ESP32forth
et Userwords

Marc PETREMANN

Version 1.0 - 05/02/26

Table des matières

Préambule

Ce manuel est destiné à comprendre comment intégrer de nouvelles fonctions issues de librairies matérielles et logicielles spécialisées. Cette intégration nécessite la création de fichiers spéciaux, dont **userwords.h** qui nous servira de base de départ.

Introduction

Avant d'aller dans le vif du sujet, il est nécessaire d'expliquer l'histoire de eForth.

L'histoire d'**eForth** est fascinante car elle représente une quête de simplicité extrême dans le monde déjà très minimaliste de Forth. Contrairement aux versions commerciales lourdes, eForth a été conçu pour être le "code source universel" des systèmes embarqués.

Voici l'essentiel de son origine et de sa philosophie.0

Les architectes : Bill Muench et Dr. C.H. Ting

Le projet eForth a débuté au début des années 90, principalement sous l'impulsion de **Bill Muench**, avec le soutien massif du **Dr. C.H. Ting**, une figure légendaire de la communauté Forth.

À l'époque, le langage Forth s'était fragmenté en de nombreuses versions propriétaires et complexes. L'objectif d'eForth était de créer un modèle **hautement portable** et **facile à implémenter** sur n'importe quel nouveau microcontrôleur.

Le "e" de eForth signifie souvent **"easy"**, **"educational"** ou **"embedded"**. Sa spécificité repose sur une architecture très particulière :

- **Le Noyau Minimal (Kernel) :** Au lieu d'écrire des centaines de mots en langage assembleur, eForth ne demande d'écrire qu'environ **30 primitives** en assembleur (les mots de base comme `+`, `DROP`, `FETCH`, etc.).
- **La Haute Couche en Forth :** Tout le reste du langage (les structures de contrôle, l'interpréteur, le compilateur) est écrit en Forth pur, en utilisant uniquement ces 30 primitives.
- **Portabilité :** Pour porter eForth sur une nouvelle puce (par exemple, passer d'un 8051 à un Arduino ou un processeur RISC-V), un programmeur n'a qu'à réécrire ces quelques primitives. Cela peut se faire en un week-end.

eForth a sauvé l'aspect "Open Source" et pédagogique de Forth à une époque où le standard **Forth-94 (ANS Forth)** devenait très volumineux.

Un héritage durable

Aujourd'hui, eForth est la base de prédilection pour ceux qui créent leurs propres microprocesseurs (sur FPGA par exemple) ou qui veulent comprendre comment fonctionne un compilateur de A à Z. C'est l'équivalent du "squelette" du langage : il est nu, mais parfaitement fonctionnel.

Note : Le Dr. Ting a publié de nombreux guides (comme le *eForth Implementation Guide*) qui restent des bibles pour les développeurs de systèmes "bare-metal".

ESP32forth est parti de cet héritage, avec un noyau écrit en C. Le langage C a été privilégié, car comme l'explique bien C. H. Ting, réécrire un noyau en assembleur XTENSA pur nécessitant les outils, les compétences dans ce nouvel assembleur.

En parallèle, la communauté ARDUINO dispose d'un outil « couteau Suisse », qui s'appelle ARDUINO IDE. Cet outil dispose d'un éditeur de code source, son compilateur C, son téléverseur de code binaire vers la carte de son choix.

Quand la gamme de cartes ESP32 évolue, l'IDE s'adapte en intégrant de nouvelles librairies. Avec un code source en fichier ino, il n'est pas nécessaire de réécrire tout le noyau.

Cette contrainte de passer par le langage C s'avère finalement être un atout majeur. Car porter une nouvelle version de eForth pour l'écosystème ESP32.

Une approche pragmatique

Passer par le langage **C** pour implémenter eForth sur un microcontrôleur moderne comme l'**ESP32** est une approche très populaire (souvent appelée *C-eForth* ou *ESP32-eForth*).

Bien que les puristes du Forth aiment l'assembleur pour le contrôle total, l'utilisation du C offre des avantages stratégiques majeurs, surtout sur une architecture complexe comme celle de l'ESP32 (processeur Xtensa ou RISC-V).

ESP32 n'est pas un simple processeur 8-bits ; c'est un monstre de puissance avec deux cœurs, du Wi-Fi et du Bluetooth.

- **Indépendance de l'architecture :** En utilisant le C, vous n'avez pas besoin d'apprendre l'assembleur spécifique aux cœurs **Xtensa** (complexe et peu documenté).
- **Évolutivité :** Si Espressif sort une nouvelle version de l'ESP32 basée sur une architecture différente (comme le passage au **RISC-V** sur les modèles récents), votre code eForth fonctionnera presque immédiatement sans réécriture majeure.

Avoir un accès simplifié aux bibliothèques Espressif est sans doute l'avantage le plus concret. L'écosystème ESP32 repose sur des milliers de lignes de code C/C++ pour :

- La gestion de la pile **TCP/IP** et du Wi-Fi.
- Le système de fichiers **LittleFS** ou SPIFFS.
- Le pilotage d'écrans complexes ou de capteurs via des bibliothèques existantes.

En écrivant eForth en C, vous pouvez créer des **"wrappers"** (enveloppes) très facilement. Vous exposez une fonction C complexe (comme `connect_wifi()`) sous la forme d'un mot Forth simple. Faire cela en assembleur pur serait un cauchemar technique.

L'ESP32 tourne généralement sous **FreeRTOS**.

- **Convivialité avec l'OS :** Un eForth écrit en C peut s'exécuter comme une simple "tâche" (task) au sein de FreeRTOS.
- Cela permet à votre code Forth de tourner sur un cœur pendant que le second cœur gère les communications sans fil, sans que l'un ne fasse planter l'autre.

Les compilateurs C modernes (comme GCC utilisé pour l'ESP32) sont extrêmement performants pour optimiser le code.

- **Direct Threading :** En C, on peut utiliser des techniques comme les "labels as values" (une extension GCC) pour implémenter un interpréteur Forth très rapide qui rivalise presque avec l'assembleur écrit à la main.

ESP32forth est une extension du code eForth pour toute la gamme des cartes ESP32. La version actuelle intègre nombre d'extensions matérielles : gestion WiFi, Bluetooth, accès GPIO, etc.

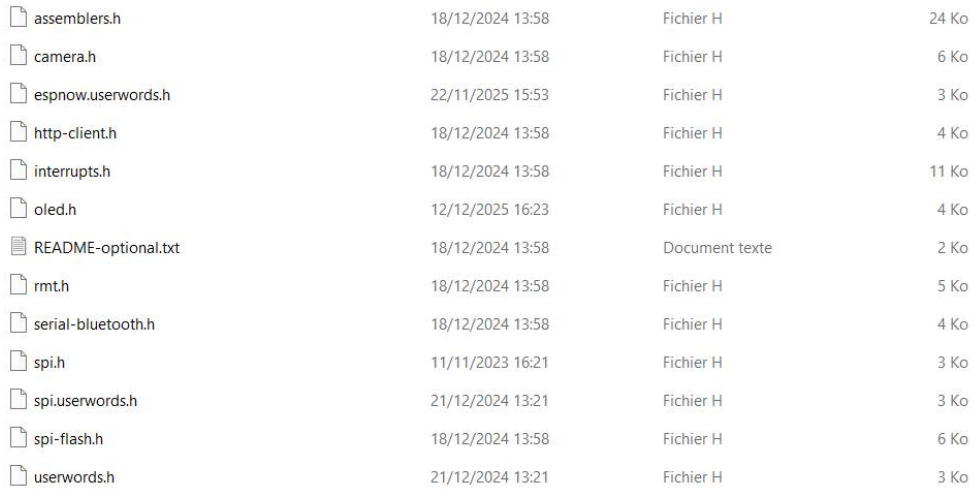
L'extensibilité de ESP32forth

Mais intégrer toutes les librairies ARDUINO dans une version monobloc devenait techniquement difficile :

- **grossissement** démesuré du vocabulaire et de la taille des fichiers binaires, taille souvent incompatible avec l'espace en mémoire Flash disponible, souvent limité à 2 Mo en version standard ;
- **délais de compilation insupportables**. La compilation et le téléversement d'un noyau de base prend de une à cinq minutes. A chaque erreur, il faut corriger les codes sources et recommencer la compilation. Tout ceci pour embarquer des extensions souvent inutilisées.

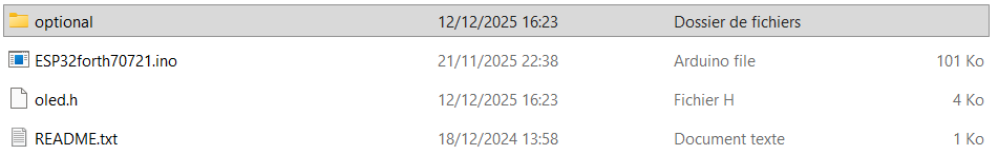
Les extensions en option

Depuis la version ESP32forth 7.0.7.14, certaine extensions sont maintenant disponibles en option. Ces extensions sont situées dans le dossier **optional** :



assemblers.h	18/12/2024 13:58	Fichier H	24 Ko
camera.h	18/12/2024 13:58	Fichier H	6 Ko
espnow.userwords.h	22/11/2025 15:53	Fichier H	3 Ko
http-client.h	18/12/2024 13:58	Fichier H	4 Ko
interrupts.h	18/12/2024 13:58	Fichier H	11 Ko
oled.h	12/12/2025 16:23	Fichier H	4 Ko
README-optional.txt	18/12/2024 13:58	Document texte	2 Ko
rmt.h	18/12/2024 13:58	Fichier H	5 Ko
serial-bluetooth.h	18/12/2024 13:58	Fichier H	4 Ko
spi.h	11/11/2023 16:21	Fichier H	3 Ko
spi.userwords.h	21/12/2024 13:21	Fichier H	3 Ko
spi-flash.h	18/12/2024 13:58	Fichier H	6 Ko
userwords.h	21/12/2024 13:21	Fichier H	3 Ko

Si vous souhaitez rajouter les mots de la librairie *oled*, il faut copier le fichier **optional/oled.h** dans le dossier contenant la version ESP32forth :



optional	12/12/2025 16:23	Dossier de fichiers
ESP32forth70721.ino	21/11/2025 22:38	Arduino file 101 Ko
oled.h	12/12/2025 16:23	Fichier H 4 Ko
README.txt	18/12/2024 13:58	Document texte 1 Ko

A ce stade, si on lance une compilation du noyau avec ARDUINO IDE, la version téléchargée de ESP32forth intégrera 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 OledSetRotation OledInvert
OledTextsize OledSetCursor OledPixel OledDrawL OledCirc OledCircF OledRect
OledRectF OledRectR OledRectRF OledDrawChar OledDrawBitmap OledTriangle
OledTriangleF OledEllipse OledEllipseF OledScrollL OledScrollR OledScrollStop
oled-builtins
```

Ainsi, si votre projet ne nécessite pas un afficheur oled, ne mettez pas cette option dans le dossier racine contenant ESP32forth. Ceci évite de surcharger inutilement le noyau ESP32forth téléversé sur la carte ESP32, laissant plus de place disponible pour vos définitions de mots en FORTH.

L'option userwords.h

Bien que figurant dans la copie écran affichée précédemment, cette option n'existe pas dans la version ESP32forth disponible ici : https://esp32forth.appspot.com/ESP32forth.html

Ce fichier doit être créé par vous. Il ne peut contenir que du code en langage C. Exemple :

```
#define USER_WORDS \
  Y(tftdemo, setuptftdemo(); DROP) \
  Y(tftinit, tft.init(); DROP) \
  Y(tftcls, tft.fillScreen(TFT_BLACK);)
```

Portez une attention particulière sur la fin de chaque ligne qui doit se terminer par `\` ce qui permet aux macros de s'enchaîner correctement.

Voyons en détail une ligne de code :

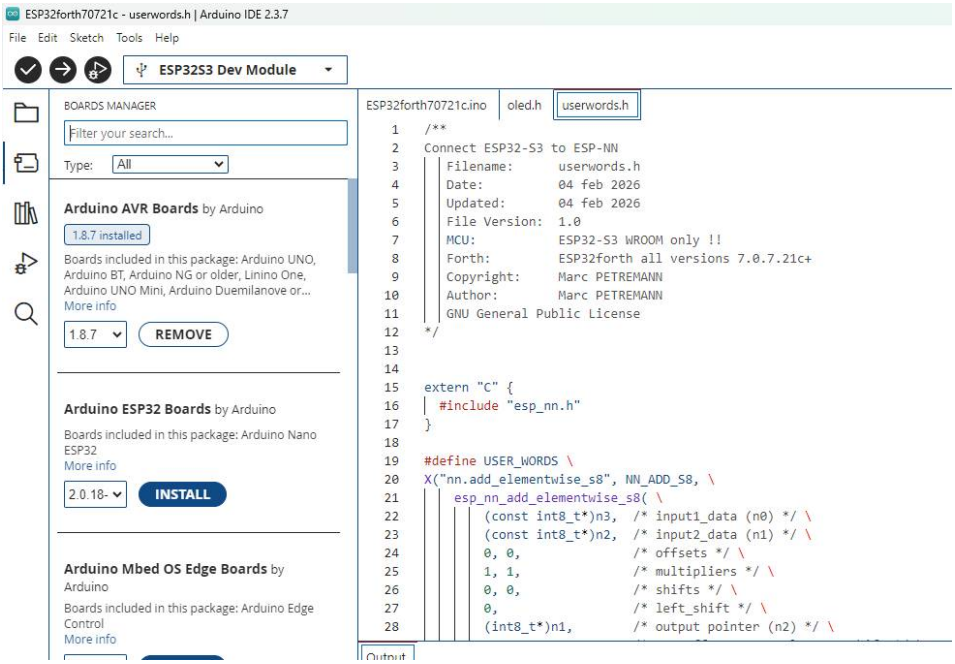
```
Y(tftdemo, setuptftdemo(); DROP) \
```

Dans celle ligne, la macro-instruction Y() permet de créer le mot `tftdemo` qui sera rajouté au dictionnaire FORTH dans ESP32forth.

L'appel de `tftdemo` depuis ESP32forth va exécuter le code C compilé dans la fonction `setuptftdemo(); DROP`.

En créant ces options dans le fichier **userwords.h**, on évite de modifier le code source **ESP32forth.ino**.

Au moment de compiler ESP32forth et le téléverser vers votre carte ESP32, vérifiez que le fichier userwords.h est bien dans le dossier racine du projet de compilation :



Nous allons aborder les différents types de macros.

Les macros de définition de mots FORTH depuis le langage C

Depuis 1983 on sait méta-compiler FORTH par lui-même. Il s'agit simplement d'utiliser un méta-compilateur, écrit en langage FORTH, pour traiter un code source, lui-même écrit en langage FORTH. Pour utiliser cette technologie, il y a cependant certaines contraintes :

1. avoir un méta-compilateur assez simple à utiliser,
2. avoir un noyau en assembleur pour le système cible,
3. dans le cas des micro-contrôleurs, générer une cible en code binaire, compatible avec les scripts de téléversement.

Pour ESP32forth, le méta-compilateur est la partie la plus facile. Mais pour ce qui est du noyau en assembleur, il faut avoir un assembleur qui sait gérer les instructions XTENSA. Ces instructions sont semblables à celles des processeurs RISC-V, mais avec des spécificités.

La partie la plus ardue concerne les innombrables registres et drapeaux que doit gérer ESP32. Une petite erreur et le code ne fonctionne pas. Rajoutez à ceci la difficulté de tracer et mettre au point un code pour un processeur cible. S'il fallait faire du reverse-engeenering sur tout le noyau écrit en C, il faudrait un temps extraordinairement long et une compétence hors norme pour tout assimiler. Rien que le livre des spécifications techniques des cartes ESP32 fait des centaines de pages.

Pour résumer, refaire ce qui existe déjà en langage C est hors de portée de quasiment n'importe quel programmeur. Non pas que définir quelques mots de base en assembleur XTENSA soit difficile. Le vrai souci reste la gestion de l'environnement final, c'est à dire la gestion mémoire, les interruptions, la couche communication.

Bref, si ce travail est déjà fait en langage C, autant exploiter ces acquis pour économiser notre temps et nos ressources personnelles.

C'est à ce stade qu'interviennent les macros en langage C, permettant de rajouter nos propres définitions lors de la compilation du noyau ESP32forth.

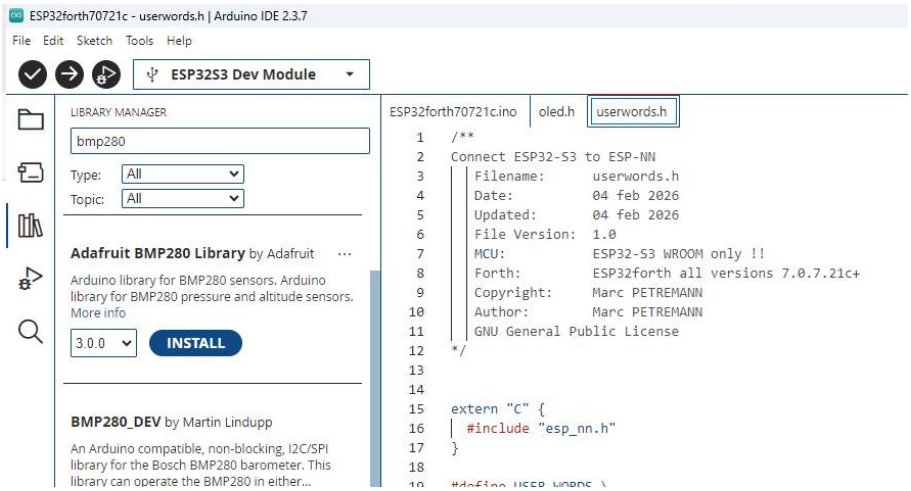
Choisir une librairie pour étendre le dictionnaire ESP32forth

La carte ESP32 peut utiliser beaucoup de modules qui sont initialement créés pour l'environnement ARDUINO : lecteurs de badge, carte SIM GSM, rubans LEDs, moteurs pas à pas, etc.

Évidemment, avec la bonne documentation, en agissant sur les registres ESP32 adéquats, on pourrait tout écrire en FORTH. Nous avons essayé. C'est vraiment très compliqué. Les codes sources ARDUINO sont répartis sur des milliers de fichiers, avec des sélections complexes dépendant de chaque version de carte et ses caractéristiques. Le compilateur intégré à ARDUINO IDE fait ce travail au moment de la sélection de la carte cible. A nous de choisir la librairie qui est destinée à notre projet.

Prenons un cas concret. Vous voulez monter une mini station météorologique. Votre choix s'arrête sur le module BMP280 :

Vérifions que cette carte dispose de sa librairie dans ARDUINO IDE :



Oui, elle est disponible. Elle apparaît sous le nom **Adafruit BMP280 Library**. Cliquez sur **INSTALL** et cette librairie sera chargée dans ARDUINO IDE.

En bas de présentation de cette librairie, il y a « More Info » :

Cliquez sur ce lien vous emmène ici :

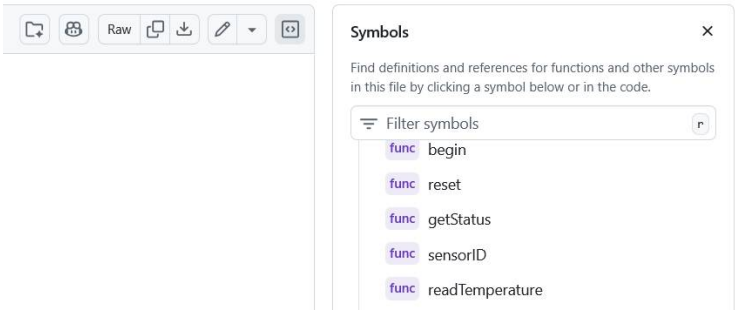
https://github.com/adafruit/Adafruit_BMP280_Library

C'est le dépôt GITHUB qui contient les fichiers sources de cette librairie pour BMP280.

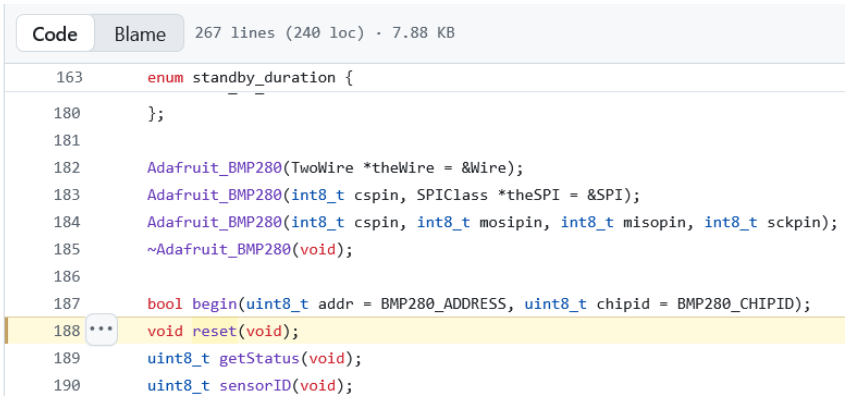
Notre fichier userwords.h devra contenir cette toute première ligne de code :

```
#include <Adafruit_BMP280.h>
```

Sur le dépôt GITHUB, ouvrez le volet latéral. Il indique les classes et fonctions disponibles dans cette librairie :



Cliquez sur une fonction, par exemple reset :



Ici, en surligné jaune, nous voyons la définition C complète du code de `reset()`. Rajoutons maintenant ces deux lignes à **userwords.h** :

```
#define USER_WORDS \
  X("bmp.reset", BMP_RESET, reset();)
```

Tant que nous y sommes, nous avons deux autres mots plutôt faciles à définir :

```
#define USER_WORDS \
  X("bmp.reset", BMP_RESET, reset();) \
  X("bmp.getStatus", BMP_GET_STATUS, PUSH getStatus();) \
  X("bmp.sensorID", BMP_SENSORID, PUSH sensorID();)
```

Le premier paramètre de la macro `X` est une chaîne de caractères dans laquelle figure le mot FORTH à rajouter au dictionnaire de ESP32forth.

Le second paramètre est un label qui permet un appel externe. Ce sera détaillé ultérieurement.

Le troisième paramètre contient le code C exécuté par le mot défini par `X()`.

Ici, nous avons choisi des noms pour notre dictionnaire en concordance similaire aux noms des fontions C. Nous arions aussi bien pu définir un mot FORTH `getBmp280Status` :

```
  X("getBmp280Status", BMP_GET_STATUS, PUSH getStatus();) \
```

La macro Y

C'est la plus simple. Elle permet d'ajouter un mot ne contenant que des lettres, chiffres et le caractère souligné. Exemple :

```
#define USER_WORDS \
  Y(MY_WORD123, c_function_to_call())
```

Paramètres :

- mot FORTH à définir, avec lettres, chiffres et éventuellement caractère souligné ;
- le code en langage C à exécuter quand le mot défini sera appelé.

Tous les mots définis par la macro Y sont intégrés au vocabulaire `forth`.

La macro X

Si le mot FORTH à définir contient d'autres caractères que A..Z,a..z,0..9 et souligné, on utilisera la macro `X` :

```
#define USER_WORDS \
  X("myword!", MY_WORD_BANG, c_function_to_call()) \
```

Paramètres :

- mot FORTH à définir, avec lettres, chiffres et éventuellement caractère souligné ;
- un label d'appel, écrit uniquement en caractères majuscules et caractère souligné. Le label doit être unique ;
- le code en langage C à exécuter quand le mot défini sera appelé.

Tous les mots définis par la macro X sont intégrés au vocabulaire `forth`.

La macro YV

Similaire à la macro `Y`, mais admet comme premier paramètre le nom du vocabulaire dans lequel le mot sera défini :

```
YV(ledc, ledcWriteNote, \
    tos = (cell_t) (1000000 * ledcWriteNote(n2, (note_t) n1, n0)); NIPn(2))
```

Ici, le mot `ledcWriteNote` sera défini dans le vocabulaire `ledc`.

La macro XV

Similaire à la macro `X`, mais admet comme premier paramètre le nom du vocabulaire dans lequel le mot sera défini :

```
XV(SPIFFS, "SPIFFS.begin", SPIFFS_BEGIN, \
   tos = SPIFFS.begin(n2, c1, n0); NIPn(2)) \
```

Ici, le mot `SPIFFS.begin` sera défini dans le vocabulaire `SPIFFS`.

La macro V

Cette macro sert à définir un nouveau vocabulaire :

```
#define OPTIONAL_HTTP_CLIENT_VOCABULARY V(HTTPClient)
```

Ici, on définit le vocabulaire `HTTPClient`.

Note : si vous ne maîtrisez pas les vocabulaires, définissez des mots en les préfixant, exemple :

```
#define USER_WORDS \
X("nn.add_elementwise_s8", NN_ADD_S8, \
    esp_nn_add_elementwise_s8( \
    ….code C here…  \
    ); \
    DROPn(4))
```

Dans ce code, on définit le mot `nn.add_elementwise_s8`. Le préfixe `nn.` Fait référence à **Neural Network**, une des librairies Xtensa. C'est une pure convention de nomage, utilisée par nombre de programmeurs. On aurait aussi bien pu définir le mot `add_s8` ou même `addS8`.

L'intérêt de préfixer certains mots permet de comprendre, en relisant son code, que le mot préfixéfait référence à un contexte précis. Ainsi, dans le vocabulaire `serial`, on retrouve ces mots :

```
Serial.begin Serial.end Serial.available Serial.readBytes Serial.write
Serial.flush Serial.setDebugOutput Serial1.begin Serial1.end Serial1.available
Serial1.readBytes Serial1.write Serial1.flush serial-builtin
```

Dans ce vocabulaire, nous avons deux mots `readBytes` (en rouge), mais préfixés respectivement `serial` et `serial1`. Ce sont bien deux mots distincts. En les retrouvant dans un code source, nous comprenons immédiatement que le mot `Serial.readBytes` opère sur le port UART0, alors que `Serial1.readBytes` opère sur le port UART1.

Attention, nous ne sommes pas dans la programmation objet. La syntaxe des mots FORTH offre une vraie liberté. Pour rappel, un mot FORTH est défini par tout groupe de caractères non séparé par un espace : `{..}` peut être un mot FORTH. Il suffit de le définir en FORTH.

Appel de sous-programmes

Il est des situations où il est nécessaire de traiter certains paramètres au travers d'une fonction utilisateur plutôt que de se connecter à une fonction de librairie :

```
static esp_err_t EspnowRegisterRecvCb(cell_t xt) {
  espnow_recv_cb_xt = xt;
  return esp_now_register_recv_cb(HandleRecv);
}

#define OPTIONAL_ESPNOW_VOCABULARY V(espnow)
#define OPTIONAL_ESPNOW_SUPPORT \
  XV(espnow, "esp_now_register_recv_cb", ESP_NOW_REGISTER_RECV_CB, n0 =
EspnowRegisterRecvCb(n0);) \
```

Ici, nous définissons le mot `esp_now_register_recv_cb` qui se réfère à notre fonction utilisateur `EspnowRegisterRecvCb`.

Passage des paramètres

Le passage de paramètres entre ESP32forth et les fonctions du langage C est assez délicat. A la moindre erreur, au mieux le mot réagit autrement que le comportement attendu. Au pire, l'interpréteur FORTH plante !

Commençons par le cas le plus simple.

Appel de fonction sans paramètre

Beaucoup de fonctions de librairies ne nécessitent aucun paramètre et ne restituent également aucune donnée :

```
YV(oled, OledCLS, oled_display->clearDisplay()) \
```

Ici, le mot `OledCLS` exécute la fonction `clearDisplay()` de la librairie oled.

Appel d'une fonction retournant une donnée

Voyons le cas d'une fonction qui renvoie une donnée mais n'en utilise aucune en entrée :

```
YV(oled, OledAddr, PUSH &oled_display) \
```

La donnée est empilée sur la pile FORTH par la macro `PUSH`. Ici, le mot OledAddr retourne l'adresse d'une structure rattachée à la librairie oled.

Appel d'une fonction utilisant une seule donnée

Voyons maintenant un mot FORTH utilisant une fonction utilisant une seule donnée, mais ne renvoyant rien sur la pile FORTH :

```
YV(oled, OledTextsize, oled_display->setTextSize(n0); DROP) \
```

Ceci définit le mot FORTH `OledTextsize` qui utilise un paramètre sur la pile. Ici, le paramètre utilise `n0` qui est un alias de `tos`. Cette définition est acceptée :

```
YV(oled, OledTextsize, oled_display->setTextSize(tos); DROP) \
```

Le code C appelle ensuite **DROP** défini dans le code source de ESP32forth :

```
#define DROP (tos = *sp--)
```

Pourquoi appeler DROP ? En fait, à l'exécution de `OledTextsize`, en FORTH, nous empilons prélablement un paramètre. Exemple :

```
2 OledTextsize
```

S'il n'y avait pas `DROP` dans la macro qui définit `OledTextsize`, cette valeur `2` resterait sur la pile de données.

Appel de fonction utilisant plusieurs données

Commençons par une fonction utilisant trois données mais ne retournant aucun paramètre :

```
YV(oled, OledPixel, oled_display->drawPixel(n2, n1, n0); DROPn(3)) \
```

Ici on définit le mot `OledPixel` qui utilise trois paramètres :

- x qui est la position en x
- y qui est la position y

- color qui est la couleur du pixel à afficher sur un écran oled.

Exemple :

```
50 20 1 OledPixel  \ affiche pixel lumineux à la position 50,20
```

Pour enlever les paramètres 50 20 et 10 de la pile FORTH, le code C aurait pu être écrit ainsi :

```
YV(oled, OledPixel, oled_display->drawPixel(n2, n1, n0); DROP; DROP; DROP;) \
```

Ici, l'ensemble du code C a été mis en évidence en bleu. Remarquez la présence de `DROP` trois fois. Ces `DROP` successifs peut être remplacés par la définition `DROPn` qui est utilisée dans l'exemple écrit un peu plus haut.

Pour vérifier l'ordre des paramètres dans une macro pour **ESP32forth**, il faut suivre une règle stricte liée à la structure de la pile (LIFO - Last In, First Out) utilisée par Brad Nelson.

Dans l'implémentation de la macro :

- **`n0`** est toujours le sommet de la pile (**TOS** - Top Of Stack).
- **`n1`** est l'élément juste en dessous, et ainsi de suite (`n2`, `n3`, ...).

Si vous écrivez en FORTH : `param3 param2 param1 param0 myWord` L'ordre dans la macro C++ sera :

- `n0` = param0
- `n1` = param1
- `n2` = param2
- `n3` = param3

Conserver un résultat après envoi de paramètres

Certaines fonctions utilisent un ou plusieurs paramètres, puis retournent une donnée. Voici un exemple qui utilise deux paramètres et empile un résultat :

```
YV(oled, OledBegin, n0 = oled_display->begin(n1, n0, true, true); NIP) \
```

La fonction `begin` de la librairie oled utilise quatre paramètres, mais n'utilise que les données `n0` et `n1` déposées sur la pile de données. Le résultat est injecté dans `n0` par begin() :

```
n0 = oled_display->begin(n1, n0, true, true); NIP)
```

Ordre de passage des paramètres

Soyez attentif à l'ordre de passage des paramètres. Pour ne pas partir dans un code confis, il est fortement recommandé de conserver en FORTH le même ordre des paramètres que celui décrit dans la fonction en langage C. Prenons cette fonction `esp_nn_add_elementwise_s8_esp32s3()` qui est une fonction de multiplication vectorielle d'un réseau neuronal. Prototype de cette fonction :

```
void esp_nn_add_elementwise_s8_esp32s3(const int8_t *input1_data,
                                       const int8_t *input2_data,
                                       const int32_t input1_offset,
                                       const int32_t input2_offset,
                                       const int32_t input1_mult,
                                       const int32_t input2_mult,
                                       const int32_t input1_shift,
                                       const int32_t input2_shift,
                                       const int32_t left_shift,
                                       int8_t *output,
                                       const int32_t out_offset,
                                       const int32_t out_mult,
```

```
                                        const int32_t out_shift,
                                        const int32_t activation_min,
                                        const int32_t activation_max,
                                        const int32_t size);
```

Si vous voulez vérifier, cette fonction est disponible ici : https://github.com/espressif/esp-nn/blob/596b08401a63da3a2e1b40868c442f582a99ae26/include/esp_nn_esp32s3.h

Après analyse du code, il s'avère possible d'exploiter seulement quatre paramètres. Voici la définition via la macro X :

```
extern "C" {
  #include "esp_nn.h"
}

#define USER_WORDS \
/* 2. Addition élément par élément (Vecteurs) */ \
/* Forth: ( addr1 addr2 addr_out len -- ) */ \
X("nn.add_elementwise_s8", NN_ADD_S8, \
    esp_nn_add_elementwise_s8( \
        (const int8_t*)n3,  /* input1_data (n3) */ \
        (const int8_t*)n2,  /* input2_data (n2) */ \
        0, 0,               /* offsets */ \
        1, 1,               /* multipliers */ \
        0, 0,               /* shifts */ \
        0,                  /* left_shift */ \
        (int8_t*)n1,        /* output pointer (n1) */ \
        0, 1, 0,            /* out_offset, out_mult, out_shift */ \
        -128, 127,          /* activation min/max */ \
        (int32_t)n0         /* size (n0) */ \
    ); \
    DROPn(4))
```

Ici on a défini un mot `nn.add_elementwise_s8`. Nous aurions pu faire plus court : `nn.add_s8`.

Mais pour le projet envisagé, il est préférable que les mots FORTH collent au plus près des noms de fonctions utilisées en langage C.

Au passage, notez le nombre de lignes qu'occupe notre code en langage C. Ce choix a été fait pour avoir un code lisible et facile à modifier.

Note : le mot FORTH `nn.add_elementwise_s8` n'a pas encore été testé. Il exécute une addition vectorielle de deux champs de longueur `size`, pointés par `n3` et `n2`. Le résultat est stocké en `n1`. Ce code n'est exploitable que sur une carte ESP32-S3. Cette carte dispose d'un espace PSRAM de 16Mo ou 32Mo selon les modèles de cartes.

Conclusion provisoire

Ce document mérite d'être complété.

Si vous avez des questions, ou constatez des omissions, des erreurs, ou souhaitez apporter des informations supplémentaires, vous pouvez me contacter : esp32forth@arduino-forth.com