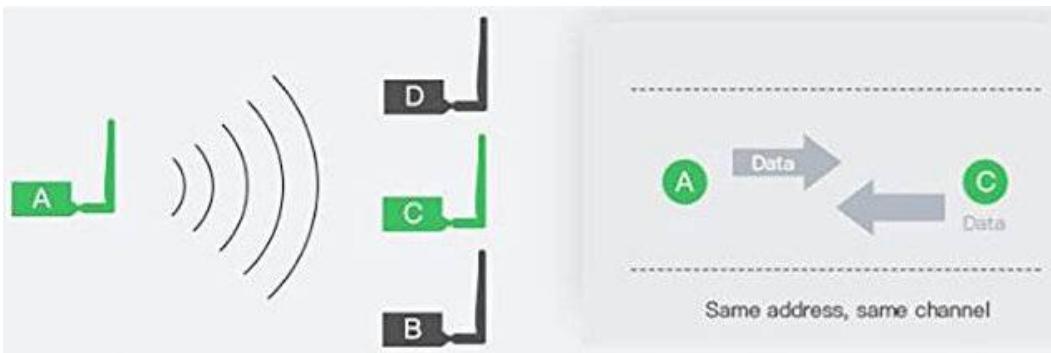
syntax	response
AT+ADDRESS=<address>	+OK
AT+ADDRESS=?	+ADDRESS=22

```
\ Set the ADDRESS of LoRa transmitter:
\ s" " value in interval [0..65535][?] (default 0)
: ATaddress ( addr len -- )
  ." AT+ADDRESS="
  type crlf
;
```

<Address>=0~65535(default 0)

Exemple: Définir l'adresse du module sur **22**. Les paramètres seront mémorisés dans LoRa.

```
s" 22" Ataddress
```



AT Test Disponibilité LoRa

syntax	response
AT	+OK

```
\ Test LoRa disponibility
: AT_ ( -- )
    ." AT"
    type crlf
;
```

BAND Réglage de la fréquence RF

syntax	response
AT+BAND=<parameter>	+OK
AT+BAND=?	+BAND=868500000

```
\ Set the BAND of LoRa transmitter:
\ s" " value is RF frequency, unit Hz
: ATband ( addr len -- )
    ." AT+BAND="
    type crlf
;
```

Parameter est la fréquence RF, unité est en Hz: 91500000Hz (default: RLY89x)

Example : Sélectionne la fréquence à 86850000Hz:

```
s" 868500000" ATband
```

CPIN Définit le mot de passe AES128 du réseau

syntax	response
AT+CPIN=<password>	+OK
AT+CPIN=?	+CPIN=FABC0002EEDCA.....

Password: mot de passe AES de 32 caractères de 00000000000000000000000000000001 à FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF.

L'échange est accepté si les deux modules ont le même mot de passe. Après reset, le précédent mot de passe est effacé.

```
\ Set the AES32 password:  
\ s" <parameter>" value is an 32 character long AES password  
\ from 00000000000000000000000000000001 to  
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFF  
: ATcpin ( addr len -- )  
. " AT+CPIN="  
type crlf  
;
```

Example: Sélectionne ce mot de passe : FABC0002EEDCAA90FABC0002EEDCAA90

```
s" FABC0002EEDCAA90FABC0002EEDCAA90" ATcpin
```

CRFOP Sélectionne la puissance RF de sortie

syntax	response
AT+CRFOP=<power>	+OK
AT+CRFOP=?	+CRFOP=10

Puissance: entre 0..15, 15dBm (par défaut)

```
\ Set the CRFOP RF output power:  
\ s" " value is RF output power between 0..15  
: ATcrfop ( addr len -- )  
. " AT+CRFOP="  
type crlf  
;
```

Example, sélectionne la puissance de sortie à 10dBm :

```
s" 10" ATcrfop
```

FACTORY Règle tous les paramètres actuels sur les valeurs par défaut

Règle tous les paramètres actuels sur les valeurs par défaut du fabricant.

syntax	response
AT+FACTORY	+FACTORY

```
\ Reset the LoRa transmitter with FACTORY parameters  
: ATfactory ( -- )  
. " AT+FACTORY"  
crlf  
;
```

IPR Règle le débit UART en bauds

syntax	response
AT+IPR=<parameter>	+OK
AT+IPR=?	+IPR=38400

Paramètre de baud UART:

- 300
- 1200
- 4800
- 9600
- 19200
- 28800
- 38400
- 57600
- 115200 (par défaut)

Les réglages seront mémorisés dans l'EEPROM.

MODE Sélectionne le mode de travail

syntax	response
AT+MODE=<parameter>	+OK
AT+MODE=?	+MODE=1

Paramètre:

- 0: Transmit and Receive mode (default).
- 1: Sleep mode.

```
\ Set work MODE:  
\ s" " value is [0,1]  
\ 0 (default) Transmit and Receive mode  
\ 1 Sleep mode  
: ATmode ( addr len -- )  
    ." AT+MODE"  
    type crlf  
    ;
```

NETWORKID Sélectionne l'ID réseau

syntax	response
AT+NETWORKID=<Network ID>	+OK
AT+NETWORKID=?	+NETWORKID=6

```
\ Set NETWORKID:  
\ s" " value is [0..16] (0 bay default)  
: ATnetworkid ( addr len -- )  
    ." AT+NETWORKID"  
    type crlf  
;
```

Network ID: 0~16(0 par défaut)

Exemple: sélectionne network ID à 6

Les réglages seront mémorisés dans l'EEPROM.

Le «0» est l'identifiant public de LoRa. Il n'est pas recommandé d'utiliser 0 pour définir l'ID RÉSEAU.

```
s" 6" Atnetworkid
```

PARAMETER définition des paramètres RF

syntax	response
AT+PARAMETER=<Spreading Factor>, <Bandwidth>, <Coding Rate>, <Programmed Preamble>	+OK
AT+PARAMETER=?	+PARAMETER=7,3,4,5

Parameters:

- Spreading Factor / facteur d'étalement
 - 7~12, (default 12)
- Bandwidth / largeur de bande (0~9):
 - 0 : 7.8KHz (not recommended, over spec.)
 - 1 : 10.4KHz (not recommended, over spec.)
 - 2 : 15.6KHz
 - 3 : 20.8 KHz
 - 4 : 31.25 KHz

- 5 : 41.7 KHz
- 6 : 62.5 KHz
- 7 : 125 KHz (default).
- 8 : 250 KHz
- 9 : 500 KHz
- Coding rate
 - 1~4, (default 1)
- Programmed Preamble
 - 4~7 (default 4)

Spreading Factor / facteur d'étalement

Spreading factor facteur d'étalement	Bitrate / débit
7	5469 bps
8	3125 bps
9	1758 bps
10	977 bps
11	537 bps
12	293 bps

Taux de codage

La modulation LoRa ajoute également une correction d'erreur directe (FEC) dans chaque transmission de données. Cette implémentation se fait en encodant des données 4 bits avec des redondances en 5 bits, 6 bits, 7 bits, voire 8 bits. L'utilisation de cette redondance permettra au signal LoRa de couvrir des interférences. La valeur du taux de codage doit être ajustée en fonction des conditions du canal utilisé pour la transmission des données. S'il y a trop d'interférences dans le canal, alors il est recommandé d'augmenter la valeur du taux de codage.

Cependant, l'augmentation de la valeur CR augmentera également la durée de la transmission.

Exemple: définition des paramètres comme ci-dessous:

<Spreading Factor> 7,<Bandwidth> 20.8KHz, <Coding Rate> 4,<Programmed Preamble>5,

```
s" 7,3,4,5" Atparameter
```

RESET logiciel

syntax	response
AT+RESET	+OK

```
\ RESET the LoRa transmitter
: ATreset ( -- )
    ." AT+RESET"
    crlf
;
```

SEND envoi de données à l'adresse désignée

syntax	response
AT+SEND=<Address>,<Payload Length>,<Data>	+OK
AT+SEND=?	+SEND=50,5,HELLO

<Address>0~65535, lorsque l'<adresse> est 0, il enverra des données à tous adresse (de 0 à 65535.)

<Payload Length>Maximum 240 octets

<Data>ASCII Format

Code Forth pour ESP32Forth :

```
\ convert a number to a decimal string
: .n ( n ---)
    base @ >r decimal
    <# #s #> type
    r> base !
;
\ SEND Send data to the appointment address
: ATsend { addr len address -- }
    ." AT+SEND="
    address .n [char] , emit
    len .n [char] , emit
    addr len type crlf
;
```

Exemple: envoie la chaîne **HELLO** à l'Adresse 50 :

```
s" HELLO" 50 ATsend \ display: AT+SEND=50;5;HELLO
```

VER pour demander la version du firmware

```
\ VER to inquire the firmware version
: ATver ( -- )
    ." AT+VER"
```

```
crlf  
;
```

Codes de résultat d'erreur

- +ERR=1 il n'y a pas «enter» ou \$0D \$0A à la fin de la commande AT
- +ERR=2 la tête de la commande AT n'est pas une chaîne «AT»
- +ERR=3 il n'y a pas de symbole «=>» dans la commande AT
- +ERR=4 commande inconnue
- +ERR=10 TX est sur le temps
- +ERR=11 RX est dépassé
- +ERR=12 erreur CRC
- +ERR=13 données TX de plus de 240 octets
- +ERR=15 Erreur inconnue

Vectorisation des émissions de caractères

Si vous avez suivi jusque là le développement de nos mots permettant de paramétrer le transmetteur LoRa REYAX RYLR890, quelque chose vous a certainement surpris :

```
\ Set the ADDRESS of LoRa transmitter:  
\ s" <address>" value in interval [0..65535][?] (default 0)  
: ATaddress ( addr len -- )  
    ." AT+ADDRESS="  
    type  crlf  
    ;
```

Car, sauf erreur, cette séquence `." AT+ADDRESS="` envoie la chaîne de caractères vers notre terminal, et non pas vers le transmetteur via l'éport série UART2, donc vers le transmetteur LoRa!

On comprend votre surprise. Et nous allons voir comment détourner le flux de caractères vers le transmetteur LoRa sans rien changer à la définition de notre mot `ATaddress`.

Comprendre la vectorisation en FORTH

Le langage FORTH dispose de certains atouts totalement inexistants dans bien d'autres langages de programmation. Parmi ces atouts, citons le mot `defer`. Ce mot permet de créer un mot dont l'action est différée :

```
defer myWords
```

`defer` crée un mot `myWords` qui ne fait RIEN!!!

Oui!

C'est à nous maintenant de lui donner une action. Voyons cette définition :

```
: (myWords) ( -- )
    cr ." I display my words: "
;
```

Pour que myWords exécute (myWords) on récupère le code d'action de (myWords) et l'affecter à myWords:

```
' (myWords) is myWords
```

A partir de maintenant, si on tape **myWords**, c'est l'action définie dans **(myWords)** qui sera exécutée.

OK. Mais là, est-ce nécessaire de faire une telle surcharge de code si on peut simplement exécuter **(myWords)** ?

Et vous avez parfaitement raison de poser cette question. Mais on peut changer l'action de **myWords** :

```
' vlist is myWords
```

Maintenant, si on tape **myWords**, c'est le mot **vlist** qui s'exécute.

Nous allons voir comment utiliser ce mécanisme pour modifier le comportement de ESP32Forth...

La vectorisation dans ESP32Forth

Commençons par un peu d'ingénierie inverse. En fouillant le code de ESP32Forth, on trouve ceci pour le mot **.**:

```
: ."
    postpone s" state @ if postpone type else type then ; immediate
```

Ici, le mot qui nous intéresse est **type** dont la définition est :

```
defer type
```

Ahhh.... Commencez-vous à comprendre?

Quelle action exécute **type**? On trouve ceci dans le code source de ESP32Forth :

```
: serial-type ( a n -- ) Serial.write drop ;
: default-type serial-type ;

' default-type is type
```

Si on s'intéresse au mot **emit**, on trouve cette définition dans le code source de ESP32Forth:

```
: emit ( n -- )
    >r rp@ 1 type rdrop ;
```

Là encore, on retrouve **type**.

C'est donc sur ce mot type que nous allons agir pour détourner l'émission de caractères vers le port série UART2.

Vectoriser type vers le port série UART2

C'est en regardant la définition de **serial-type** que nous définissons notre version pour transmettre vers le port série UART2:

```
: serial2-type ( a n -- )
    Serial2.write drop ;
```

Ça va jusque là? C'est pas trop difficile?

Maintenant, pour détourner le flux d'émission des caractères de ESP32Forth vers le port série UART2, il suffit d'exécuter la séquence '**serial2-type is type**'.

Mais si vous faites ça, vous aurez un peu de mal à revenir à un comportement normal de ESP32Forth sauf à restituer à **type** son action initiale avec la séquence '**default-type is type**'.

Encapsulons ces séquences dans ces deux mots :

```
: typeToLoRa ( -- )
    ['] serial2-type is type
;

: typeToTerm ( -- )
    ['] default-type is type
;
```

Et maintenant, pour exécuter notre mot **ATaddress** en lui faisant transmettre les caractères vers le port série UART2, il suffit de taper :

```
typeToLoRa
s" 45" ATaddress \ send AT+ADDRESS=45 to UART2
typeToTerm
```

Et là, j'attends votre remarque: "mais quel intérêt de passer par la vectorisation?"

Dans notre cas, la vectorisation offre beaucoup d'avantages :

- écrire un code simple avec des mots déjà connus du dictionnaire FORTH de ESP32Forth;
- offre la possibilité de tester vers le terminal tous les mots de paramétrage du transmetteur LoRa
- possibilité de détourner le flux vers un autre périphérique, par exemple I2S ou UART1, sans avoir à réécrire ces définitions de paramétrage...

Indépendamment de notre gestion des paramètres du transmetteur LoRa, on comprend qu'il suffit d'exploiter ce même mécanisme de vectorisation pour les caractères reçus

depuis le port série UART2 pour prendre facilement le contrôle de ESP32Forth depuis ce port série!

C'est d'ailleurs ce que fait ESP32Forth si on active le port WiFi ou Bluetooth! Je vous invite à explorer le code source de ESP32Forth. Regardez la définition de **server** :

```
: server ( port -- )
  server
  ['] serve-key is key
  ['] serve-type is type
  webserver-task start-task
;
```

Réécriture d'un listing complet

Les paramètres minimaux pour communiquer entre cartes ESP32+LoRa sont: fréquence et adresse:

```
\ *** defining LoRa Setup words ****

create $crlf
$0d c, $0a c,

:crlf ( -- )           \ same action as cr, but adapted for LoRa
$crlf 2 type
;

\ Set the ADDRESS of LoRa transmitter:
\ s" " value in interval [0..65535][?] (default 0)
: ATaddress ( addr len -- )
." AT+ADDRESS="
type crlf ;

\ Set the BAND of LoRa transmitter:
\ s" " value is RF frequency, unit Hz
: ATband ( addr len -- )
.s" AT+BAND=" type
type crlf ;
```

Les transmetteurs LoRa, sortis de leur emballage d'origine, communiquent théoriquement à 115200 bauds avec la carte ESP32 :

```
\ 115200 speed communication for LoRa REYAX
115200 value #SERIAL2_RATE

\ definition of OUTput and INput buffers
128 string LoRaRX  \ buffer LoRa transmitter -> ESP32

Serial \ Select Serial vocabulary

\ initialise Serial2
: Serial2.init ( -- )
#SERIAL2_RATE Serial2.begin
```

```
;
```

On récupère aussi le mot **LoRaInput** qui lit les messages restitués par le transmetteur LoRa sur le port UART2. Le mot **rx.** a été rajouté pour faciliter les manipulations:

```
\ input from LoRa transmitter
: LoRaInput ( -- n )
    Serial2.available if
        LoRaRX maxlen$ nip
        Serial2.readBytes
        LoRaRX drop cell - !
    else
        0 LoRaRX drop cell - !
then
;

: rx.
    LoRaINPUT
    LoRaRX type
;
```

Ici, les mots **typeToLoRa** et **typeToTerm** permettent de transférer l'affichage de texte du terminal vers le port UART2:

```
\ *** defining defered words ****
serial \ Select Serial vocabulary

: serial2-type ( a n -- )
    Serial2.write drop ;

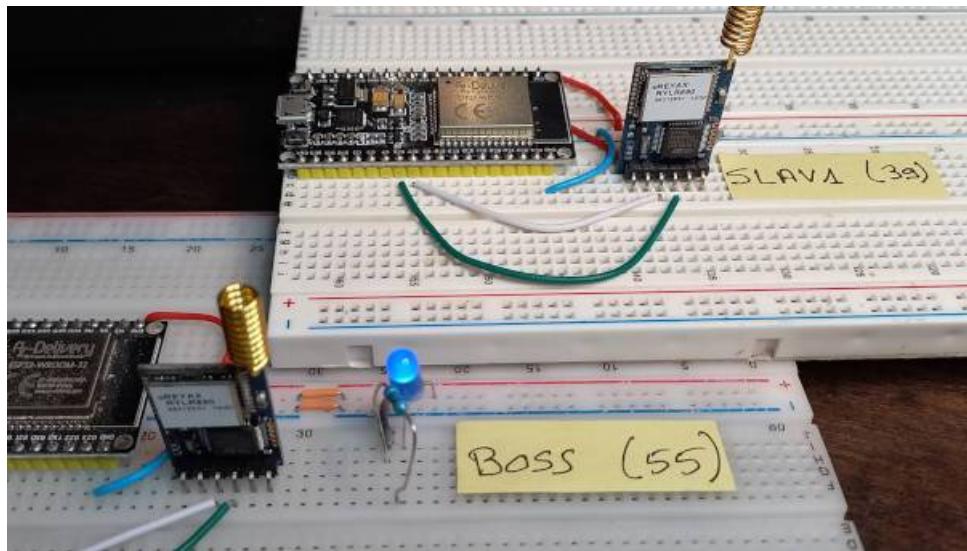
: typeToLoRa ( -- )
    0 echo !      \ disable display echo from terminal
    '[' serial2-type is type
    ;

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

Nous disposons là des mots FORTH nécessaires et suffisants pour paramétrier nos trois transmetteurs LoRa REYAX RYLR890.

Paramétrage des transmetteurs LoRa

Sur cette photo, vous avez des étiquettes BOSS et SLAV1. Ce sont de simples post-it adhésifs collés sur les plaques d'essai.



Nous allons créer trois constantes associées à ces étiquettes:

```
55 constant LoRaBOSS
39 constant LoRaSLAV1
40 constant LoRaSLAV2
```

Pour communiquer entre eux, nos transmetteurs LoRa doivent être paramétrés pour utiliser la même fréquence. La fréquence choisie est de 868,5 Mhz, soit 868500000 Hz:

```
: emptyRX ( -- )
    LoRaINPUT
;

: SETband ( -- )
    emptyRX
    typeToLoRa
    s" 868500000" ATband
    typeToTerm
;
```

Lançons le paramétrage de notre premier transmetteur loRa:

```
serial2.init
SETband
rx. \ display +OK
```

Si tout va bien, l'exécution de `rx.` affiche **+OK**.

C'est le message restitué par le transmetteur LoRa. Il est possible d'avoir un message d'erreur, comme **+ERR=1**. Réitérez la commande de paramétrage.

La fréquence est indiquée sur 9 chiffres, sans séparateur ou espace. L'unité est le Hz¹².

Le module LoRa REYAX RYLR896 peut exploiter les fréquences de 862 Mhz à 1020 Mhz.

¹² Pour la FRANCE, la bande de fréquence libre va de 863 Mhz à 868,6 Mhz. Source: ARCEP Le "portail bandes libres"

ATTENTION : l'antenne doit être accordée à la fréquence utilisée! L'antenne équipant le module LoRa REYAX est accordée pour les fréquences autour de 868 Mhz. L'utilisation d'une antenne mal accordée diminuera considérablement l'efficacité du module LoRa en émission.

Les modules LoRa émettent en bande étroite. Choisissez une fréquence quelconque parmi les fréquences autorisées dans votre pays.

Détermination de l'adresse des transmetteurs LoRa

Pour être opérationnels, tous les transmetteurs d'un réseau doivent être sur la même fréquence. Quand on veut transmettre un message à un transmetteur en particulier, on doit indiquer l'adresse du transmetteur destinataire. Par exemple, si **BOSS** veut envoyer à message à **SLAV1**, on transmettra le message vers le transmetteur qui a l'adresse 39.

ATTENTION : on ne peut pas avoir deux transmetteurs avec une même adresse sur une même fréquence!

Ici définition du mot permettant de paramétriser l'adresse 55 pour le transmetteur **BOSS** :

```
: SETaddress ( n -- )
    emptyRX
    typeToLoRa
    str ATband
    typeToTerm
;

LoRaBOSS SETaddress
rx. \ display +OK
```

On peut prendre une valeur quelconque pour chaque transmetteur, dans l'intervalle [1..65535]. L'adresse 0 est réservée aux transmissions vers tous les transmetteurs LoRa à l'écoute sur la même fréquence.

Maintenant que notre transmetteur **BOSS** est paramétré, on peut le déconnecter du PC et brancher celui étiqueté **SLAV1**. On compile le script source et on lance le paramétrage de **SLAV1**:

```
serial2.init
SETband
rx. \ display +OK

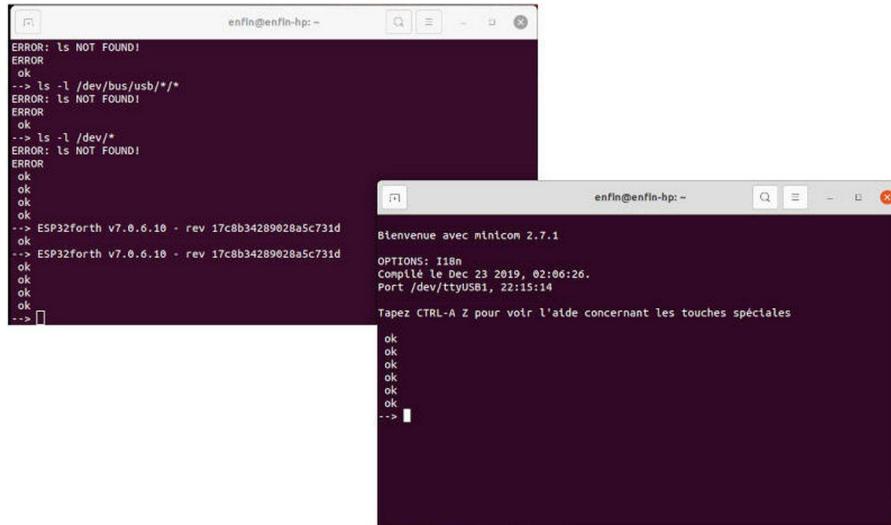
LoRaSLAV1 SETaddress
rx. \ display +OK
```

Communication entre deux transmetteurs LoRa REYAX RYLR890

Pour initialiser nos transmetteurs LoRa REYAX RYLR890, il faut:

- disposer de deux transmetteurs LoRa REYAX RYLR890 avec la carte ESP32
- connecter chaque carte ESP32 sur un port USB disponible sur votre PC

Ici, sous Linux, nous avons ouvert deux terminaux Minicom :



Commandes Linux pour ouvrir ces deux terminaux:

- depuis le clavier, lancer le terminal de commandes par CTRL-ALT-T
- dans la fenêtre de commande, taper sudo minicom lance le terminal minicom connecté à /dev/ttyUSB0
- dans l'invite, tapez le mot de passe administrateur Linux. Le premier terminal donne accès à votre première carte ESP32
- à nouveau, depuis le clavier, lancer le terminal de commandes par CTRL-ALT-T
- dans cette nouvelle fenêtre de commande, taper sudo minicom -D /dev/ttyUSB1 ce qui lance un autre terminal minicom connecté à /dev/ttyUSB1
- dans l'invite, tapez le mot de passe administrateur Linux. Cet autre terminal donne accès à la seconde carte ESP32

Transmission depuis BOSS vers SLAV2

Le listing de notre code FORTH n'est que très peu différent de celui du précédent chapitre. On a simplement retiré les mots **ATaddress** et **ATband**. Ces mots ne sont plus nécessaires pour paramétriser nos transmetteurs LoRa **BOSS**, **SLAV1** et **SLAV2**.

Une fois paramétré, un transmetteur LoRa conserve ce paramétrage, même hors tension. La mise sous tension d'une carte ESP32 et son transmetteur LoRa ne change pas le paramétrage du transmetteur LoRa.

Les mots **ATaddress** et **ATband** sont remplacés par **Atsend** :

```
\ SEND Send data to the appointment address
: Atsend { addr len address -- }
  ." AT+SEND="
  address n. [char] , emit
  len      n. [char] , emit
  addr len type crlf
;
```

Maintenant, on intègre ce mot dans **toSLAV2**:

```
: toSLAV2 ( addr len -- )
  emptyRX
  typeToLoRa
  LoRaSLAV2 Atsend
  typeToTerm
;
```



On compile le même programme sur chaque ESP32 (**BOSS** et **SLAV2**). C'est très facile. Il suffit de copier depuis le listing et de coller dans le terminal. Chaque carte ESP32 va compiler son listing en une dizaine de secondes.

Ici, les deux fenêtres de notre terminal minicom. La fenêtre de gauche permet de contrôler **BOSS**.

La fenêtre de droite contrôle **SLAV2** :

```

enfin@enfin-hp: ~
ok  LoRaSLAV2 ATsend
ok  typeToTerm
ok ;
ok
ok
ok
ok serial2.init
AT+SEND=39,27,this is a transmission testV1
ok
ok
ok 100 ms
+ERR=1.
ok
ok serial2.init
ok s" this is a transmission test" toSLAV2
ok
ok 100 ms
ok rx.
ok
+OK rx.
ok
--> □

```

lable Serial2.end
Serial.available Serial

```

ok->
ok->
ok-> serial2.available
+RCV=55,27,this is a transmission test,-36,40
ok
0 47 --> □

```

Testons la transmission de **BOSS** vers **SLAV2**. Pour ceci, on pose le pointeur de souris dans la fenêtre de gauche et on tape directement :

```

serial2.init
s" this is a transmission test" toSLAV2
rx. \ display: +OK

```

Est-on certain que **SLAV2** a reçu le message? Rien de plus facile.

On pose le curseur de souris dans la fenêtre de droite et on tape simplement :

```

serial2.init
rx. \ display: +RCV=55,27,this is a transmission test,-36,40

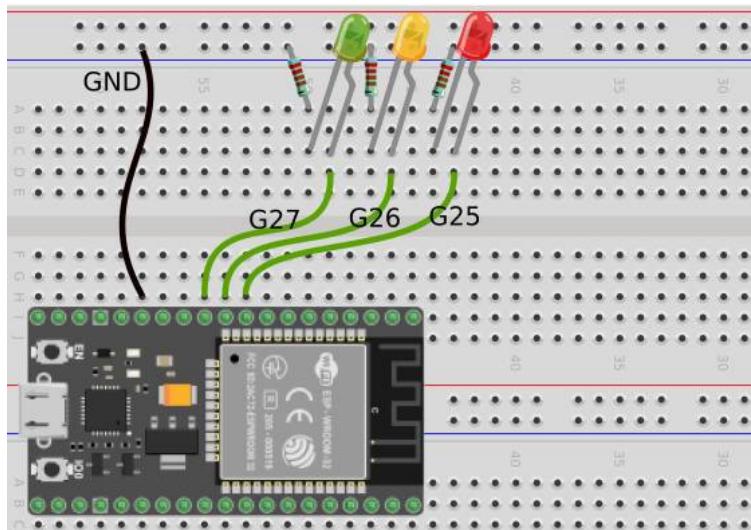
```

Le résultat de cette transmission par **SLAV2** est stocké dans la variable alphanumérique **LoRaRX**.

Interfacer une transmission LoRa avec ESP32Forth

Pour démontrer l'incroyable souplesse du langage FORTH, et plus particulièrement la version ESP32Forth sur ESP32, on va reprendre le programme utilisé dans le chapitre *Gérer un feu tricolore avec ESP32*.

L'ennui, avec les définitions dans ce chapitre, c'est que le contrôle des LEDs exploite les



bornes GPIO de la liaison série. On déplace donc le branchement des LEDs comme ceci :

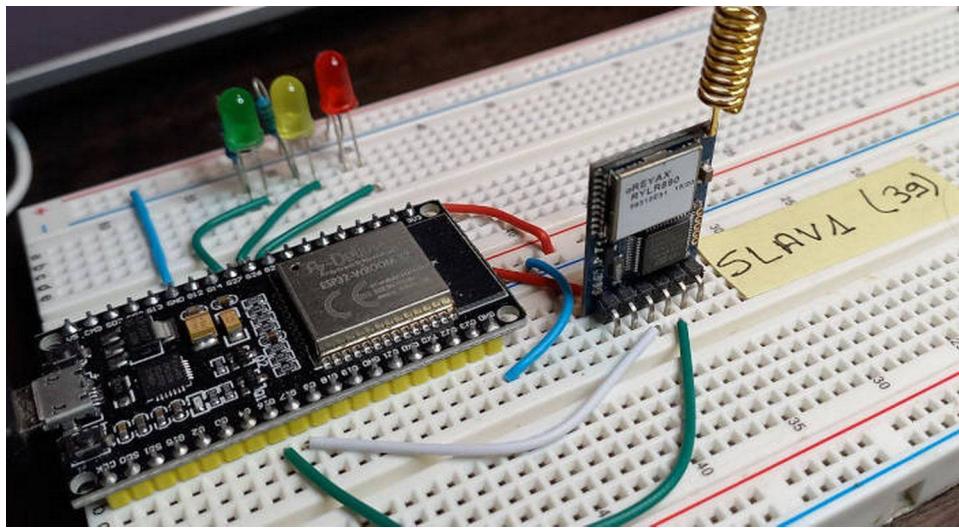
Voici la seule et unique adaptation de code qui est réalisée pour s'adapter au nouveau branchement des LEDs :

```
\ new code
27 constant ledGREEN          \ green LED on GPIO2
26 constant ledYELLOW         \ yellow LED on GPIO21
25 constant ledRED            \ red LED on GPIO17
```

Voici les séquences de code en langage FORTH pour activer ou désactiver sélectivement chaque LED. Ces séquences sont exécutables depuis la fenêtre du terminal connecté à la carte ESP32 :

```
LEDinit
ledGREEN high      \ set GREEN led on
ledRED high       \ set RED led on
ledGREEN low       \ set GREEN led off
```

Et ce sont ces séquences, et elles seules qui seront transmises et reçues par les transmetteurs LoRa. Voici le montage des LEDs et du transmetteur LoRa sur notre plaque d'essai nommée **SLAV1** :



Le programme côté transmetteur LoRa nommé BOSS

On va compléter le programme chargé des transmissions. On part de ce qui est décrit dans le précédent chapitre.

Le listing reprend les composants essentiels permettant une transmission LoRa depuis la carte ESP32 marquée BOSS.

On rajoute simplement quelques définitions simples pour exécuter à distance l'allumage et l'extinction des LEDs qui elles sont sur la carte marquée **SLAV1** :

```
\ 55 constant LoRaBOSS
39 constant LoRaSLAV1
\ 40 constant LoRaSLAV2

: toSLAV1 ( addr len -- )
  emptyRX
  typeToLoRa
  LoRaSLAV1 ATsend
  typeToTerm
;

: REDhigh ( -- )
  s" LEDred high"      toSLAV1
;

: REDlow ( -- )
  s" LEDred low"       toSLAV1
;

: YELLOWhigh ( -- )
  s" ledYELLOW high"   toSLAV1
;

: YELLOWlow ( -- )
```

```

    s" ledYELLOW low"      toSLAV1
;

: GREENhigh ( -- )
    s" ledGREEN high"    toSLAV1
;

: GREENlow ( -- )
    s" ledGREEN low"     toSLAV1
;

```

On crée une définition par commande, ce dans le but de faire au plus simple. Libre à vous de réaliser un moyen plus interactif. Ce n'est pas le but de ce chapitre. Pour le moment, une fois la carte marquée **BOSS** branchée et le code compilé, si on veut transmettre une commande à **SLAV1**, on tape simplement dans le terminal :

```

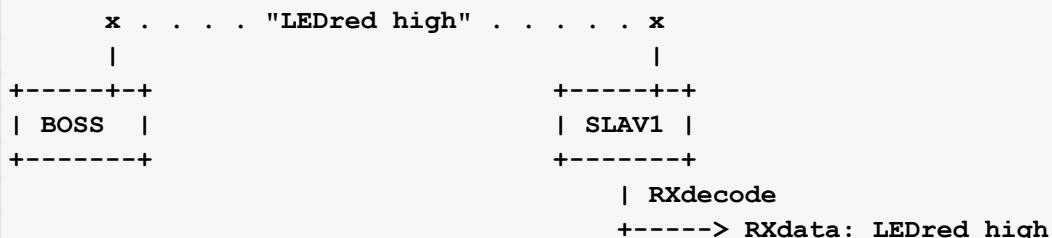
serial2.init
REDhigh

```

Ceci transmet le message **LEDred high** vers **SLAV1**. La dernière phase sera de faire exécuter cette commande comme si elle a été tapée depuis un terminal connecté à **SLAV1**.

Réception et exécution des commandes FORTH par SLAV1

Résumons : le transmetteur BOSS envoie un message, par exemple **LEDred high** à destination du transmetteur SLAV1. Le transmetteur SLAV1 reçoit le message et exécute **RXdecode** pour stocker cette commande FORTH dans la variable alphanumérique **RCVdata**.



Exécution d'une commande reçue par LoRa

Dans notre schéma, nous avons deux cartes ESP32, chacune disposant d'un transmetteur LoRa :

- **BOSS** (adresse 55) qui transmet des commandes FORTH
- **SLAV1** (adresse 39) qui reçoit ces commandes FORTH

La seule différence, avec des commandes FORTH entrées directement au clavier du PC et transmises par le programme terminal connecté à SLAV1 concerne les commandes transmises par LoRa et stockées dans **RXdata**.

Comment exécuter ces commandes stockées dans **Rxdata**? La réponse est consternante de simplicité :

```
RCVdata evaluate
```

Nous n'avons absolument besoin de rien d'autre POUR INTERFACER la transmission LoRa avec n'importe quel programme embarqué dans une carte ESP32!!!

Voici une définition sécurisée de cette interfaçage :

```
: RXinterface ( -- )
    RCVdata ?dup if
        evaluate
    else
        2drop
    then
;
```

Voici quelques manipulations en FORTH pour tester cet interface :

```
LEDinit          \ initialize GPIOs
s" LEDred high" RCVdata $!
RCVdata RXinterface \ turn RED led on
s" LEDred low" RCVdata $!
RCVdata RXinterface \ turn RED led off
```

Nous avons tenu notre promesse : agir depuis LoRa sur n'importe quel programme sans en changer une seule ligne de code.

Dans n'importe quelle carte ESP32, vous pourrez compiler et tester facilement avec le terminal toutes les fonctionnalités de vos programmes.

Pour interfaçer ces programmes, il suffira ensuite de rajouter la couche de transmission LoRa et son code d'interfaçage.

Le transmetteur distant n'aura plus qu'à envoyer des commandes FORTH pour agir sur vos programmes.

Seul le langage FORTH permet un interface transmission -> application aussi simple!

Voyons maintenant le dernier point: lire régulièrement le tampon de réception du transmetteur LoRa....

Boucle de gestion des transmissions LoRa

Le transmetteur LoRa est connecté au port série UART2. Quand une transmission est reçue, le mot **Serial2.available** indique le nombre d'octets en attente dans le tampon série de UART2. S'il n'y a pas de transmission, la valeur remontée par **Serial2.available** sera nulle. Voici le code pour tester la présence de caractères reçus par UART2 :

```
Serial
\ final loop
```

```

: LoRaLoop ( -- )
begin
    Serial2.available \ not 0 if chars available
    if
        100 ms          \ ensures that entire transmission is received
        LoRaRX maxlen$ nip
        Serial2.readBytes
        LoRaRX drop cell - !
        RXdecode         \ analyse content of LoRa message
        RXinterface      \ interpret content or RCVdata
    then
        pause           \ skip to next task
    again
;

```

Le code de **LoRaLoop** utilise une boucle infinie. Il est déconseillé d'exécuter ce mot tel quel. Si vous le faites, vous n'aurez plus la main sur l'interpréteur FORTH de ESP32Forth.

Pour utiliser **LoRaLoop** sans bloquer l'interpréteur FORTH, nous allons définir une nouvelle tâche **my-loop** comme ceci:

```
' LoRaLoop 100 100 task my-loop
my-loop start-task
```

A partir de ce moment, toute transmission effectuée depuis la carte marquée **BOSS** sera interprétée sur cette carte marquée **SLAV1**.

Pour que toute notre programmation reste persistante dans notre carte **SLAV1**, on finalise l'initialisation générale:

```
\ 115200 speed communication for LoRa REYAX
115200 value #SERIAL2_RATE

Serial
: mainInit ( -- )
    cr ." Starting SLAV1 LoRa" cr
    LEDinit
    #SERIAL2_RATE Serial2.begin      \ initialise Serial2
    my-loop start-task
;
startup: mainInit
```

A compter de cet instant, une fois le programme compilé dans la carte ESP32 marquée **SLAV1**, au redémarrage de la carte, le mot **mainInit** sera exécuté, ce que confirme le message **Starting SLAV1 LoRa** qui doit normalement s'afficher.

Voici en photo les actions exécutées depuis la carte BOSS, incrustation en bas à gauche :



Sur la carte marquée **SLAV1**, Les LEDs réagissent avec une latence de une à deux secondes. Ce délai est normal. Il résulte du protocole LoRa qui est certes lent, mais extrêmement robuste. Sur la photo ci-dessus, les tests ont été effectués avec une distance de un mètre. Les cartes **BOSS** et **SLAV1** ont été rapprochées pour la photo.

Si vous laissez **SLAV1** connecté au terminal, vous aurez toujours la main sur l'interpréteur FORTH. Là aussi, c'est normal! Le mot **LoRaLoop** s'exécute en multi-tâche.

Depuis ses origines, le langage FORTH est multi-tâche. Il l'était déjà sur des versions sous MS-DOS quand MS-DOS n'était pas multi-tâche.

Avec ESP32Forth, nous restons dans la continuité des fonctionnalités de FORTH, dont justement les possibilités d'activer des tâches concurrentes. Dans notre cas précis, la tâche moniteur vous laisse la main sur l'interpréteur tout en gérant les LEDs depuis la tâche **LoRaLoop**.

Un interface WEB simple pour ESP32Forth

Auteur : Václav POSELT

J'ai redémarré l'utilisation de Forth après des années sans programmation avec FlashFORTH sur Atmega 328 et Arduino. Après avoir créé ma première construction, il était nécessaire de construire un panneau de commande pour l'électronique, des boutons, un affichage également. J'ai pensé que l'heure était venue de l'IoT et du contrôle sans fil, alors mieux vaut épargner les travaux de construction et contrôler tout ce qui est sans fil. Pour cela je suis passé à ESP32 avec WiFi et BT, j'ai trouvé des dizaines d'exemples de programmes d'interfaces web en Arduino C avec JavaScript, mais rien en ESP32Forth sur ESP32. Pour moi, en tant que débutant, c'était un problème.

Voici donc le résultat de mes efforts - un exemple simple d'interface Web, sur un serveur Web fonctionnant sur ESP32Forth. Le code est basé sur l'exemple de Peter Forth (peter-webpage-dht11-graphic-example.txt). Le code entier se trouve dans le fichier joint **example_web.fs**.

Le serveur Web fonctionne sur la carte ESP32 avec une connexion WiFi activée et répond aux demandes des clients (navigateur sur PC, mobile, etc.).

L'interface Web de base est donc simple :

```
: runpage begin handleClient if serve-page 100 ms then 500 ms again ;
```

où **handleClient** détecte s'il y a des demandes du client, résout la demande et donne du contenu HTML au client avec la page de service Word. La temporisation **ms** permet l'amélioration de la stabilité de la connexion wifi dans mon réseau domestique.

```
: serve-page ( -- ) \ simple parsing and action of client respond
  path s" /" str= if
    htmlpagesend exit \ exit leaves from serve-page
  then
  path s" /26/on" str= if
    cr ." ACTION for /26/on " cr \ here put action word
    0 to GPIO26 htmlpagesend exit
  then
  path s" /26/off" str= if
    cr ." ACTION for /26/off " cr
    1 to GPIO26 htmlpagesend exit
  then
  path s" /27/on" str= if
    cr ." ACTION for /27/on " cr
    0 to GPIO27 htmlpagesend exit
  then
  path s" /27/off" str= if
    cr ." ACTION for /27/off " cr
```

```

    1 to GPIO27 htmlpagesend exit
then
path respond      \ actions for html forms
htmlpagesend exit \ resend html page
;

```

Le mot **serve-page** utilise le texte de la demande du client à partir du chemin de mot sous la forme addr len et le compare avec les réponses possibles du client, chaque correspondance active l'action pertinente et actualise le contenu de la page HTML avec le mot **htmlpagesend**. Les mots d'action peuvent être mis à la place des substituts tels quels . " ACTION for /26/on ".

```

: htmlpagesend \ send whole html page
  s" text/html" ok-response
  htmlpage   \ create html page in webintstream buffer
  webcontent send \ and send it to client
;

```

Le mot **htmlpagesend** renvoie au client (navigateur) le premier code d'état et le type de données HTML. Ensuite, le code de la page HTML est créé dynamiquement sous forme de texte et finalement envoyé au client pour l'afficher dans le navigateur.

En résumé, c'est tout un processus.

Pour une utilisation pratique, je me suis concentré sur trois types d'informations générées par l'interface Web ESP32 :

- données textuelles passives simples telles que les résultats de certaines mesures, par exemple provenant d'une station météorologique
- boutons pour la commutation marche/arrêt pour contrôler quelque chose par le circuit ESP32
- Formulaires HTML pour ajuster certains paramètres dans un programme exécuté sur ESP32.

Pour cela j'ai créé cet exemple simple de page web générée sur ESP32 avec l'adresse IP 92.168.1.6.

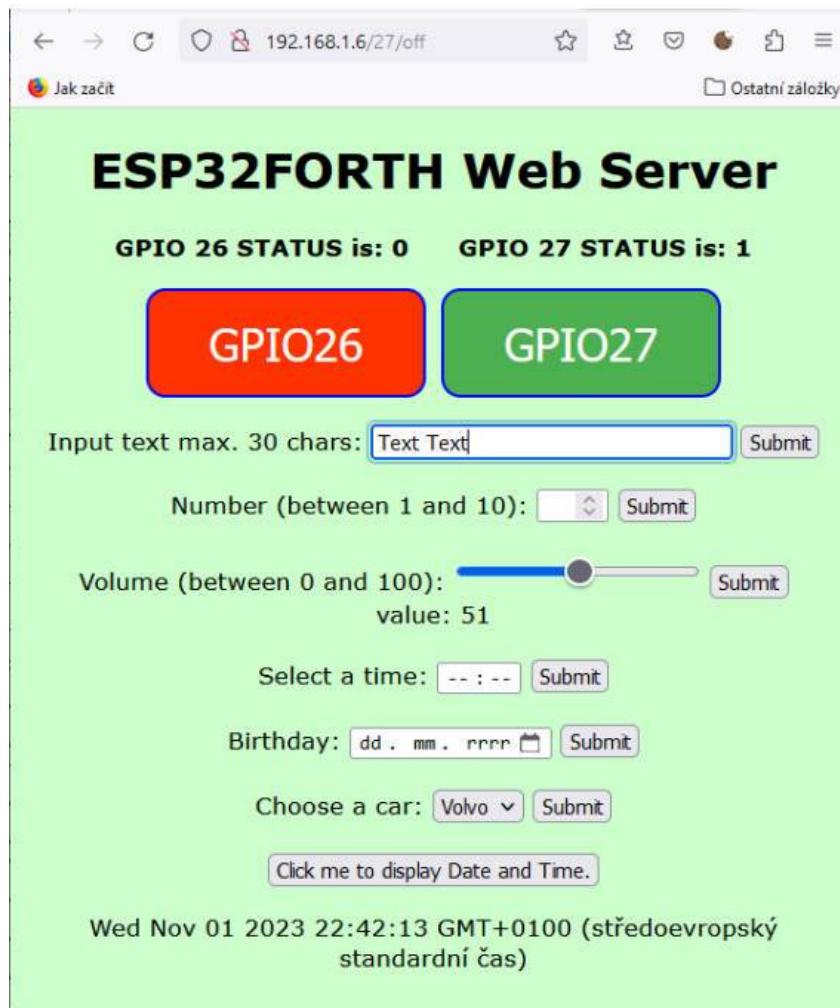


Figure 30: page web générée par ESP32forth

Ainsi, l'état du texte GPIO 26 est : une information textuelle et la valeur 0 ou 1 est la valeur FORTH pour GPIO26 incluse dans la page Web lors de la génération de la page HTML.

Les boutons GPIO26 et GPIO27 peuvent changer la valeur appropriée pour contrôler quelque chose, par exemple, un relais connecté aux GPIO ESP32.

Les autres formulaires HTML peuvent contrôler un ajustement plus avancé des paramètres du programme.

Le dernier *Click me to display...* génère uniquement des informations de date/heure réelles du navigateur client sans aucune connexion de programme au code ESP32forth.

Vous trouverez le script **example_web.fs** dans le fichier **ESP32forth-book.zip** disponible ici :

https://github.com/MPETREMANN11/ESP32forth/blob/main/_documentation/ESP32forth-book.zip

Ensuite seulement en bref :

Les lignes 8 à 29 créent le mot d'aide **mvbar**, utilisé comme **mvbar** pour tout texte multiligne | pour créer une chaîne temporaire comme addr len sur plus de lignes de texte.

Le tampon pour le texte de la page HTML est créé à la ligne 31 par **stream webintstream** et utilise **word >stream** pour ajouter des parties de texte ensemble.

Le mot **htmlpage** sur les lignes 46 à 135 crée dynamiquement du texte html après chaque activation. Les lignes 67 à 71 créent un texte avec les valeurs réelles des valeurs FORTH GPIO26, GPIO27. S'il est utilisé pour afficher en continu certaines valeurs mesurées, il est nécessaire de mettre du code pour l'actualisation automatique de la page HTML dans le code HTML généré.

Les lignes 72 à 88 créent des boutons de couleur rouge ou verte en fonction des valeurs GPIO avec les informations client /26/on ou /26/off pour la détection dans le mot de la page de service.

Les lignes 91 à 127 génèrent des données de formulaires HTML pour ajuster certaines valeurs telles que les données, l'heure, le texte ou la plage. Les lignes 128 à 131 génèrent uniquement des informations de date/heure réelles avec du code JavaScript.

À la fin du code, il y a l'activation du serveur avec wifi et le démarrage de l'interface Web en tant que tâche en arrière-plan.

Je présente ce code comme base d'expérimentations. Je suis sûr qu'il est possible de l'améliorer, les commentaires sont les bienvenus.

Contenu détaillé des vocabulaires ESP32forth

ESP32forth met à disposition de nombreux vocabulaires :

- **FORTH** est le principal vocabulaire ;
- certains vocabulaires servent à la mécanique interne pour ESP32Forth, comme **internals, asm...**
- beaucoup de vocabulaires permettent la gestion de ports ou accessoires spécifiques, comme **bluetooth, oled, spi, wifi, wire...**

Vous trouverez ici la liste de tous les mots définis dans ces différents vocabulaires.

Certains mots sont présentés avec un lien coloré :

`align` est un mot FORTH ordinaire ;

`CONSTANT` est mot de définition ;

`begin` marque une structure de contrôle ;

`key` est un mot d'exécution différée ;

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

`--registers` marque un vocabulaire.

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

Version v 7.0.7.15

FORTH

<code>-</code>	<code>-rot</code>	<code>-</code>	<code>:</code>	<code>:</code>	<code>:noname</code>	<code>!</code>
<code>?</code>	<code>?do</code>	<code>?dup</code>	<code>."</code>	<code>.s</code>	<code>[IF]</code>	<code>[THEN]</code>
<code>(local)</code>	<code>I</code>	<code>I'1</code>	<code>[char]</code>	<code>[ELSE]</code>	<code>*</code>	<code>*/</code>
<code>1</code>	<code>1</code>	<code>1</code>	<code>}transfer</code>	<code>@</code>	<code>#></code>	<code>#fs</code>
<code>* / MOD</code>	<code>/</code>	<code>/mod</code>	<code>#</code>	<code>#!</code>	<code>+to</code>	<code><</code>
<code>#s</code>	<code>#tib</code>	<code>±</code>	<code>+!</code>	<code>+loop</code>	<code>>BODY</code>	<code>>params</code>
<code><#</code>	<code><=</code>	<code>⇒</code>	<code>=</code>	<code>≥</code>	<code>>name</code>	<code>1/F</code>
<code>>flags</code>	<code>>flags&</code>	<code>>in</code>	<code>>link</code>	<code>>link&</code>	<code>1-</code>	<code>2drop</code>
<code>>R</code>	<code>>size</code>	<code>0<</code>	<code>0<></code>	<code>0=</code>	<code>2/</code>	<code>2dup</code>
<code>1+</code>	<code>2!</code>	<code>2@</code>	<code>2*</code>	<code>abs</code>	<code>accept</code>	<code>adc</code>
<code>4*</code>	<code>4/</code>	<code>abort</code>	<code>abort"</code>	<code>align</code>	<code>aligned</code>	<code>allocate</code>
<code>afliteral</code>	<code>aft</code>	<code>again</code>	<code>ahead</code>	<code>ansi</code>	<code>ARSHIFT</code>	<code>asm</code>
<code>allot</code>	<code>also</code>	<code>analogRead</code>	<code>AND</code>	<code>bq</code>	<code>BIN</code>	<code>binary</code>
<code>assert</code>	<code>at-xy</code>	<code>base</code>	<code>begin</code>	<code>cat</code>	<code>catch</code>	<code>CELL</code>
<code>bl</code>	<code>blank</code>	<code>block</code>	<code>block-fid</code>	<code>block-id</code>	<code>buffer</code>	<code>bye</code>
<code>c,</code>	<code>C!</code>	<code>C@</code>	<code>CASE</code>	<code>CLOSE-DIR</code>	<code>CLOSE-FILE</code>	<code>cmove</code>
<code>cell/</code>	<code>cell+</code>	<code>cells</code>	<code>char</code>			

cmove>	CONSTANT	context	copy	cp	cr	CREATE
CREATE-FILE	current	dacWrite	decimal	default-key	default-key?	
default-type		default-use	defer	DEFINED?	definitions	DELETE-FILE
depth	digitalRead	digitalWrite		do	DOES>	DROP
dump	dump-file	DUP	duty	echo	editor	else
emit	empty-buffers		ENDCASE	ENDOF	erase	ESP
ESP32-C3?	ESP32-S2?	ESP32-S3?	ESP32?	evaluate	EXECUTE	exit
extract	F-	f.	f.s	F*	F**	F/
F+ F>S	F< F0<	F<=	F<>	F=	F>	F>=
fdepth	FDROP	FDUP	FEXP	fg	file-exists?	
FILE-POSITION		FILE-SIZE	fill	FIND	fliteral	FLN
FLOOR	flush	FLUSH-FILE	FMAX	FMIN	FNEGATE	FNIP
for	forget	FORTH	forth-builtins	FOVER		FP!
FP0	fp0	free	freq	FROT	FSIN	FSINCOS
FSQRT	FSWAP	fvariable	handler	here	hex	HIGH
hld	hold	htpd	I	if	IMMEDIATE	include
included	included?	INPUT	internals	invert	is	J
K	key	key?	L!	latesttxt	leave	LED
ledc	list	literal	load	login	loop	LOW
ls	LSHIFT	max	MDNS.begin	min	mod	ms
MS-TICKS	mv	n.	needs	negate	nest-depth	next
nip	nl	NON-BLOCK	normal	octal	OF	ok
only	open-blocks	OPEN-DIR	OPEN-FILE	OR	order	OUTPUT
OVER	pad	page	PARSE	pause	PI	pin
pinMode	postpone	precision	previous	prompt	PSRAM?	pulseIn
quit	r"	R@	R/O	R/W	R>	rl
r~	rdrop	read-dir	READ-FILE	recurse	refill	registers
remaining	remember	RENAME-FILE	repeat	REPOSITION-FILE		required
reset	resize	RESIZE-FILE	restore	revive	RISC-V?	rm
rot	RP!	RP@	rp0	RSHIFT	rtos	s"
S>F	s>z	save	save-buffers	scr		SD
SD-MMC	sealed	see	Serial	set-precision		set-title
sf,	SF!	SF@	SFLOAT	SFLOAT+	SFLOATS	sign
SL@	sockets	SP!	SP@	sp0	space	spaces
SPIFFS	start-task	startswith?	startup:	state	str	str=
streams	structures	SW@	SWAP	task	tasks	telnetd
terminate	then	throw	thru	tib	to	tone
touch	transfer	transfer	type	u.	U/MOD	UL@
UNLOOP	until	update	use	used	UW@	value
VARIABLE	visual	vlist	vocabulary	W!	W/O	web-
interface						
webui	while	WiFi	Wire	words	WRITE-FILE	XOR
Xtensa?	z"	z>s				

asm

```

xtensa disasm disasml matchit address istep sextend m. m@ for-ops op >operands
>mask >pattern >length >xt op-snap opcodes coden, names operand 1 o bits
bit skip advance advance-operand reset reset-operand for-operands operands
>printop >inop >next >opmask& bit! mask pattern length demask enmask >>1
odd? high-bit end-code code, code4, code3, code2, code1, callot chere reserve
code-at code-start

```

bluetooth

```
SerialBT.new SerialBT.delete SerialBT.begin SerialBT.end SerialBT.available  
SerialBT.readBytes SerialBT.write SerialBT.flush SerialBT.hasClient  
SerialBT.enableSSP SerialBT.setPin SerialBT.unpairDevice SerialBT.connect  
SerialBT.connectAddr SerialBT.disconnect SerialBT.connected  
SerialBT.isReady bluetooth-builtins
```

editor

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

ESP

```
getHeapSize getFreeHeap getMaxAllocHeap getChipModel getChipCores getFlashChipSize  
getCpuFreqMHz getSketchSize deepSleep getEfuseMac esp_log_level_set ESP-builtins
```

httpd

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

insides

```
run normal-mode raw-mode step ground handle-key quit-edit save load backspace  
delete handle-esc insert update crtype cremit ndown down nup up caret length  
capacity text start-size fileh filename# filename max-path
```

internals

```
assembler-source xtensa-assembler-source MALLOC SYSFREE REALLOC heap_caps_malloc  
heap_caps_free heap_caps_realloc heap_caps_get_total_size heap_caps_get_free_size  
heap_caps_get_minimum_free_size heap_caps_get_largest_free_block RAW-YIELD  
RAW-TERMINATE READDR DIRCODE CALL0 CALL1 CALL2 CALL3 CALL4 CALL5 CALL6  
CALL7 CALL8 CALL9 CALL10 CALL11 CALL12 CALL13 CALL14 CALL15 DOFLIT S>FLOAT?  
fill32 'heap 'context 'latestxt 'notfound 'heap-start 'heap-size 'stack-cells  
'boot 'boot-size 'tib 'argc 'argv 'runner 'throw-handler NOP BRANCH OBRANCH  
DONEEXT DOLIT DOSET DOCOL DOCON DOVAR DOCREATE DODOES ALITERAL LONG-SIZE  
S>NUMBER? 'SYS YIELD EVALUATE1 'builtins internals-builtins autoexec  
arduino-remember-filename  
arduino-default-use esp32-stats serial-key? serial-key serial-type yield-task  
yield-step e' @line grow-blocks use?! common-default-use block-data block-dirty  
clobber clobber-line include+ path-join included-files raw-included include-file  
sourcedirname sourcefilename! sourcefilename sourcefilename# sourcefilename&  
starts../ starts./ dirname ends/ default-remember-filename remember-filename  
restore-name save-name forth-wordlist setup-saving-base 'cold park-forth  
park-heap saving-base crtype cremit cases (+to) (to) --? }? ?room scope-create  
do-local scope-clear scope-exit local-op scope-depth local+! local! local@  
<>locals locals-here locals-area locals-gap locals-capacity ?ins. ins.  
vins. onlines line-pos line-width size-all size-vocabulary vocs. voc. voclist
```

```

voclist-from see-all >vocnext see-vocabulary nonvoc? see-xt ?see-flags
see-loop see-one indent+! icr see. indent mem= ARGS _MARK -TAB +TAB NONAMED
BUILTIN_FORK SMUDGE IMMEDIATE_MARK relinquish dump-line ca@ cell-shift
cell-base cell-mask MALLOC_CAP_RTCRAM MALLOC_CAP_RETENTION MALLOC_CAP_IRAM_8BIT
MALLOC_CAP_DEFAULT MALLOC_CAP_INTERNAL MALLOC_CAP_SPIRAM MALLOC_CAP_DMA
MALLOC_CAP_8BIT MALLOC_CAP_32BIT MALLOC_CAP_EXEC #f+s internalized BUILTIN_MARK
zplace $place free. boot-prompt raw-ok [SKIP] [SKIP] ?stack sp-limit input-limit
tib-setup raw.s $@ digit parse-quote leaving, leaving )leaving leaving(
value-bind evaluate&fill evaluate-buffer arrow ?arrow. ?echo input-buffer
immediate? eat-till-cr wascr *emit *key notfound last-vocabulary voc-stack-end
xt-transfer xt-hide xt-find& scope

```

interrupts

```

pinchange #GPIO_INTR_HIGH_LEVEL #GPIO_INTR_LOW_LEVEL #GPIO_INTR_ANYEDGE
#GPIO_INTR_NEDGE #GPIO_INTR_POSEDGE #GPIO_INTR_DISABLE ESP_INTR_FLAG_INTRDISABLED
ESP_INTR_FLAG_IRAM ESP_INTR_FLAG_EDGE ESP_INTR_FLAG_SHARED ESP_INTR_FLAG_NMI
ESP_INTR_FLAG_LEVELn ESP_INTR_FLAG_DEFAULT gpio_config gpio_reset_pin gpio_set_intr_type
gpio_intr_enable gpio_intr_disable gpio_set_level gpio_get_level gpio_set_direction
gpio_set_pull_mode gpio_wakeup_enable gpio_wakeup_disable gpio_pullup_en
gpio_pulldown_en gpio_pulldown_dis gpio_hold_en gpio_hold_dis
gpio_deep_sleep_hold_en gpio_deep_sleep_hold_dis gpio_install_isr_service
gpio_isr_handler_add gpio_isr_handler_remove
gpio_set_drive_capability gpio_get_drive_capability esp_intr_alloc esp_intr_free
interrupts-builtins

```

ledc

```

ledcSetup ledcAttachPin ledcDetachPin ledcRead ledcReadFreq ledcWrite ledcWriteTone
ledcWriteNote ledc-builtins

```

oled

```

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

```

registers

```
m@ m!
```

riscv

```

C.FSWSP, C.SWSP, C.FSDSP, C.ADD, C.JALR, C.EBREAK, C.MV, C.JR, C.FLWSP,
C.LWSP, C.FLDSP, C.SLLI, BNEZ, BEQZ, C.J, C.ADDW, C.SUBW, C.AND, C.OR,
C.XOR, C.SUB, C.ANDI, C.SRAI, C.SRLI, C.LUI, C.LI, C.JAL, C.ADDI, C.NOP,
C.FSW, C.SW, C.FSD, C.FLW, C.FLD, C.ADDI4SP, C.ILL, EBREAK, ECALL,
AND, OR, SRA, SRL, XOR, SLTU, SLT, SLL, SUB, ADD, SRAI, SRLI, SLLI, ANDI,
ORI, XORI, SLTIU, SLTI, ADDI, SW, SH, SB, LHU, LBU, LW, LH, LB, BGEU, BLTU,
BGE, BLT, BNE, BEO, JALR, JAL, AUIPC, LUI, J-TYPE U-TYPE B-TYPE S-TYPE
I-TYPE R-TYPE rs2' rs2#' rs2 rs2# rs1' rs1#' rs1 rs1# rd' rd#' rd rd# offset
ofs ofs. >ofs iiiii i numeric register' reg'. reg>reg' register reg. nop
x31 x30 x29 x28 x27 x26 x25 x24 x23 x22 x21 x20 x19 x18 x17 x16 x15 x14
x13 x12 x11 x10 x9 x8 x7 x6 x5 x4 x3 x2 x1 zero

```

rtos

```
vTaskDelete xTaskCreatePinnedToCore xPortGetCoreID rtos-builtins
```

SD

```
SD.begin SD.beginFull SD.beginDefaults SD.end SD.cardType SD.totalBytes  
SD.usedBytes SD-builtins
```

SD_MMC

```
SD\_MMC.begin SD_MMC.beginFull SD_MMC.beginDefaults SD_MMC.end SD_MMC.cardType  
SD_MMC.totalBytes SD\_MMC.usedBytes SD_MMC-builtins
```

Serial

```
Serial.begin Serial.end Serial.available Serial.readBytes Serial.write  
Serial.flush Serial.setDebugOutput Serial2.begin Serial2.end Serial2.available  
Serial2.readBytes Serial2.write Serial2.flush Serial2.setDebugOutput serial-  
builtins
```

Sockets

```
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 gethostname errno sockets-builtins
```

spi

```
SPI.begin SPI.end SPI.setHwCs SPI.setBitOrder SPI.setDataMode SPI.setFrequency  
SPI.setClockDivider SPI.getClockDivider SPI.transfer SPI.transfer8 SPI.transfer16  
SPI.transfer32 SPI.transferBytes SPI.transferBits SPI.write SPI.write16  
SPI.write32 SPI.writeBytes SPI.writePixels SPI.writePattern SPI-builtins
```

SPIFFS

```
SPIFFS.begin SPIFFS.end SPIFFS.format SPIFFS.totalBytes SPIFFS.usedBytes  
SPIFFS-builtins
```

streams

```
stream> <stream stream>ch ch>stream wait-read wait-write empty? full? stream#  
>offset >read >write stream
```

structures

```
field struct-align align-by last-struct struct long ptr i64 i32 i16 i8  
type last-align
```

tasks

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

telnetd

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

visual

```
edit insides
```

web-interface

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

WiFi

```
WIFI_MODE_APSTA WIFI_MODE_AP WIFI_MODE_STA WIFI_MODE_NULL WiFi.config WiFi.begin
WiFi.disconnect WiFi.status WiFi.macAddress WiFi.localIP WiFi.mode WiFi.setTxPower
WiFi.getTxPower WiFi.softAP WiFi.softAPIP WiFi.softAPBroadcastIP
WiFi.softAPNetworkID
WiFi.softAPConfig WiFi.softAPdisconnect WiFi.softAPgetStationNum WiFi-builtins
```

Wire

```
Wire.begin Wire.setClock Wire.getClock Wire.setTimeout Wire.getTimeout
Wire.beginTransmission Wire.endTransmission Wire.requestFrom Wire.write
Wire.available Wire.read Wire.peek Wire.flush Wire-builtins
```

xtensa

```
WUR, WSR, WITLB, WER, WDTLB, WAITI, SSXU, SSX, SSR, SSL, SSIU, SSI, SSAI,
SSA8L, SSA8B, SRLI, SRL, SRC, SRAI, SRA, SLLI, SLL, SICW, SICT, SEXT, SDCT,
RUR, RSR, RSIL, RFI, ROTW, RITLB1, RITLB0, RER, RDITLB1, RDITLB0, PITLB,
PDTLB, NSAU, NSA, MULA.DD.HH, MULA.DD.LH, MULA.DD.HL, MULA.DD.LL, MULS.DD
MULA.DA.HH, MULA.DA.LH, MULA.DA.HL, MULA.DA.LL, MULS.DA MULA.AD.HH, MULA.AD.LH,
MULA.AD.HL, MULA.AD.LL, MULS.AD MULA.AA.HH, MULA.AA.LH, MULA.AA.HL, MULA.AA.LL,
MULS.AA MULA.DD.HH.LDINC, MULA.DD.LH.LDINC, MULA.DD.HL.LDINC, MULA.DD.LL.LDINC,
MULA.DD.LDINC MULA.DD.HH.LDDEC, MULA.DD.LH.LDDEC, MULA.DD.HL.LDDEC,
MULA.DD.LL.LDDEC,
MULA.DD.LDDEC MULA.DD.HH, MULA.DD.LH, MULA.DD.HL, MULA.DD.LL, MULA.DD
MULA.DA.HH.LDINC,
MULA.DA.LH.LDINC, MULA.DA.HL.LDINC, MULA.DA.LL.LDINC, MULA.DA.LDINC
MULA.DA.HH.LDDEC,
MULA.DA.LH.LDDEC, MULA.DA.HL.LDDEC, MULA.DA.LL.LDDEC, MULA.DA.LDDEC MULA.DA.HH,
MULA.DA.LH, MULA.DA.HL, MULA.DA.LL, MULA.DA MULA.AD.HH, MULA.AD.LH, MULA.AD.HL,
MULA.AD.LL, MULA.AD MULA.AA.HH, MULA.AA.LH, MULA.AA.HL, MULA.AA.LL, MULA.AA
MUL16U, MUL16S, MUL.DD.HH, MUL.DD.LH, MUL.DD.HL, MUL.DD.LL, MUL.DD MUL.DA.HH,
MUL.DA.LH, MUL.DA.HL, MUL.DA.LL, MUL.DA MUL.AD.HH, MUL.AD.LH, MUL.AD.HL,
MUL.AD.LL, MUL.AD MUL.AA.HH, MUL.AA.LH, MUL.AA.HL, MUL.AA.LL, MUL.AA MOV,
MOV, MOV.S, MOV.S, MOVGEZ.S, MOVLTZ.S, MOVNEZ.S, MOVEQZ.S, ULE.S, OLE.S,
ULT.S, OLT.S, UEQ.S, OEQ.S, UN.S, CMPSOP NEG.S, WFR, RFR, ABS.S, MOV.S,
ALU2.S UTRUNC.S, UFLOAT.S, FLOAT.S, CEIL.S, FLOOR.S, TRUNC.S, ROUND.S,
MSUB.S, MADD.S, MUL.S, SUB.S, ADD.S, ALU.S MOVE, MOVGEZ, MOVLTZ, MOVNEZ,
MOVEQZ, MAXU, MINU, MAX, MIN, CONDOP MOV, LSXU, LSX, L32E, LICW, LICT,
```

```

LDCT, JX, IITLB, IDTLB, LSIU, LSI, LDINC, LDDEC, L32R, EXTUI, S32E, S32RI,
S32C1I, ADDMI, ADDI, L32AI, L16SI, S32I, S16I, S8I, L32I, L16UI, L8UI,
LDSTORE MOVI, IIU, IHU, IPFL, DIWBI, DIWB, DIU, DHU, DPFL, CACHING2 III,
IHI, IPF, DII, DHI, DHWBI, DHWB, DPFWO, DPFRO, DFW, DPFR, CACHING1 CLAMPS,
BREAK, CALLX12, CALLX8, CALLX4, CALLX0, CALLXOP CALL12, CALL8, CALL4, CALL0,
CALLOP LOOPGTZ, LOOPNEZ, LOOP, BT, BF, BRANCH2b J, BGEUI, BGEI, BGEZ, BLTUI,
BLTI, BLTZ, BNEI, BNEZ, ENTRY, BEQI, BEQZ, BRANCH2e BRANCH2a BRANCH2 BBSI,
BBS, BNALL, BGEU, BGE, BNE, BANY, BBCI, BBC, BALL, BLTU, BLT, BEQ, BNONE,
BRANCH1 REMS, REMU, QUOS, QUOU, MULSH, MULUH, MULL, XORB, ORBC, ORB, ANDBC,
ANDB, ALU2 ALL8, ANY8, ALL4, ANY4, ANYALL SUBX8, SUBX4, SUBX2, SUB, ADDX8,
ADDX4, ADDX2, ADD, XOR, OR, AND, ALU XSR, ABS, NEG, RFDO, RFDD, SIMCALL,
SYSCALL, RFWU, RFWO, RFDE, RFUE, RFME, RFE, NOP, EXTW, MEMW, EXCW, DSYNC,
ESYNC, RSYNC, ISYNC, RETW, RET, ILL, ILL.N, NOP.N, RETW.N, RET.N, BREAK.N,
MOV.N, MOVI.N, BNEZ.N, BEQZ.N, ADDI.N, ADD.N, S32I.N, L32I.N, tttt t ssss
s rrrr r bbbb b y w iiiii i xxxx x sa sa. >sa entry12 entry12' entry12.
>entry12 coffset18 cofs cofs. >cofs offset18 offset12 offset8 ofs18 ofs12
ofs8 ofs18. ofs12. ofs8. >ofs sr imm16 imm8 imm4 im numeric register reg.
nop a15 a14 a13 a12 a11 a10 a9 a8 a7 a6 a5 a4 a3 a2 a1 a0

```

Annexe A – Sommaire des registres

GPIO registers

Name	Description	Address	Access
GPIO_OUT_REG	GPIO 0-31 output register	\$3FF44004	R/W
GPIO_OUT_W1TS_REG	GPIO 0-31 output register_W1TS	\$3FF44008	WO
GPIO_OUT_W1TC_REG	GPIO 0-31 output register_W1TC	\$3FF4400C	WO
GPIO_OUT1_REG GPIO	GPIO 32-39 output register	\$3FF44010	R/W
GPIO_OUT1_W1TS_REG	GPIO 32-39 output bit set register	\$3FF44014	WO
GPIO_OUT1_W1TC_REG	GPIO 32-39 output bit clear register	\$3FF44018	WO
GPIO_ENABLE_REG	GPIO 0-31 output enable register	\$3FF44020	R/W
GPIO_ENABLE_W1TS_REG	GPIO 0-31 output enable register_W1TS	\$3FF44024	WO
GPIO_ENABLE_W1TC_REG	GPIO 0-31 output enable register_W1TC	\$3FF44028	WO
GPIO_ENABLE1_REG	GPIO 32-39 output enable register	\$3FF4402C	R/W
GPIO_ENABLE1_W1TS_REG	GPIO 32-39 output enable bit set register	\$3FF44030	WO
GPIO_ENABLE1_W1TC_REG	GPIO 32-39 output enable bit clear register	\$3FF44034	WO
GPIO_STRAP_REG	Bootstrap pin value register	\$3FF44038	RO
GPIO_IN_REG	GPIO 0-31 input register	\$3FF4403C	RO
GPIO_IN1_REG	GPIO 32-39 input register	\$3FF44040	RO
GPIO_STATUS_REG	GPIO 0-31 interrupt status register	\$3FF44044	R/W
GPIO_STATUS_W1TS_REG	GPIO 0-31 interrupt status register_W1TS	\$3FF44048	WO
GPIO_STATUS_W1TC_REG	GPIO 0-31 interrupt status register_W1TC	\$3FF4404C	WO
GPIO_STATUS1_REG	GPIO 32-39 interrupt status register1	\$3FF44050	R/W
GPIO_STATUS1_W1TS_REG	GPIO 32-39 interrupt status bit set register	\$3FF44054	WO
GPIO_STATUS1_W1TC_REG	GPIO 32-39 interrupt status bit clear register	\$3FF44058	WO
GPIO_ACPU_INT_REG	GPIO 0-31 APP_CPU interrupt status	\$3FF44060	RO
GPIO_ACPU_NMI_INT_REG	GPIO 0-31 APP_CPU non-maskable interrupt status	\$3FF44064	RO
GPIO_PCPU_INT_REG	GPIO 0-31 PRO_CPU interrupt status	\$3FF44068	RO
GPIO_PCPU_NMI_INT_REG	GPIO 0-31 PRO_CPU non-maskable interrupt status	\$3FF4406C	RO
GPIO_ACPU_INT1_REG	GPIO 32-39 APP_CPU interrupt status	\$3FF44074	RO
GPIO_ACPU_NMI_INT1_REG	GPIO 32-39 APP_CPU non-maskable interrupt status	\$3FF44078	RO
GPIO_PCPU_INT1_REG	GPIO 32-39 PRO_CPU interrupt status	\$3FF4407C	RO
GPIO_PCPU_NMI_INT1_REG	GPIO 32-39 PRO_CPU non-maskable interrupt status	\$3FF44080	RO
GPIO_PIN0_REG	Configuration for GPIO pin 0	\$3FF44088	R/W
GPIO_PIN1_REG	Configuration for GPIO pin 1	\$3FF4408C	R/W

Name	Description	Address	Access
GPIO_PIN2_REG	Configuration for GPIO pin 2	\$3FF44090	R/W
GPIO_PIN38_REG	Configuration for GPIO pin 38	\$3FF44120	R/W
GPIO_PIN39_REG	Configuration for GPIO pin 39	\$3FF44124	R/W
GPIO_FUNC0_IN_SEL_CFG_REG	Peripheral function 0 input selection register	\$3FF44130	R/W
GPIO_FUNC1_IN_SEL_CFG_REG	Peripheral function 1 input selection register	\$3FF44134	R/W
GPIO_FUNC254_IN_SEL_CFG_REG	Peripheral function 254 input selection register	\$3FF44528	R/W
GPIO_FUNC255_IN_SEL_CFG_REG	Peripheral function 255 input selection register	\$3FF4452C	R/W
GPIO_FUNC0_OUT_SEL_CFG_REG	Peripheral output selection for GPIO 0	\$3FF44530	R/W
GPIO_FUNC1_OUT_SEL_CFG_REG	Peripheral output selection for GPIO 1	\$3FF44534	R/W
GPIO_FUNC38_OUT_SEL_CFG_REG	Peripheral output selection for GPIO 38	\$3FF445C8	R/W
GPIO_FUNC39_OUT_SEL_CFG_REG	Peripheral output selection for GPIO 39	\$3FF445CC	R/W
IO_MUX_PIN_CTRL	Clock output configuration register	\$3FF49000	R/W
IO_MUX_GPIO36_REG	Configuration register for pad GPIO36	\$3FF49004	R/W
IO_MUX_GPIO37_REG	Configuration register for pad GPIO37	\$3FF49008	R/W
IO_MUX_GPIO38_REG	Configuration register for pad GPIO38	\$3FF4900C	R/W
IO_MUX_GPIO39_REG	Configuration register for pad GPIO39	\$3FF49010	R/W
IO_MUX_GPIO34_REG	Configuration register for pad GPIO34	\$3FF49014	R/W
IO_MUX_GPIO35_REG	Configuration register for pad GPIO35	\$3FF49018	R/W
IO_MUX_GPIO32_REG	Configuration register for pad GPIO32	\$3FF4901C	R/W
IO_MUX_GPIO33_REG	Configuration register for pad GPIO33	\$3FF49020	R/W
IO_MUX_GPIO25_REG	Configuration register for pad GPIO25	\$3FF49024	R/W
IO_MUX_GPIO26_REG	Configuration register for pad GPIO26	\$3FF49028	R/W
IO_MUX_GPIO27_REG	Configuration register for pad GPIO27	\$3FF4902C	R/W
IO_MUX_MTMS_REG	Configuration register for pad MTMS	\$3FF49030	R/W
IO_MUX_MTDI_REG	Configuration register for pad MTDI	\$3FF49034	R/W
IO_MUX_MTCK_REG	Configuration register for pad MTCK	\$3FF49038	R/W
IO_MUX_MTDO_REG	Configuration register for pad MTDO	\$3FF4903C	R/W
IO_MUX_GPIO2_REG	Configuration register for pad GPIO2	\$3FF49040	R/W
IO_MUX_GPIO0_REG	Configuration register for pad GPIO0	\$3FF49044	R/W
IO_MUX_GPIO4_REG	Configuration register for pad GPIO4	\$3FF49048	R/W
IO_MUX_GPIO16_REG	Configuration register for pad GPIO16	\$3FF4904C	R/W
IO_MUX_GPIO17_REG	Configuration register for pad GPIO17	\$3FF49050	R/W
IO_MUX_SD_DATA2_REG	Configuration register for pad SD_DATA2	\$3FF49054	R/W
IO_MUX_SD_DATA3_REG	Configuration register for pad SD_DATA3	\$3FF49058	R/W
IO_MUX_SD_CMD_REG	Configuration register for pad SD_CMD	\$3FF4905C	R/W
IO_MUX_SD_CLK_REG	Configuration register for pad SD_CLK	\$3FF49060	R/W
IO_MUX_SD_DATA0_REG	Configuration register for pad SD_DATA0	\$3FF49064	R/W
IO_MUX_SD_DATA1_REG	Configuration register for pad SD_DATA1	\$3FF49068	R/W

Name	Description	Address	Access
IO_MUX_GPIO5_REG	Configuration register for pad GPIO5	\$3FF4906C	R/W
IO_MUX_GPIO18_REG	Configuration register for pad GPIO18	\$3FF49070	R/W
IO_MUX_GPIO19_REG	Configuration register for pad GPIO19	\$3FF49074	R/W
IO_MUX_GPIO20_REG	Configuration register for pad GPIO20	\$3FF49078	R/W
IO_MUX_GPIO21_REG	Configuration register for pad GPIO21	\$3FF4907C	R/W
IO_MUX_GPIO22_REG	Configuration register for pad GPIO22	\$3FF49080	R/W
IO_MUX_U0RXD_REG	Configuration register for pad U0RXD	\$3FF49084	R/W
IO_MUX_U0TXD_REG	Configuration register for pad U0TXD	\$3FF49088	R/W
IO_MUX_GPIO23_REG	Configuration register for pad GPIO23	\$3FF4908C	R/W
IO_MUX_GPIO24_REG	Configuration register for pad GPIO24	\$3FF49090	R/W

GPIO configuration / data registers

RTCIO_RTC_GPIO_OUT_REG	RTC GPIO output register	0x3FF48400	R/W
RTCIO_RTC_GPIO_OUT_W1TS_REG	RTC GPIO output bit set register	0x3FF48404	WO
RTCIO_RTC_GPIO_OUT_W1TC_REG	RTC GPIO output bit clear register	0x3FF48408	WO
RTCIO_RTC_GPIO_ENABLE_REG	RTC GPIO output enable register	0x3FF4840C	R/W
RTCIO_RTC_GPIO_ENABLE_W1TS_REG	RTC GPIO output enable bit set register	0x3FF48410	WO
RTCIO_RTC_GPIO_ENABLE_W1TC_REG	RTC GPIO output enable bit clear register	0x3FF48414	WO
RTCIO_RTC_GPIO_STATUS_REG	RTC GPIO interrupt status register	0x3FF48418	WO
RTCIO_RTC_GPIO_STATUS_W1TS_REG	RTC GPIO interrupt status bit set register	0x3FF4841C	WO
RTCIO_RTC_GPIO_STATUS_W1TC_REG	RTC GPIO interrupt status bit clear register	0x3FF48420	WO
RTCIO_RTC_GPIO_IN_REG	RTC GPIO input register	0x3FF48424	RO
RTCIO_RTC_GPIO_PIN0_REG	RTC configuration for pin 0	0x3FF48428	R/W
RTCIO_RTC_GPIO_PIN1_REG	RTC configuration for pin 1	0x3FF4842C	R/W
RTCIO_RTC_GPIO_PIN2_REG	RTC configuration for pin 2	0x3FF48430	R/W
RTCIO_RTC_GPIO_PIN3_REG	RTC configuration for pin 3	0x3FF48434	R/W
RTCIO_RTC_GPIO_PIN4_REG	RTC configuration for pin 4	0x3FF48438	R/W
RTCIO_RTC_GPIO_PIN5_REG	RTC configuration for pin 5	0x3FF4843C	R/W
RTCIO_RTC_GPIO_PIN6_REG	RTC configuration for pin 6	0x3FF48440	R/W
RTCIO_RTC_GPIO_PIN7_REG	RTC configuration for pin 7	0x3FF48444	R/W
RTCIO_RTC_GPIO_PIN8_REG	RTC configuration for pin 8	0x3FF48448	R/W
RTCIO_RTC_GPIO_PIN9_REG	RTC configuration for pin 9	0x3FF4844C	R/W
RTCIO_RTC_GPIO_PIN10_REG	RTC configuration for pin 10	0x3FF48450	R/W
RTCIO_RTC_GPIO_PIN11_REG	RTC configuration for pin 11	0x3FF48454	R/W
RTCIO_RTC_GPIO_PIN12_REG	RTC configuration for pin 12	0x3FF48458	R/W
RTCIO_RTC_GPIO_PIN13_REG	RTC configuration for pin 13	0x3FF4845C	R/W
RTCIO_RTC_GPIO_PIN14_REG	RTC configuration for pin 14	0x3FF48460	R/W
RTCIO_RTC_GPIO_PIN15_REG	RTC configuration for pin 15	0x3FF48464	R/W
RTCIO_RTC_GPIO_PIN16_REG	RTC configuration for pin 16	0x3FF48468	R/W

Name	Description	Address	Access
RTCIO_RTC_GPIO_PIN17_REG	RTC configuration for pin 17	0x3FF4846C	R/W
RTCIO_DIG_PAD_HOLD_REG	RTC GPIO hold register	0x3FF48474	R/W
GPIO RTC function configuration registers			
RTCIO_HALL_SENS_REG	Hall sensor configuration	0x3FF48478	R/W
RTCIO_SENSOR_PADS_REG	Sensor pads configuration register	0x3FF4847C	R/W
RTCIO_ADC_PAD_REG	ADC configuration register	0x3FF48480	R/W
RTCIO_PAD_DAC1_REG	DAC1 configuration register	0x3FF48484	R/W
RTCIO_PAD_DAC2_REG	DAC2 configuration register	0x3FF48488	R/W
RTCIO_XTAL_32K_PAD_REG	32KHz crystal pads configuration register	0x3FF4848C	R/W
RTCIO_TOUCH_CFG_REG	Touch sensor configuration register	0x3FF48490	R/W
RTCIO_TOUCH_PAD0_REG	Touch pad configuration register	0x3FF48494	R/W
””	””		
RTCIO_TOUCH_PAD9_REG	Touch pad configuration register	0x3FF484B8	R/W
RTCIO_EXT_WAKEUP0_REG	External wake up configuration register	0x3FF484BC	R/W
RTCIO_XTL_EXT_CTR_REG	Crystal power down enable GPIO source	0x3FF484C0	R/W
RTCIO_SAR_I2C_IO_REG	RTC I2C pad selection	0x3FF484C4	R/W

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>
- **ESP32forth** (eforth for ESP32) page maintenue par Peter FORTH
<https://www.forth2020.org/esp32forth>

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 PETREMAN
<https://github.com/MPETREMAN11/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

1/F.....	75	flush.....	117	renommer fichiers.....	129
analogRead.....	186	forget.....	50	rerun.....	170
and.....	37	FORTH.....	303	riscv.....	306
ansi.....	87	fsqrt.....	75	rm.....	129
asm.....	304	fvariable.....	75	rtos.....	307
BASE.....	78	GIT.....	134	S".....	82
bg.....	56	gpio_set_intr_type.....	161	S>F.....	76
binary.....	32	handleClient.....	299	SD.....	307
bluetooth.....	305	heure réelle.....	178	SD_MMC.....	307
c!.....	53	hex.....	32	see.....	47
c@.....	53	HEX.....	78	Serial.....	307
canaux ADC.....	185	HOLD.....	79	server.....	112
Codage ANSI.....	56	httpd.....	305	set-precision.....	74
commande AT.....	274	include.....	128	SF!.....	75
constant.....	54	insides.....	305	SF@.....	75
conversion analogique		internals.....	305	shift.....	36
numérique.....	191	interrupts.....	306	sockets.....	307
Couleurs de texte.....	56	interval.....	170	SPACE.....	82
create.....	94	is.....	90	spi.....	307
decimal.....	32	ledc.....	236, 306	SPI.....	217
DECIMAL.....	78	ledcAttachPin.....	236	SPIFFS.....	128, 307
defer.....	90	ledcWriteTone.....	236	streams.....	307
defPin:.....	140	liste des fichiers.....	128	struct.....	69
démarrage automatique.....	103	liste fichiers.....	128	structures.....	69, 71, 87, 307
DOES>.....	94	load.....	117	tasks.....	307
drop.....	52	login.....	111	telnetd.....	112, 308
dump.....	48	ls.....	128	Tera Term.....	105
dup.....	52	m!.....	143, 151, 261	thru.....	117
editor.....	87, 115, 305	m@.....	146, 154	to.....	61
effacer fichier.....	129	mémoire.....	53	u.....	35
EMIT.....	81	ms-ticks.....	178	value.....	54
ESP.....	305	Netbeans.....	134	variable.....	54
EXECUTE.....	89	normal.....	56	variables locales.....	60
f.....	74	oled.....	87, 194, 306	visual.....	308
F**.....	75	page.....	56	voir contenu fichier.....	129
F>S.....	76	pile de retour.....	53	web-interface.....	308
FATAN2.....	75	position de l'affichage.....	56	WiFi.....	308
fconstant.....	75	pseudo répertoire.....	128	wipe.....	116
FCOS.....	75	RECORDFILE.....	121	Wire.....	308
fg.....	56	registers.....	306	Wire.detect.....	198

xtensa.....	308	."	82	#.....	79
xtensa-assembler.....	244, 248	.s.....	48	#>.....	79
;.....	50	{.....	60	#S.....	79
:.....	50	}.....	60	+to.....	61
:noname.....	92	@.....	53	<#.....	79