ESP32forth and Userwords

Marc PETREMANN

Version 1.0 - 05/02/26

Table of Contents

Preamble 3
Introduction 4
The architects: Bill Muench and Dr. CH Ting 4
A lasting legacy 4
A pragmatic approach 5
The extensibility of ESP32forth 5
Optional extensions 6
The userwords.h ....6 option
FORTH word definition macros from the C8 ....language
8 dictionary..
10 macro...
Macro X 11
11 macro...
The macro XV 11
Macro V 11
Subprogram call 12
Passing parameters 13
Parameterless Function Call 13
Calling a function that returns data 13
Calling a function using a single piece of data 13
Function call using multiple data 13
Retaining a result after sending parameters 14
Parameter passing order 14
Provisional conclusion 16

Preamble

This manual aims to explain how to integrate new functions from specialized hardware and software libraries. This integration requires the creation of special files, including **userwords.h** , which will serve as our starting point.

Introduction

Before getting to the heart of the matter, it is necessary to explain the history of eForth.

The story of **eForth** is fascinating because it represents a quest for extreme simplicity in the already minimalist world of Forth. Unlike heavyweight commercial versions, eForth was designed to be the "universal source code" for embedded systems.

Here is the essence of its origin and philosophy.

The architects: Bill Muench and Dr. CH Ting

The eForth project began in the early 90s, primarily driven by **Bill Muench** , with massive support from **Dr. CH Ting** , a legendary figure in the Forth community.

At the time, the Forth language had fragmented into numerous proprietary and complex versions. eForth's goal was to create a **highly portable model** that **was easy to implement** on any new microcontroller.

The "e" in eForth often stands for **"easy"** , **"educational"** , or **"embedded"** . Its unique feature lies in its very specific architecture:

- **The Minimal Kernel:** Instead of writing hundreds of words in assembly language, eForth only requires writing about **30 primitives** in assembly (basic words like `+` , `DROP` , `FETCH` , etc.).
- **The High Layer in Forth:** Everything else in the language (control structures, interpreter, compiler) is written in pure Forth, using only these 30 primitives.
- **Portability:** To port eForth to a new chip (for example, moving from an 8051 to an Arduino or a RISC-V processor), a programmer only needs to rewrite these few primitives. This can be done in a weekend.

eForth saved the "Open Source" and educational aspect of Forth at a time when the **Forth-94 standard (ANS Forth)** was becoming very bulky.

A lasting legacy

Today, eForth is the preferred base for those who create their own microprocessors (on FPGA for example) or who want to understand how a compiler works from A to Z. It is the equivalent of the "skeleton" of the language: it is bare, but perfectly functional.

Note: Dr. Ting has published numerous guides (such as the *eForth Implementation Guide*) which remain bibles for developers of "bare-metal" systems.

ESP32forth started from this heritage, with a kernel written in C. The C language was preferred because, as CH Ting explains, rewriting a kernel in pure XTENSA assembler requires the tools and skills in this new assembler.

In parallel, the ARDUINO community has a "Swiss Army knife" tool called ARDUINO IDE. This tool includes a source code editor, a C compiler, and a binary code uploader for the board of your choice.

As the ESP32 board range evolves, the IDE adapts by integrating new libraries. With source code in an .ino file, it's not necessary to rewrite the entire kernel.

This constraint of using the C language ultimately proves to be a major advantage, as it allows for porting a new version of eForth to the ESP32 ecosystem.

A pragmatic approach

Using the **C language** to implement eForth on a modern microcontroller like the **ESP32** is a very popular approach (often called *C-eForth* or *ESP32-eForth*).

Although Forth purists like assembler for total control, using C offers major strategic advantages, especially on a complex architecture like that of the ESP32 (Xtensa or RISC-V processor).

ESP32 is not just an 8-bit processor; it's a powerhouse with two cores, Wi-Fi, and Bluetooth.

- **Architecture independence:** By using C, you do not need to learn the **Xtensa core-specific assembler** (complex and poorly documented).
- **Scalability:** If Espressif releases a new version of ESP32 based on a different architecture (such as the switch to **RISC-V** on recent models), your eForth code will work almost immediately without major rewriting.

Having simplified access to Espressif libraries is undoubtedly the most tangible advantage. The ESP32 ecosystem relies on thousands of lines of C/C++ code to:

- **TCP/IP** stack and Wi-Fi management.
- **LittleFS** or SPIFFS file system .
- Controlling complex screens or sensors via existing libraries.

By writing eForth in C, you can create **"wrappers"** very easily. You expose a complex C function (like `connect_wifi()`) as a simple Forth word. Doing this in pure assembly would be a technical nightmare.

The ESP32 usually runs under **FreeRTOS** .

- **OS compatibility:** An eForth written in C can run as a simple "task" within FreeRTOS.
- This allows your Forth code to run on one core while the second core handles wireless communications, without one crashing the other.

Modern C compilers (like GCC used for ESP32) are extremely efficient at optimizing code.

- **Direct Threading:** In C, one can use techniques like "labels as values" (a GCC extension) to implement a very fast Forth interpreter that almost rivals hand-written assembly.

ESP32forth is an extension of the eForth code for the entire range of ESP32 boards. The current version integrates a number of hardware extensions: WiFi management, Bluetooth, GPIO access, etc.

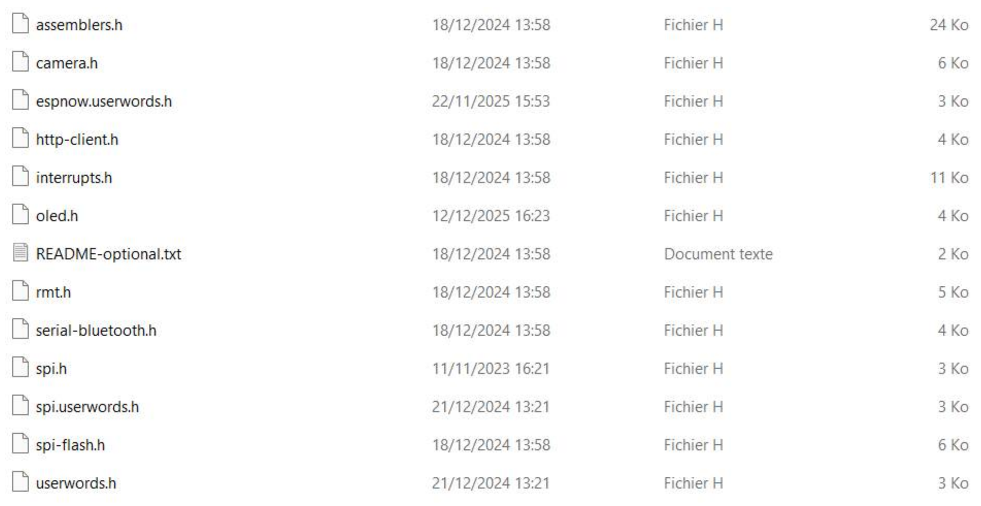
The extensibility of ESP32forth

However, integrating all the ARDUINO libraries into a single-block version was becoming technically difficult:

- **enlargement** of the vocabulary and size of binary files, size often incompatible with the available Flash memory space, often limited to 2 MB in standard version;
- **Unbearable compilation times** . Compiling and uploading a basic kernel takes one to five minutes. Each error requires correcting the source code and restarting the compilation. All this just to include extensions that are often unused.

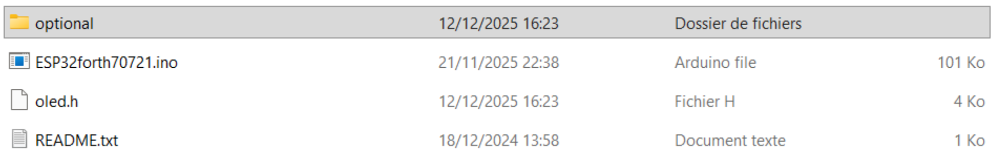
Optional extensions

Since version ESP32forth 7.0.7.14, certain extensions are now available as options. These extensions are located in the **optional folder** :



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

If you wish to add the words from the *oled library* , you must copy the **optional/oled.h file** into the folder containing the ESP32forth version:



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

At this stage, if we launch a kernel compilation with ARDUINO IDE, the downloaded version of ESP32forth will integrate the `OLED vocabulary` :

```
--> 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
```

Therefore, if your project does not require an OLED display, do not place this option in the root folder containing ESP32forth. This avoids unnecessarily overloading the ESP32forth kernel uploaded to the ESP32 board, leaving more space available for your FORTH word definitions.

The userwords.h option

Although shown in the screenshot displayed earlier, this option does not exist in the ESP32forth version available here: https://esp32forth.appspot.com/ESP32forth.html

This file must be created by you. It can only contain code written in the C language. Example:

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

Pay particular attention to the end of each line, which must end with `\` , allowing the macros to chain together correctly.

Let's look at a line of code in detail:

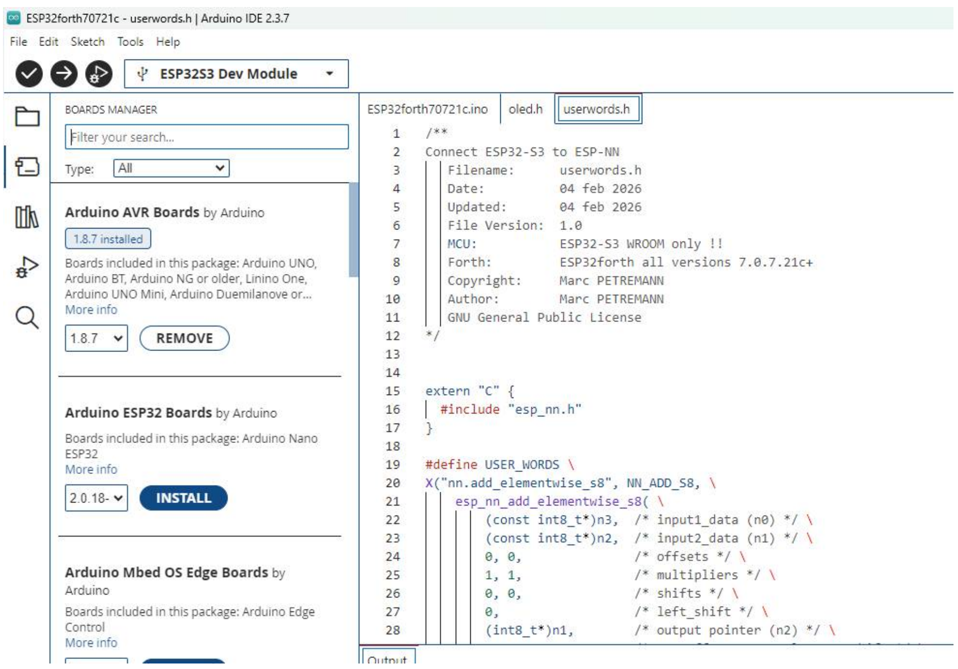
```
   Y( tftdemo , setuptftdemo(); DROP) \
```

In this line, the macro instruction Y() allows the creation of the word `tftdemo` which will be added to the FORTH dictionary in ESP32forth.

Calling `tftdemo` from ESP32forth will execute the compiled C code in the `setuptftdemo()` `function; DROP` .

By creating these options in the **userwords.h file** , we avoid modifying the **ESP32forth.ino source code** .

When compiling ESP32forth and uploading it to your ESP32 board, verify that the userwords.h file is indeed in the root folder of the compilation project:



Arduino AVR Boards by Arduino

1.8.7 installed

Boards included in this package: Arduino UNO, Arduino BT, Arduino NG or older, Linino One, Arduino UNO Mini, Arduino Duemilanove or...
More info

1.8.7 REMOVE

Arduino ESP32 Boards by Arduino

Boards included in this package: Arduino Nano ESP32
More info

2.0.18- INSTALL

Arduino Mbed OS Edge Boards by Arduino

Boards included in this package: Arduino Edge Control
More info

```
/**
Connect ESP32-S3 to ESP-NN
    Filename:      userwords.h
    Date:          04 feb 2026
    Updated:       04 feb 2026
    File Version:  1.0
    MCU:           ESP32-S3 WROOM only !!
    Forth:         ESP32forth all versions 7.0.7.21c+
    Copyright:     Marc PETREMANN
    Author:        Marc PETREMANN
    GNU General Public License
*/


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

#define USER_WORDS \
X("nn.add_elementwise_s8", NN_ADD_S8, \
    esp_nn_add_elementwise_s8( \
        (const int8_t*)n3,  /* input1_data (n0) */ \
        (const int8_t*)n2,  /* input2_data (n1) */ \
        0, 0,               /* offsets */ \
        1, 1,               /* multipliers */ \
        0, 0,               /* shifts */ \
        0,                  /* left_shift */ \
        (int8_t*)n1,        /* output pointer (n2) */ \
```

We will discuss the different types of macros.

FORTH word definition macros from the C language

Since 1983, it has been possible to meta-compile FORTH on its own. This simply involves using a meta-compiler, written in the FORTH language, to process source code, itself written in FORTH. However, there are certain constraints to using this technology:

1. to have a meta-compiler that is fairly simple to use,
2. to have a kernel in assembler for the target system,
3. In the case of microcontrollers, generate a target in binary code, compatible with upload scripts.

For ESP32forth, the metacompiler is the easiest part. But for the kernel in assembler, you need an assembler that can handle XTENSA instructions. These instructions are similar to those of RISC-V processors, but with specific characteristics.

The most challenging part involves the countless registers and flags that the ESP32 has to manage. One small error and the code won't work. Add to that the difficulty of designing and debugging code for a target processor. If you had to reverse-engineer the entire kernel written in C, it would take an extraordinarily long time and exceptional skill to assimilate everything. The ESP32 technical specifications book alone is hundreds of pages long.

In short, rewriting what already exists in C is beyond the reach of almost any programmer. It's not that defining a few basic words in XTENSA assembly is difficult. The real challenge lies in managing the final environment, namely memory management, interrupts, and the communication layer.

In short, if this work has already been done in C language, we might as well use those skills to save our time and personal resources.

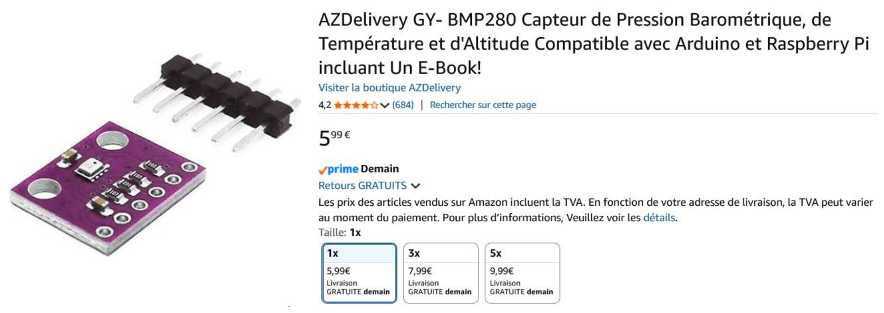
This is where C language macros come in, allowing us to add our own definitions during the compilation of the ESP32forth kernel.

Choose a library to extend the ESP32forth dictionary

The ESP32 board can use many modules that are originally created for the ARDUINO environment: badge readers, GSM SIM card, LED strips, stepper motors, etc.

Obviously, with the right documentation and by manipulating the appropriate ESP32 registers, we could write everything in FORTH. We tried. It's really very complicated. The Arduino source code is spread across thousands of files, with complex selections depending on each board version and its characteristics. The compiler integrated into the Arduino IDE handles this when selecting the target board. It's up to us to choose the library that's right for our project.

Let's take a concrete example. You want to set up a mini weather station. You choose the BMP280 module:



AZDelivery GY- BMP280 Capteur de Pression Barométrique, de Température et d'Altitude Compatible avec Arduino et Raspberry Pi incluant Un E-Book!

Visiter la boutique AZDelivery

4,2 ★★★★☆ (684) | Rechercher sur cette page

5,99 €

prime Demain

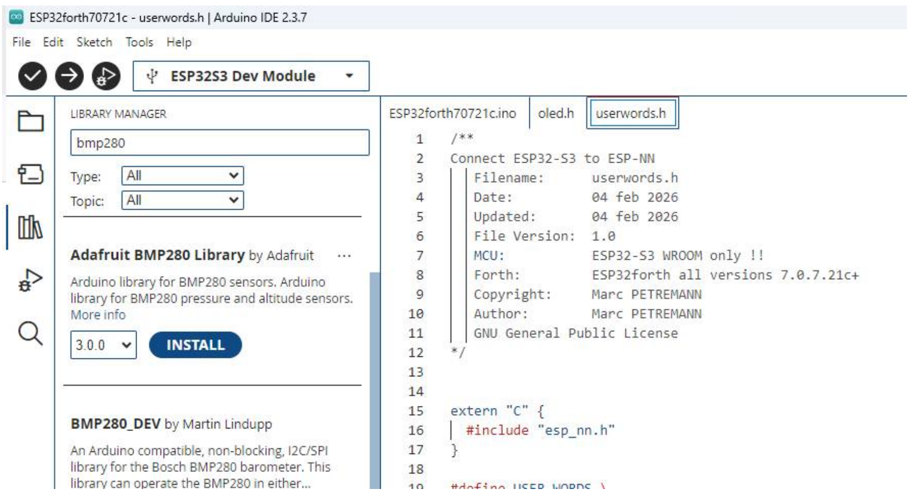
Retours GRATUITS

Les prix des articles vendus sur Amazon incluent la TVA. En fonction de votre adresse de livraison, la TVA peut varier au moment du paiement. Pour plus d'informations, Veuillez voir les détails.

Taille: 1x

1x	3x	5x
5,99€ Livraison GRATUITE **demain**	7,99€ Livraison GRATUITE **demain**	9,99€ Livraison GRATUITE **demain**

Let's check that this board has its library in the ARDUINO IDE:


ESP32forth70721c - userwords.h | Arduino IDE 2.3.7
File Edit Sketch Tools Help
ESP32S3 Dev Module
ESP32forth70721c.ino
oled.h
userwords.h

LIBRARY MANAGER

bmp280

Type: All

Topic: All

Adafruit BMP280 Library by Adafruit

Arduino library for BMP280 sensors. Arduino library for BMP280 pressure and altitude sensors. More info

3.0.0 INSTALL

BMP280_DEV by Martin Lindupp

An Arduino compatible, non-blocking, I2C/SPI library for the Bosch BMP280 barometer. This library can operate the BMP280 in either...

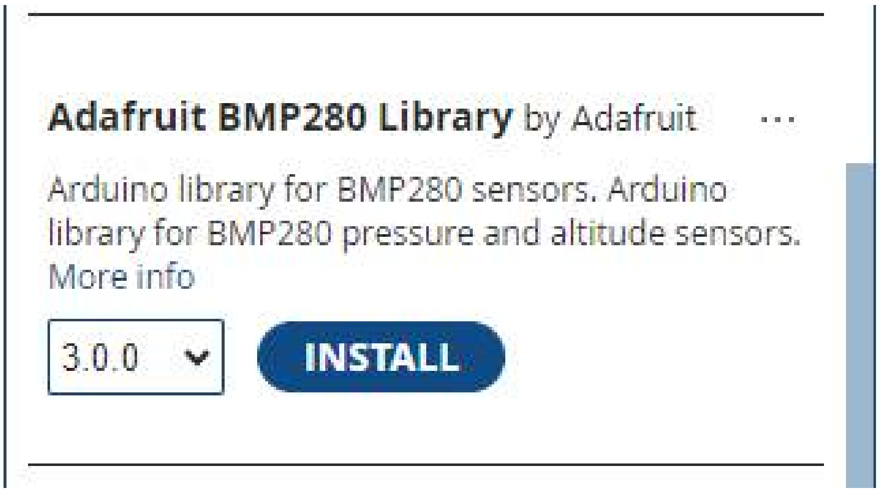
```
/**
Connect ESP32-S3 to ESP-NN
  Filename:      userwords.h
  Date:          04 feb 2026
  Updated:       04 feb 2026
  File Version:  1.0
  MCU:           ESP32-S3 WROOM only !!
  Forth:         ESP32forth all versions 7.0.7.21c+
  Copyright:     Marc PETREMANN
  Author:        Marc PETREMANN
  GNU General Public License
*/


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

```

Yes, it's available. It appears as the **Adafruit BMP280 Library** . Click **INSTALL** and this library will be loaded into the ARDUINO IDE.

At the bottom of this bookstore's description, there is "More Info":



Adafruit BMP280 Library by Adafruit

Arduino library for BMP280 sensors. Arduino library for BMP280 pressure and altitude sensors. More info

3.0.0 INSTALL

Clicking on this link will take you here:

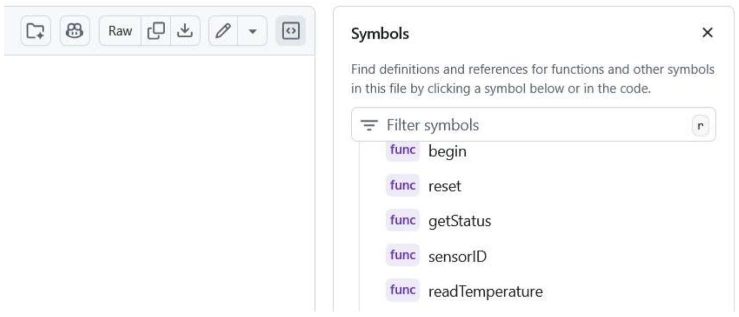
https://github.com/adafruit/Adafruit_BMP280_Library

The source files for this BMP280 library are located in the GITHUB repository.

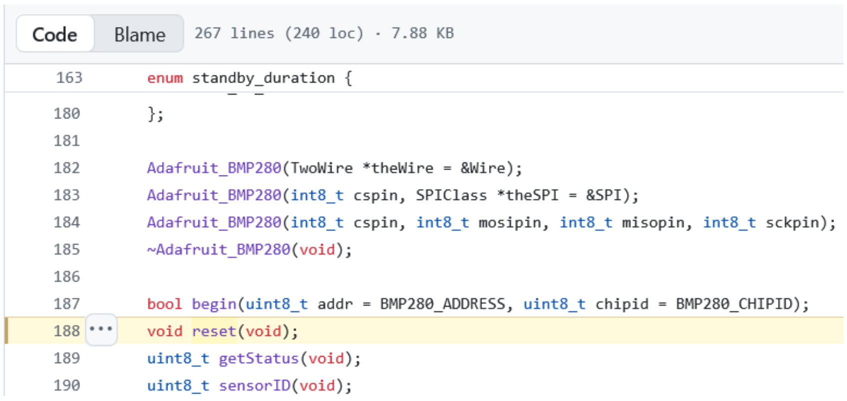
Our userwords.h file must contain this very first line of code:

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

On the GitHub repository, open the side panel. It shows the classes and functions available in this library:


Raw
Symbols
Find definitions and references for functions and other symbols in this file by clicking a symbol below or in the code.
Filter symbols
func begin
func reset
func getStatus
func sensorID
func readTemperature

Click on a function, for example reset:



```
enum standby_duration {
};

Adafruit_BMP280(TwoWire *theWire = &Wire);
Adafruit_BMP280(int8_t cspin, SPIClass *theSPI = &SPI);
Adafruit_BMP280(int8_t cspin, int8_t mosipin, int8_t misopin, int8_t sckpin);
~Adafruit_BMP280(void);

bool begin(uint8_t addr = BMP280_ADDRESS, uint8_t chipid = BMP280_CHIPID);
void reset(void);
uint8_t getStatus(void);
uint8_t sensorID(void);
```

Here, highlighted in yellow, we see the complete C definition of the `reset() code` . Now let's add these two lines to **userwords.h** :

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

While we're at it, we have two other words that are rather easy to define:

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

The first parameter of macro `X` is a string containing the word FORTH to be added to the ESP32forth dictionary.

The second parameter is a label that allows for an external call. This will be detailed later.

The third parameter contains the C code executed by the word defined by `X()` .

Here, we chose names for our dictionary that corresponded similarly to the names of C functions. We could just as easily have defined a word like FORTH `getBmp280Status` :

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

The Y macro

This is the simplest. It allows you to add a word containing only letters, numbers, and the underscore character. Example:

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

Settings:

- word FORTH to be defined, with letters, numbers and possibly underlined character;
- the C language code to be executed when the defined word is called.

All words defined by the Y macro are integrated into the `forth vocabulary` .

Macro X

If the word FORTH to be defined contains characters other than A..Z, a..z, 0..9 and is underlined, macro `X will be used` :

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

Settings:

- word FORTH to be defined, with letters, numbers and possibly underlined character;
- a call label, written only in capital letters and underlined characters. The label must be unique;
- the C language code to be executed when the defined word is called.

All words defined by macro X are integrated into the `forth vocabulary` .

The YV macro

Similar to the `Y macro` , but accepts as its first parameter the name of the vocabulary in which the word will be defined:

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

Here, the word `ledcWriteNote` will be defined in the `ledc vocabulary` .

The macro XV

Similar to macro `X` , but accepts as its first parameter the name of the vocabulary in which the word will be defined:

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

Here, the word `SPIFFS.begin` will be defined in the `SPIFFS vocabulary` .

The V macro

This macro is used to define a new vocabulary:

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

Here, we define the `HTTPClient vocabulary` .

Note : If you are not familiar with the vocabulary, define words by adding prefixes, for example:

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

In this code, we define the word `nn.add_elementwise_s8` . The prefix `nn.` refers to **Neural Network** , one of the Xtensa libraries. It's purely a naming convention, used by many programmers. We could just as easily have defined the word `add_s8` or even `addS8` .

The advantage of prefixing certain words is that it allows us to understand, by rereading their code, that the prefixed word refers to a specific context. Thus, in the `serial vocabulary` , we find these words:

```
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
```

In this vocabulary, we have two words: `` `readBytes` `` (in red), prefixed respectively by `` `serial` `` and `` `serial1` `` . These are indeed two distinct words. When we find them in source code, we immediately understand that ` `` Serial.readBytes` `` operates on UART0 port, while ` `` Serial1.readBytes` `` operates on UART1 port.

Note that this is not object-oriented programming. The syntax of FORTH words offers considerable freedom. As a reminder, a FORTH word is defined as any group of characters not separated by a space: `{..}` can be a FORTH word. Simply define it as FORTH.

Subprogram call

There are situations where it is necessary to process certain parameters through a user function rather than connecting to a library function:

```
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);) \
```

Here we define the word `esp_now_register_recv_cb` which refers to our user function `EspnowRegisterRecvCb` .

Passing the parameters

Passing parameters between ESP32forth and C language functions is quite tricky. At the slightest error, at best the word behaves differently than expected. At worst, the FORTH interpreter crashes!

Let's start with the simplest case.

Parameterless function call

Many library functions require no parameters and also return no data:

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

`OledCLS` word executes the `clearDisplay() function` from the oled library.

Calling a function that returns data

Let's look at the case of a function that returns data but doesn't use any input:

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

The data is pushed onto the FORTH stack by the `PUSH macro` . Here, the word OledAddr returns the address of a structure attached to the oled library.

Calling a function using a single piece of data

Let's now look at a FORTH word using a function that takes a single piece of data, but returns nothing on the FORTH stack:

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

`OledTextsize` word , which uses a parameter on the stack. Here, the parameter uses `n0` , which is an alias for `tos` . This definition is accepted.

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

The C code then calls **DROP,** which is defined in the ESP32forth source code:

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

Why call DROP? Actually, when `OledTextsize is executed` , in FORTH mode, we first push a parameter onto the stack. Example:

```
2 OledTextsize
```

If there was no `DROP` in the macro that defines `OledTextsize` , this value `2` would remain on the data stack.

Function call using multiple data

Let's start with a function that uses three pieces of data but returns no parameters:

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

Here we define the word `OLEDPixel` which uses three parameters:

- x which is the position at x
- y, which is the position y
- color which is the color of the pixel to be displayed on an OLED screen.

Example :

```
50 20 1 OLEDPixel \ displays bright pixel at position 50,20
```

To remove parameters 50, 20, and 10 from the FORTH stack, the C code could have been written as follows:

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

Here, the entire C code has been highlighted in blue. Note the presence of `DROP` three times. These successive `DROP statements` can be replaced by the `DROPn definition` used in the example written earlier.

To check the order of parameters in a macro for **ESP32forth** , you must follow a strict rule related to the stack structure (LIFO - Last In, First Out) used by Brad Nelson.

In the macro implementation:

- **`n0`** is always the top of the stack (**TOS** - Top Of Stack).
- **`n1`** is the element just below, and so on (`n2` , `n3` , ...).

If you write in FORTH: `param3 param2 param1 param0 myWord` The order in the C++ macro will be:

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

Save a result after sending parameters

Some functions take one or more parameters and then return a value. Here's an example that uses two parameters and pushes a result onto the stack:

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

The `` `begin` function `` in the `oled` library uses four parameters, but only uses the data `` `n0` `` and `` `n1` `` placed on the data stack. The result is injected into `` `n0` `` by `begin()`:

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

Order in which parameters are passed

Pay close attention to the order in which parameters are passed. To avoid creating convoluted code, it is strongly recommended to maintain the same parameter order in FORTH as that described in the C function. Let's take the function ` `esp_nn_add_elementwise_s8_esp32s3()` ` , which is a vector multiplication function for a neural network. Prototype of this function:

```
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);
```

If you want to check, this function is available here:
https://github.com/espressif/esp-nn/blob/596b08401a63da3a2e1b40868c442f582a99ae26/include/esp_nn_esp32s3.h

After analyzing the code, it appears that only four parameters can be used. Here is the definition via 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))
```

Here we defined a word `nn.add_elementwise_s8` . We could have made it shorter: `nn.add_s8`.

But for the project under consideration, it is preferable that the words FORTH stick as closely as possible to the function names used in the C language.

Incidentally, note the number of lines our C code occupies. This choice was made to have readable and easily modifiable code.

`nn.add_elementwise_s8` keyword has not yet been tested. It performs a vector addition of two fields of length `size` , pointed to by `n3` and `n2` . The result is stored in `n1` . This code is only usable on an ESP32-S3 board. This board has 16MB or 32MB of PSRAM, depending on the board model.

Preliminary conclusion

This document needs to be completed.

If you have any questions, notice any omissions or errors, or wish to provide additional information, you can contact me at: esp32forth@arduino-forth.com