

ESP32forth 帳本

版本 1.5 - 2024 年 1 月 6 日



作者

- Marc PETREMANN petremann@arduino-forth.com

合作者

- Bob EDWARDS
- Vaclav POSELT
- Thomas SCHREIN

內容

作者.....	1
合作者.....	1
介紹.....	6
翻譯幫助.....	6
ESP32 卡的發現.....	7
推介會.....	7
優點.....	7
ESP32 上的 GPIO 輸入/輸出.....	8
ESP32 週邊設備.....	10
不同的 ESP32 卡.....	11
ESP32forth 的最終安裝.....	12
ESP32 Wroom 32 卡.....	12
連接器板.....	13
ESP32 Wrover 板.....	13
連接器板.....	14
ESP32 S3 卡.....	14
連接器板.....	15
安裝 ESP32Forth	16
下載 ESP32forth.....	16
編譯並安裝 ESP32forth.....	16
ESP32 WROOM 的設置.....	18
開始編譯.....	18
修復上傳連線錯誤.....	20
為什麼在 ESP32 上使用 FORTH 語言程式設計?	23
前言.....	23
語言與應用之間的界限.....	23
FORTH 字是什麼?	24
一個字就是一個函數?	24
FORTH 語言與 C 語言的比較.....	25
與 C 語言相比, FORTH 可以讓您做什麼.....	25
但為什麼是堆疊而不是變數呢?	26
你確信嗎?	26
有沒有用 FORTH 寫的專業應用程式?	26
在 ESP32Forth 中使用數字.....	28
FORTH 解譯器的數字.....	28
輸入不同基數的數字.....	29
改變數值基數.....	29
二進制和十六進制.....	30
FORTH 資料堆疊上數字的大小.....	31
記憶體存取和邏輯運算.....	32
使用 ESP32Forth 的真正 32 位元 FORTH	35

資料堆疊上的值.....	35
記憶體中的值.....	35
根據資料大小或類型進行文字處理.....	36
結論.....	37
評論和澄清.....	38
編寫可讀的 FORTH 程式碼.....	38
原始碼縮排.....	39
評論.....	40
堆疊評論.....	40
註解中堆疊參數的意義.....	40
單字定義 單字註釋.....	41
文字評論.....	41
註釋在原始碼開頭.....	42
診斷和調整工具.....	42
反編譯器.....	42
記憶體轉儲.....	43
電池監視器.....	43
字典/堆疊/變數/常數.....	45
展開字典.....	45
字典管理.....	45
堆疊和逆波蘭表示法.....	45
處理參數堆疊.....	46
返回堆疊及其用途.....	47
記憶體使用情況.....	47
變數.....	47
常數.....	47
偽常數值.....	48
記憶體分配的基本工具.....	48
終端機上的文字顏色和顯示位置.....	49
端子的 ANSI 編碼.....	49
文字著色.....	49
顯示位置.....	51
ESP32Forth 的局部變數.....	53
介紹.....	53
假堆疊註釋.....	53
對局部變數的操作.....	54
ESP32forth 的資料結構.....	56
前言.....	56
FORTH 中的表格.....	56
一維 32 位元資料數組.....	56
表格定義字.....	57
在表中讀取和寫入.....	57
管理虛擬螢幕的實際範例.....	58
複雜結構的管理.....	61
精靈的定義.....	63

ESP32forth 的實數.....	66
真正的 ESP32forth.....	66
ESP32forth 的實數精度.....	66
實數常數和變數.....	67
實數的算術運算符.....	67
實數的數學運算符.....	67
實數上的邏輯運算符.....	68
整數 ↔ 實數轉換.....	68
將麵包板適應 ESP32 板.....	69
ESP32 測試板.....	69
建造適合 ESP32 板的麵包板.....	69
RECORDFILE 和 FORTH 專案管理.....	72
將 RECORDFILE 儲存在 autoexec.fs 檔案中.....	72
使用 autoexec.fs 檔案的修改內容.....	73
使用 ESP32forth 分解項目.....	74
範例專案.....	74
黑盒子的概念.....	76
新增 SPI 庫.....	78
ESP32forth.ino 檔案的更改.....	79
第一次修改.....	79
第二次修改.....	79
第三次修改.....	79
第四次修改.....	80
與 MAX7219 顯示模組通信.....	80
找到 ESP32 板上的 SPI 端口.....	81
MAX7219 顯示模組上的 SPI 連接器.....	81
SPI 埠軟體層.....	82
安裝 HTTP 用戶端.....	84
編輯 ESP32forth.ino 文件.....	84
HTTP 用戶端測試.....	85
ESP32forth 詞彙詳細內容.....	88
Version v 7.0.7.15.....	88
FORTH.....	88
asm.....	89
bluetooth.....	89
editor.....	90
ESP.....	90
httpd.....	90
insides.....	90
internals.....	90
interrupts.....	91
ledc.....	91
oled.....	91
registers.....	91
riscv.....	91
rtos.....	92

SD.....	92
SD_MMC.....	92
Serial.....	92
sockets.....	92
spi.....	92
SPIFFS.....	92
streams.....	92
structures.....	92
tasks.....	92
telnetd.....	93
visual.....	93
web-interface.....	93
WiFi.....	93
Wire.....	93
xtensa.....	93
資源.....	95
用英語.....	95
法語.....	95
GitHub.....	95

介紹

自 2019 年以來，我管理了幾個致力於 ARDUINO 和 ESP32 卡的 FORTH 語言開發的網站，以及 eForth 網頁版：

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

這些網站有兩種語言版本：法語和英語。每年我都會支付主網站 **arduino-forth.com** 的託管費用。

遲早——而且盡可能晚——我將不再能夠確保這些網站的可持續性。其後果將是這些網站傳播的訊息消失。

這本書是我網站內容的彙編。它是從 Github 儲存庫免費分發的。這種分發方法將比網站具有更大的可持續性。

順便說一句，如果這些頁面的一些讀者希望做出貢獻，我們歡迎：

- 建議章節；
- 報告錯誤或建議更改；
- 幫忙翻譯...

翻譯幫助

谷歌翻譯可以让你輕鬆翻譯文本，但會出現錯誤。所以我請求幫助來糾正翻譯。

在實務中，我提供了已經翻譯成 LibreOffice 格式的章節。如果您想幫助完成這些翻譯，您的角色只是更正並返回這些翻譯。

修改一章只需要很少的時間，從一個到幾個小時不等。

聯絡我: petremann@arduino-forth.com

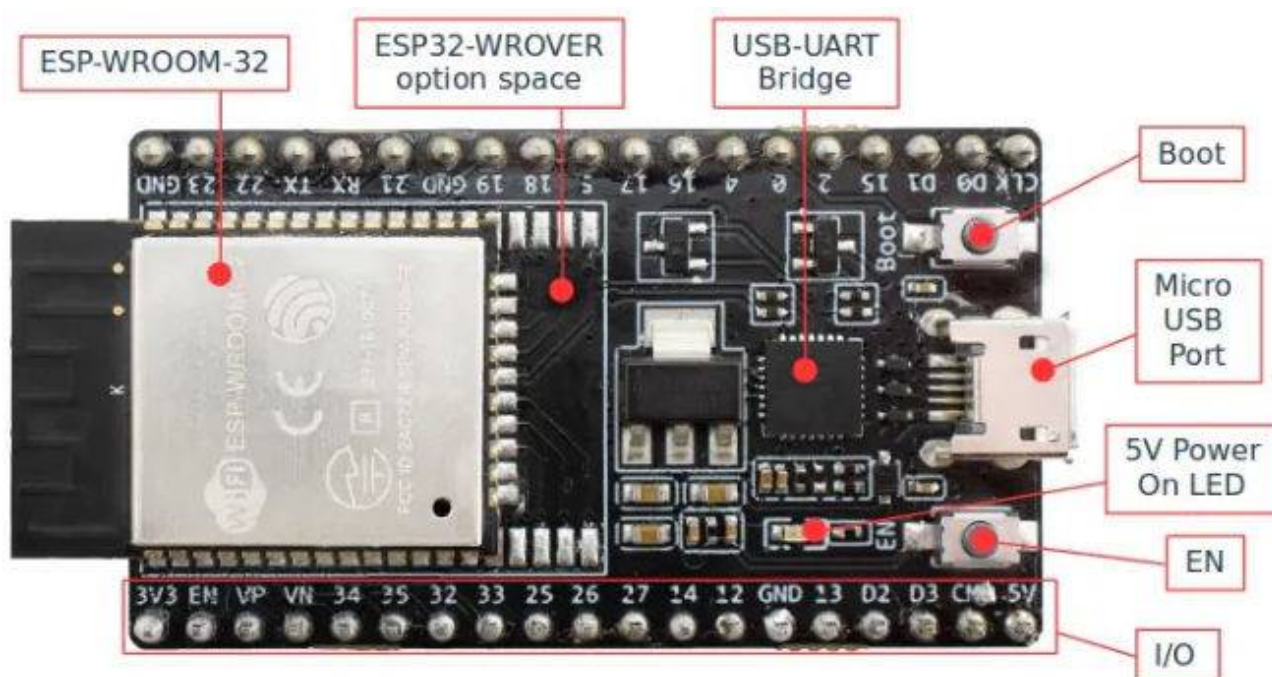
ESP32 卡的發現

推介會

ESP32 板不是 ARDUINO 板。然而，開發工具利用了 ARDUINO 生態系統的某些元素，例如 ARDUINO IDE。

優點

就可用連接埠數量而言，ESP32 卡位於 ARDUINO NANO 和 ARDUINO UNO 之間。基本型號有 38 個連接器：



ESP32 設備包括：

- 18 通道類比數位轉換器 (ADC)
- 3 個 SPI 接口
- 3 個 UART 接口
- 2 個 I2C 接口
- 16 個 PWM 輸出通道
- 2 個數位類比轉換器 (DAC)
- 2 個 I2S 接口

- 10 個電容感應 GPIO

ADC（類比數位轉換器）和 DAC（數位類比轉換器）功能被指派給特定的靜態引腳。但是，您可以決定哪些引腳是 UART、I2C、SPI、PWM 等。您只需在程式碼中分配它們即可。這要歸功於 ESP32 晶片的複用功能。

大多數連接器都有多種用途。

但 ESP32 板的與眾不同之處在於，它標配了 WiFi 和藍牙支持，而 ARDUINO 板僅以擴展的形式提供這些功能。

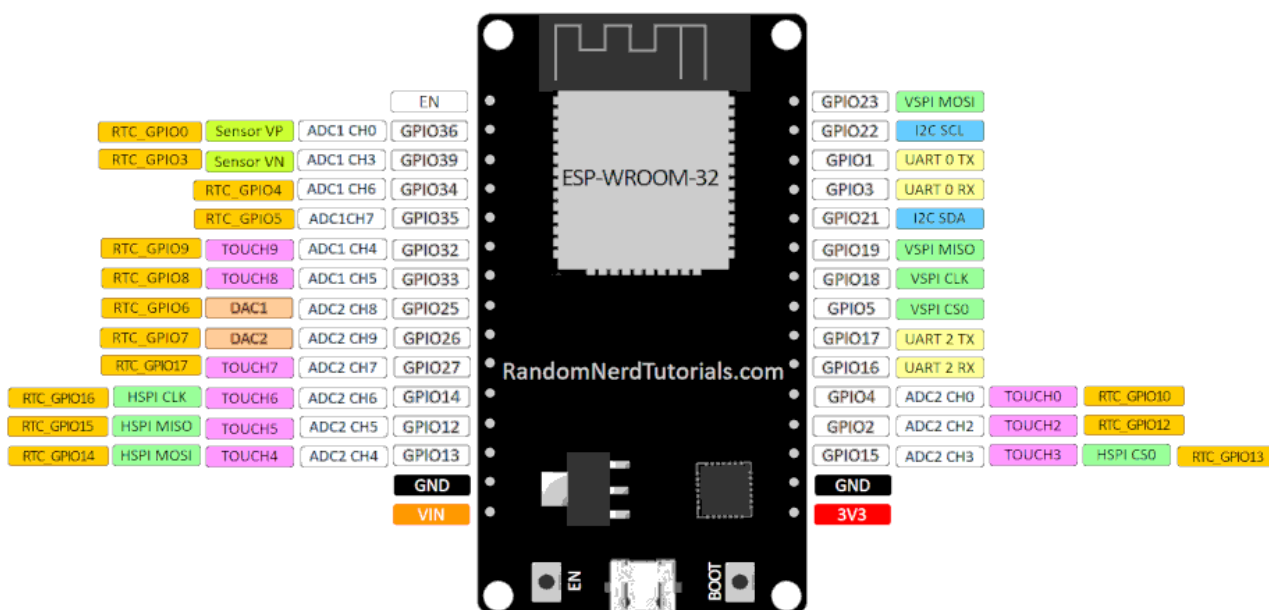
ESP32 上的 GPIO 輸入/輸出

以下是 ESP32 卡的照片，我們將從中解釋不同 GPIO 輸入/輸出的作用：



GPIO I/O 的位置和數量可能會根據卡品牌而變化。如果是這樣的話，那麼只有實體圖上出現的指示才是真實的。如圖，底行由左至右：

CLK、SD0、SD1、G15、G2、G0、G4、G16.....G22、G23、GND。



在此圖中，我們看到底部行以 **3V3** 開頭，而在照片中，此 **I/O** 位於頂部行的末尾。因此，不要依賴圖表，而是仔細檢查實體 **ESP32** 卡上的周邊設備和組件的正確連接，這一點非常重要。

基於 **ESP32** 的開發板除了電源接腳外，一般還有 **33** 個接腳。一些 **GPIO** 引腳具有一些特殊的功能：

通用輸入輸出介面	可能的名稱
6	SCK/CLK
7	SCK/CLK
8	SDO/SD0
9	SDI/SD1
10	SHD/SD2
11	CSC/CMD

如果你的 **ESP32** 卡有 **I/O GPIO6、GPIO7、GPIO8、GPIO9、GPIO10、GPIO11**，你絕對不應該使用它們，因為它們連接到 **ESP32** 的快閃記憶體。如果您使用它們，**ESP32** 將無法運作。

GPIO1(TX0) 和 **GPIO3(RX0)** **I/O** 用於透過 **USB** 連接埠與電腦進行 **UART** 通訊。如果您使用它們，您將無法再與該卡通訊。

GPIO36(VP)、**GPIO39(VN)**、**GPIO34**、**GPIO35** **I/O** 只能用作輸入。它們也沒有內建的內部上拉和下拉電阻。

EN 端子可讓您透過外部導線控制 ESP32 的點火狀態。它連接到卡上的 EN 按鈕。當 ESP32 開啟時，電壓為 3.3V。如果我們將該引腳接地，ESP32 將關閉。當 ESP32 位於盒子中並且您希望能夠透過開關打開/關閉它時，您可以使用它。

ESP32 週邊設備

為了與模組、感測器或電子電路交互，ESP32 與任何微控制器一樣，擁有大量週邊設備。它們的數量比經典 Arduino 板上的數量還要多。

ESP32 有以下週邊：

- 3 個 UART 接口
- 2 個 I2C 接口
- 3 個 SPI 接口
- 16 個 PWM 輸出
- 10 個電容式感測器
- 18 個類比輸入 (ADC)
- 2 個 DAC 輸出

ESP32 在其基本操作過程中已經使用了一些週邊設備。因此，每個設備可能的介面較少。

不同的 ESP32 卡

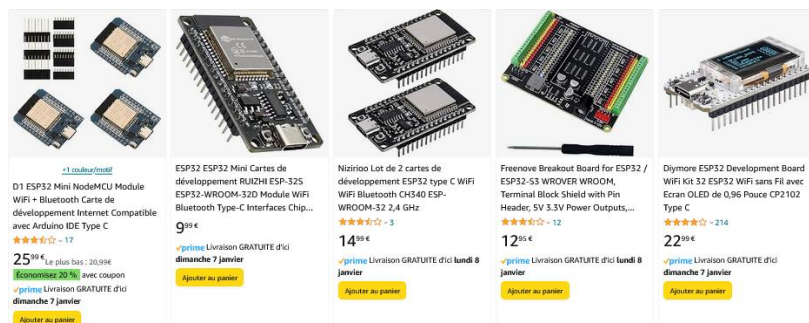


Figure 1: un grand choix de cartes ESP32 sur AMAZON

如果您去線上銷售網站訂購 ESP32 卡，您最終可能會得到非常多的卡選擇。

因此，出現了幾個問題來指導選擇：

- 哪塊板可以承載 ESP32forth?
- 哪些卡最適合我的專案?
- 我對某個專案的預算是多少?

如果普通 ESP32 卡的價格仍然可以承受，某些型號的價格可能會飆升。如果您的目標是先進行小型實驗，請從簡單的 ESP32 板開始。要正確進行實驗，您將需要：

- 測試板，至少取 10 個。每張 ESP32 卡允許兩個測試板；
- 靈活的杜邦型連接器；
- LED、電阻等
- 週邊設備：OLED 顯示器、LCD、繼電器、同步或普通馬達、伺服馬達等。
- USB 線連接 PC 和 ESP32 卡。建議使用 USB 集線器。若 USB 埠意外注入電流，集線器的 USB 埠將先於 PC 的 USB 埠損壞；

價格約 50 歐元（或美元），就有現成的套件，包括 ESP32 卡以及週邊設備和組件。

沒有套件是完整的。如果您正在進行實驗，理想的情況是訂購套件，然後是幾張 ESP32 卡（至少 4 個）、一系列測試板、扁平帶、電池電源等...

在開始雄心勃勃的專案之前，例如透過 3G/4G/5G 網路進行遠端控制、視訊分析等。從簡單的 C 或 FORTH 實驗開始。

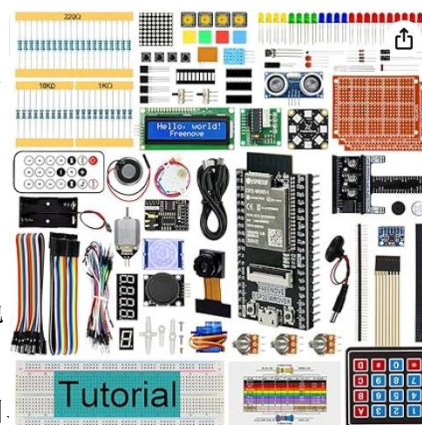


Figure 2: kit ESP32

ESP32forth 的最終安裝

不！ESP32forth 在 ESP32 卡上的安裝不是永久的！如果您在一塊或多塊 ESP32 板上安裝了 ESP32Forth，您可以在編譯 **ino** 擴充檔的內容後，輕鬆地從任何 C 原始程式碼下載二進位程式碼到您的 ESP32 板上，並放棄 FORTH 程式設計。

但是，冒著為 ESP32forth 做廣告的風險，許多「創客」最終決定使用 FORTH 語言進程式設計。舉一個例子，**0033mer** 的 YouTube 頻道：

<https://www.youtube.com/@0033mer>

他是 YouTube 上最多產的 FORTH 貢獻者之一。儘管他很少使用 ESP32，但他的大部分貢獻都使用 FORTH 語言。

使用 FORTH 語言進程式設計需要智力勞動。這種努力並沒有白費，因為它帶來了某些可以在其他程式語言中使用的良好實踐。

FORTH 是唯一可以安裝在電子卡上的程式語言，整合了解譯器、編譯器、彙編器、SPIFFS 檔案系統，並且具有很大的開發空間。

ESP32 卡是非常罕見的卡片之一，還具有透過 WiFi 或藍牙進行串行通訊功能（透過 USB 連接器的 UART0 連接埠）。大多數卡還具有眾多多功能 GPIO 連接埠：邏輯、類比、PWM、UART、SPI、I2C 輸入和輸出等。所有這些都需要一個或兩個接近 160Mhz 的處理器，或比普通 ARDUINO 卡快 10 倍。。

最後，ARDUINO 的大部分 C 庫都可以在 ESP32 上使用。有些可以從 ESP32forth 存取。

ESP32 Wroom 32 卡

ESP32 Wroom 是樂鑫 ESP 卡系列的最新成員。這是一款特別時尚的開發板系列，因為其價格低、功耗低且尺寸小，使其成為執行小型物聯網專案的理想產品。

- ESP-WROOM-32 處理器
- 無線區域網路 802.11 b/g/n
- 藍牙 4.2/BLE
- Tensilica L108 32 位元 160 MHz 處理器
- 512 KB SRAM 和 16 MB 快閃記憶體
- 32 個數字 I/O 腳位(3.3V)
- 6 個類比數位引腳
- 3 個 UART、2 個 SPI、2 個 I2C
- USB 轉 UART CP2102 接口

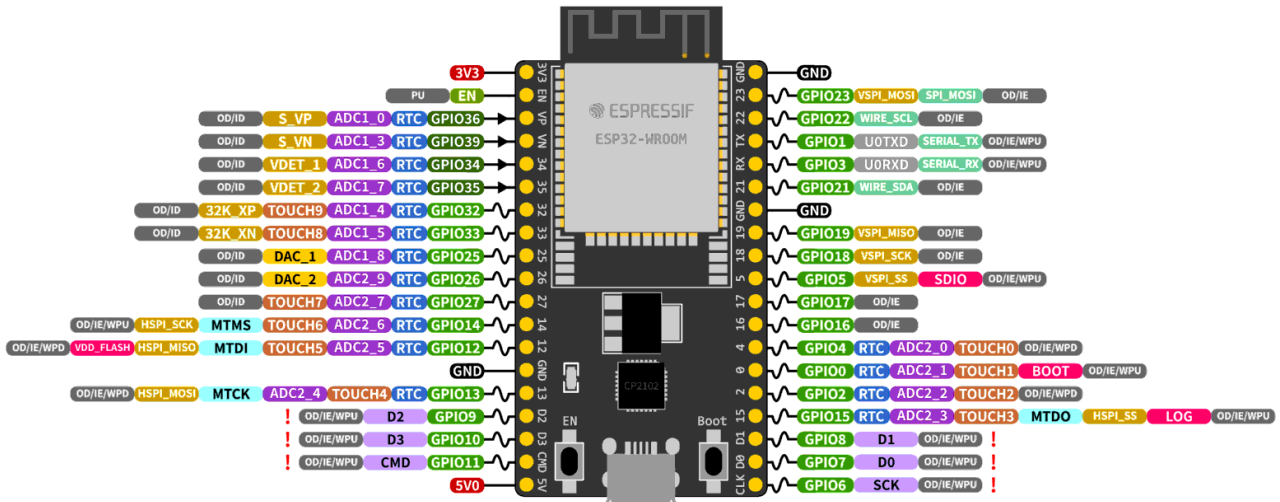


Figure 3: carte ESP32 Wroom 32

如果您為此開發板編譯 ESP32forth，請在 ARDUINO IDE 上考慮以下參數：工具 → 開發板 → ESP32 → ESP32 開發模組：

- 木板: **ESP32 開發模組**
 分區方案: **無 OTA (2M APP、2M SPIFFS) ←非預設**
 上傳速度: 921600
CPU 頻率: 240MHz
 快閃記憶體頻率: 80MHz
 快閃記憶體模式: QIO
 快閃記憶體大小: 4MB (32Mb)
 核心偵錯等級: 無
PSRAM: 停用

連接器板



ESP32 Wrover 板

樂鑫 ESP32-WROVER MCU 模組是功能強大的通用 Wi-Fi/BT/BLE MCU 模組，面向廣泛的應用。

這些模組的目標應用範圍從低功耗感測器網路到最嚴苛的任務，例如語音編碼、音樂串流和 MP3 解碼。

ESP32-WROVER 模組使用 PCB 天線，而 ESP32-WROVER-I 使用 IPEX 天線。這些模組具有外部 4 MB SPI 快閃記憶體、外部 4 MB PSRAM 和 32 Mbit SPI PSRAM。

如果您為此開發板編譯 ESP32forth，請在 ARDUINO IDE 上考慮以下參數：工具 → 開發板 → ESP32 → ESP32 開發模組：

- 木板: **ESP32 開發模組**
 分區方案: **無 OTA (2M APP、2M SPIFFS) ←非預設**
 上傳速度: 921600
CPU 頻率: 240MHz
 快閃記憶體頻率: 80MHz
 快閃記憶體模式: QIO
 快閃記憶體大小: 4MB (32Mb)

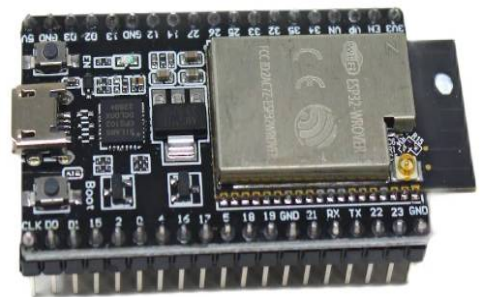
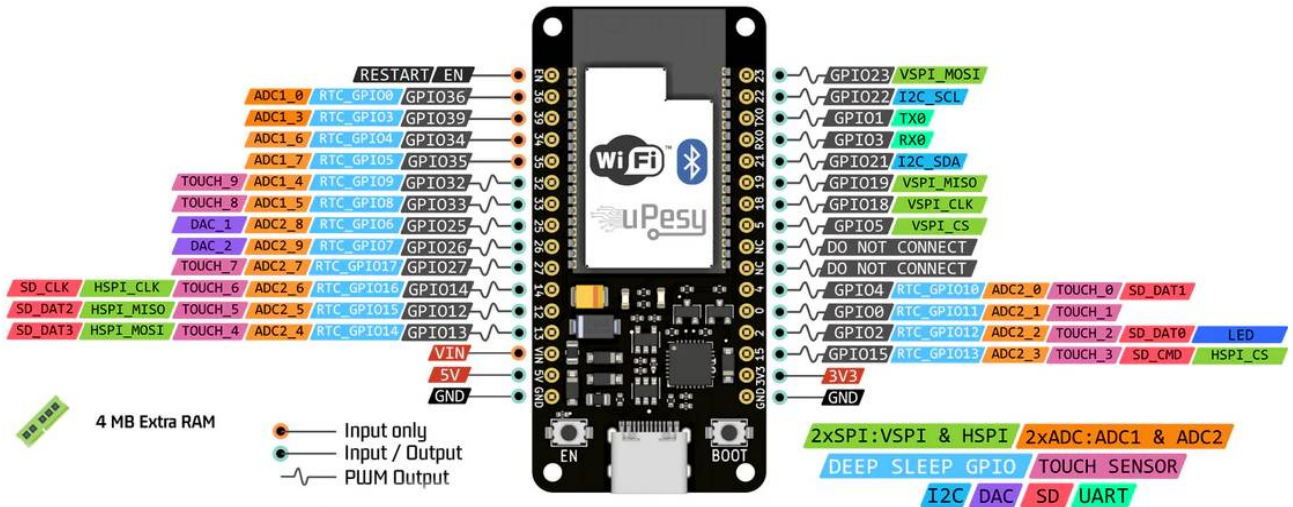


Figure 4: carte ESP32 Wrover

核心偵錯等級：無

PSRAM：啟用

連接器板



ESP32 S3 卡

ESP32-S3 是一款基於 Espressif ESP32-S3-WROOM-2 微控制器的開發板，具有 WiFi 和藍牙低功耗介面。

- Xtensa LX7 雙核心 32 位元微處理器
- PSRAM 記憶體：8 MB
- SRAM 記憶體：512 KB
- ROM 記憶體：384 KB
- SRAM 記憶體 (RTC)：16 KB SPI
- 快閃記憶體：32 MB
- WiFi 介面：802.11 b/g/n 2.4 GHz
- BLE 5 網網路狀介面

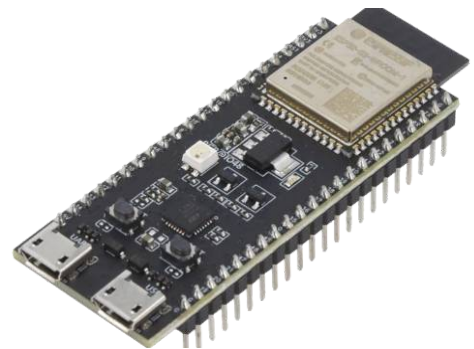


Figure 5: carte ESP32 S3

如果您為此開發板編譯 ESP32forth，請在 ARDUINO IDE 上考慮以下參數：工具 → 開發板 → ESP32 → ESP32 開發模組：

- 木板：ESP32S2 開發模組
- 分區方案：無 OTA (2M APP, 2M SPIFFS) ←非預設
- 上傳速度：921600
- USB CDC 啟動時：停用
- USB 韌體 MSC 啟動時：停用
- USB DFU 啟動時：停用
- 上傳模式：UART0
- CPU 頻率：240MHz
- Flash 頻率：80MHz
- Flash 模式：QIO
- 快閃記憶體大小：4MB (32Mb)

PSRAM: 啟用

ESP32-S3-WROOM-1

3V3
3V3
RST

ADC1_3 TOUCH4 RTC GPIO4
ADC1_4 TOUCH5 RTC GPIO5
ADC1_5 TOUCH6 RTC GPIO6
ADC1_6 TOUCH7 RTC GPIO7

XTAL_32K_P ADC2_4 U0RTS RTC GPIO15
XTAL_32K_N ADC2_5 U0CTS RTC GPIO16
ADC2_6 U1TXD RTC GPIO17
CLK_OUT3 ADC2_7 U1RXD RTC GPIO18
ADC1_7 TOUCH8 RTC GPIO18
JTAG ADC1_2 TOUCH3 RTC GPIO13
LOG GPIO46

FSPIHD SUBSPIHD ADC1_8 TOUCH9 RTC GPIO9
FSPICS0 SUBSPICS0 FSPIIO4 ADC1_9 TOUCH10 RTC GPIO10
FSPID SUBSPID FSPIIO5 ADC2_0 TOUCH11 RTC GPIO11
FSPICLK SUBSPICLK FSPIIO6 ADC2_1 TOUCH12 RTC GPIO12
FSPIQ SUBSPIQ FSPIIO7 ADC2_2 TOUCH13 RTC GPIO13
FSPIWP SUBSPIWP FSPIDQS ADC2_3 TOUCH14 RTC GPIO14

5V0
GND

UART

USB

GND

TX
RX

GPIO1 RTC TOUCH1 ADC1_0
GPIO2 RTC TOUCH2 ADC1_1

MTMS GPIO42
MTDI GPIO41 CLK_OUT1
MTDO GPIO40 CLK_OUT2
MTCK GPIO39 CLK_OUT3 SUBSPICS1

GPIO38 FSPIWP SUBSPIWP
GPIO37 SPIDQS FSPIQ SUBSPIQ
GPIO36 SPIIO7 FSPICLK SUBSPICLK
GPIO35 SPIIO6 FSPID SUBSPID

GPIO0 BOOT
GPIO45 VSPI

GPIO48 SPICLK_N RGB_LED
GPIO47 SPICLK_P

GPIO21 RTC

USB_D+ GPIO20 RTC U1CTS ADC2_9 CLK_OUT1
USB_D- GPIO19 RTC U1RTS ADC2_8 CLK_OUT2

GND

RGB@IO48

CP2102

BOOT

RESET

安裝 ESP32Forth

下載 ESP32forth

第一步包括恢復 ESP32forth 的 C 語言原始碼。最好使用最新版本：

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

下載檔案的內容：

```
ESP32forth-7.0.xx
  ESP32forth
    自述文件.txt
    esp32forth.ino
    選修的
      SPI-flash.h
      串行藍牙.h
      ... ETC...
```

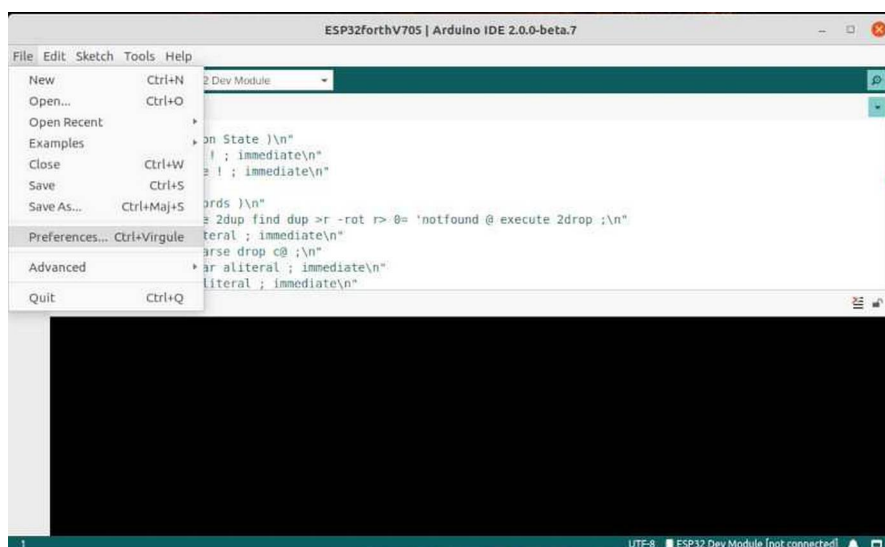
編譯並安裝 ESP32forth

esp32forth.ino 檔案複製到工作目錄中。可選目錄包含允許擴展 ESP32forth 的檔案。對於我們第一次建置和上傳 ESP32forth，不需要這些檔案。

要編譯 ESP32forth，您的電腦上必須已安裝 ARDUINO IDE：

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

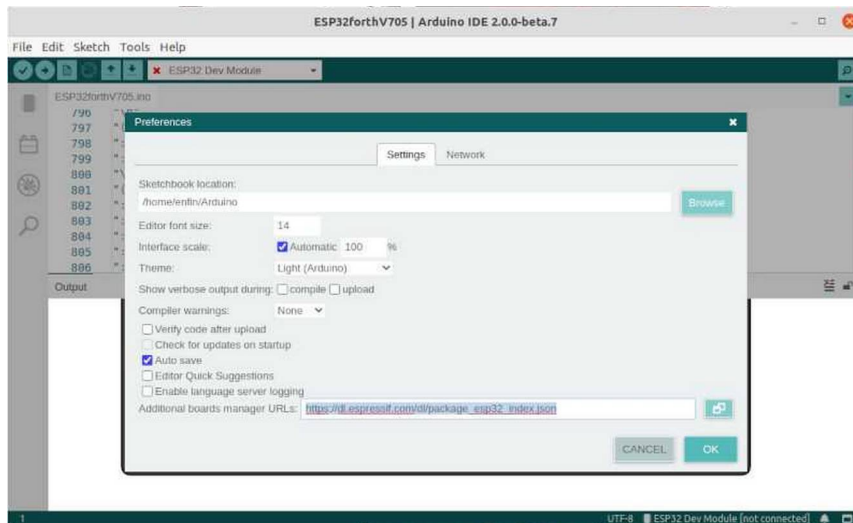
安裝 ARDUINO IDE 後，啟動它。ARDUINO IDE 已經開放，這裡是 2.0¹版本。點擊檔案並選擇首選項：



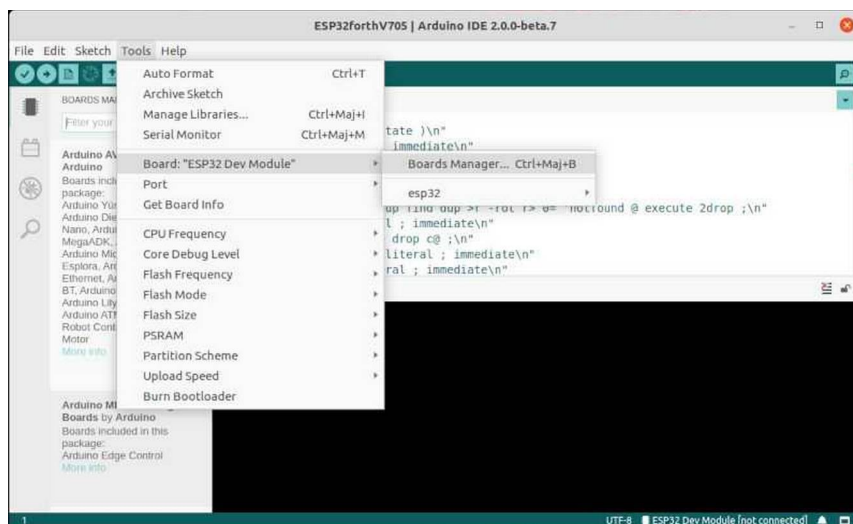
在出現的視窗中，轉到標記為「附加板管理器 URL」的輸入框，然後輸入以下行：

1 關於 ESP32forth 版本的注意事項 - 所謂的穩定版本 7.0.6.19 需要正確編譯 Espressif 板庫 1.0.6，最新版本 7.0.7.15 需要庫 2.0.x。

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



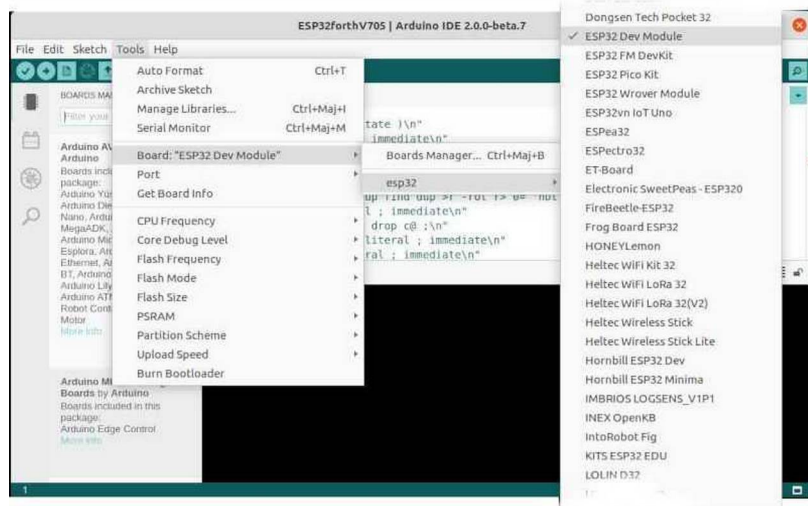
接下來，點擊“工具”並選擇“板：”。



此選擇將為您提供 ESP32 軟體包的安裝。接受此安裝。

然後您應該能夠存取 ESP32 卡的選擇：

ESP32 開發模組板選型：



ESP32 WROOM 的設置

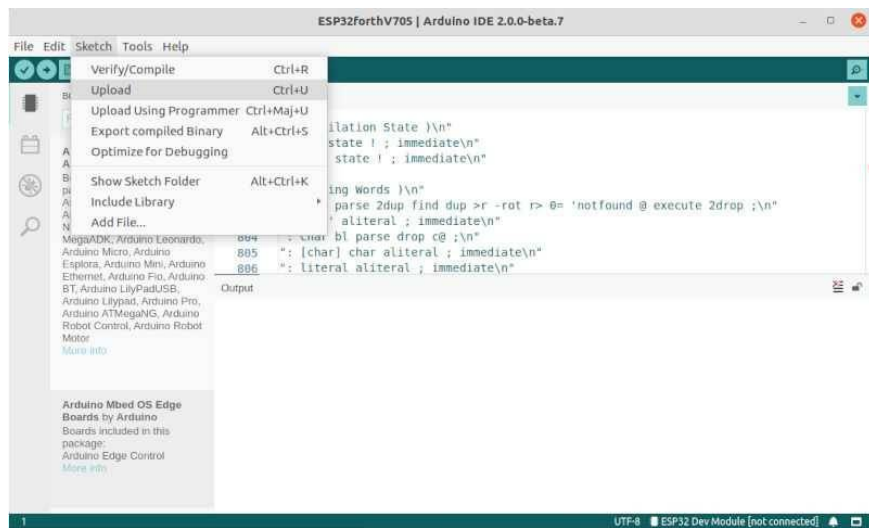
以下是編譯 ESP32forth 之前所需的其他設定。再次點選 “工具” 存取設定：

```
-- 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
```

開始編譯

剩下的就是編譯 ESP32forth。透過 *File* 和 *Open* 載入原始碼。

假設您的 ESP32 板已連接到 USB 連接埠。點選 *Sketch* 並選擇 *Upload* 開始編譯：



如果一切正常，您應該將二進位代碼自動傳輸到 ESP32 板中。如果編譯沒有錯誤，但出現傳輸錯誤，請重新編譯 **esp32forth.ino** 檔案。傳輸時，按下 ESP32 板上標示 **BOOT** 的按鈕。這將使該卡可用於傳輸 ESP32forth 二進位代碼。

影片中 ARDUINO IDE 的安裝與配置：

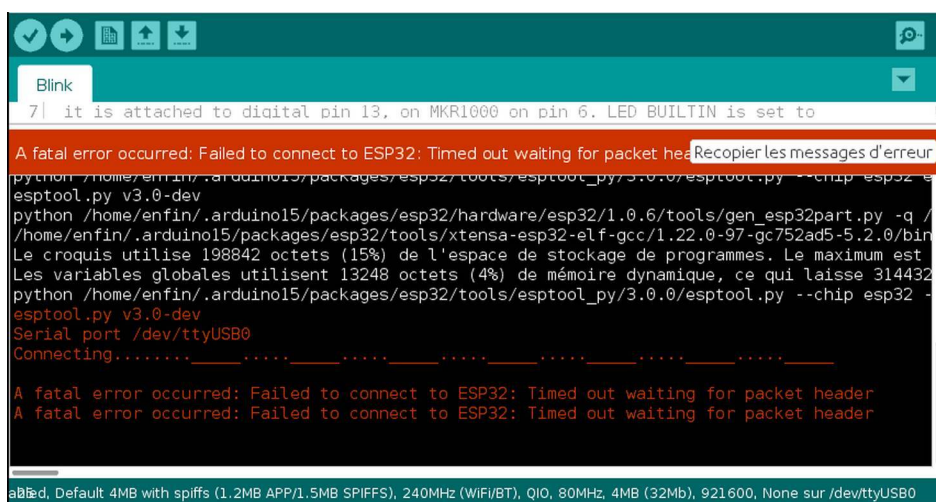
- Windows: <https://www.youtube.com/watch?v=2AZQfieHv9g>
- Linux: https://www.youtube.com/watch?v=JeD3nz0_nc

• 修復上傳連線錯誤

了解如何修復在嘗試一次將新程式碼上傳到 ESP32 卡時發生的致命錯誤：「無法連接到 ESP32：等待資料包標頭逾時」。

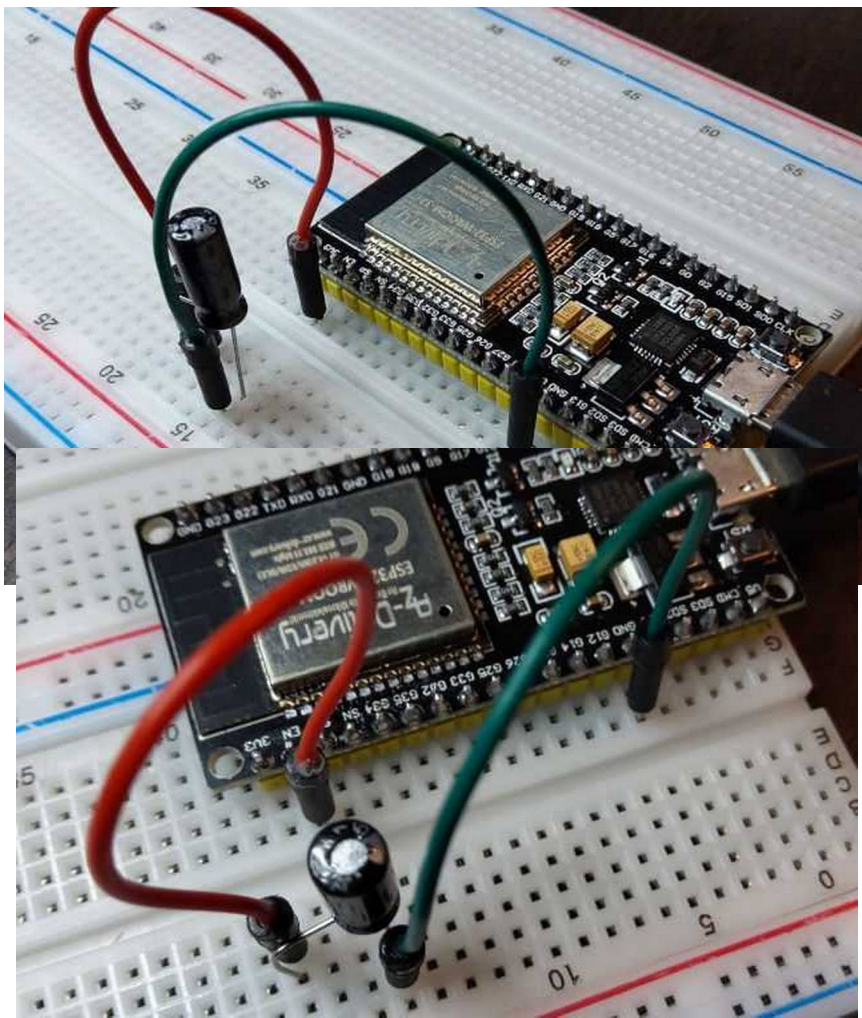
下載新程式碼時，某些 ESP32 開發板（請參閱最佳 ESP32 板）不會自動進入快閃記憶體/上傳模式。

這表示當您嘗試將新草圖上傳到 ESP32 開發板時，ARDUINO IDE 無法連線到您的開發板，並且您會收到以下錯誤訊息：



```
7| 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  
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.....  
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  
added, Default 4MB with spiiffs (1.2MB APP/1.5MB SPIFFS), 240MHz (WiFi/BT), QIO, 80MHz, 4MB (32Mb), 921600, None sur /dev/ttyUSB0
```

為了讓 ESP32 板自動切換到 flash/下載模式，我們可以在 EN 和 GND 接腳之間連接 10uF 電解電容：



只有當您處於從 **ARDUINO IDE** 上傳 **ESP32forth** 的階段時，才需要進行此操作。一旦 **ESP32forth** 安裝在 **ESP32** 板上，就不再需要使用該電容器。

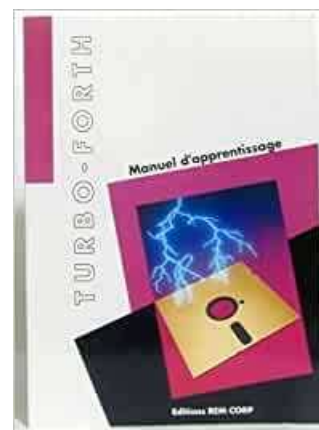
為什麼在 ESP32 上使用 FORTH 語言程式設計？

前言

我自 1983 年以來一直使用 FORTH 進程式設計。我於 1996 年停止使用 FORTH 進程式設計。但我從未停止監視這種語言的演變。我於 2019 年重新開始在 ARDUINO 上使用 FlashForth 和 ESP32forth 進程式設計。

我是幾本有關 FORTH 語言的書籍的合著者：

- Introduction au ZX-FORTH (ed Eyrolles - 1984 - ASIN:B0014IGOXO)
- Tours de FORTH (ed Eyrolles - 1985 - ISBN-13: 978-2212082258)
- FORTH pour CP/M et MSDOS (ed Loisetech - 1986)
- TURBO-Forth, manuel d'apprentissage (ed Rem CORP - 1990)
- TURBO-Forth, guide de référence (ed Rem CORP - 1991)



使用 FORTH 語言程式設計一直是我的愛好，直到 1992 年，一家汽車行業分包商公司的經理聯繫了我。他們對 C 語言軟體開發關心，需要訂購一台工業自動機。

該公司的兩位軟體設計人員以 C 語言進程式設計：準確地說是來自 Borland 的 TURBO-C。而且他們的程式碼不夠緊湊和快速，無法適應 64 KB 的 RAM 記憶體。那是 1992 年，快閃記憶體類型擴充還不存在。在這 64 KB RAM 中，我們必須容納 MS-DOS 3.0 和應用程式！

一個月來，C 語言開發人員一直在各個方向上扭轉這個問題，甚至使用 SOURCER（反彙編器）進行逆向工程，以消除可執行程式碼中非必要的部分。

我分析了向我提出的問題。從頭開始，我獨自一人在一周內創建了一個符合規範、完美運行的原型。從 1992 年到 1995 年的三年時間裡，我創建了該應用程式的多個版本，並在多家汽車製造商的組裝線上使用。

語言與應用之間的界限

所有程式語言共享如下：

- 解釋器和可執行原始程式碼：BASIC、PHP、MySQL、JavaScript 等...該應用程式包含在一個或多個檔案中，必要時將對其進行解釋。系統必須永久託管執行原始碼的解釋器；
- 編譯器和/或組譯器：C、Java 等。有些編譯器生成機器碼，也就是說可以在系統上專門執行。其他的，像是 Java，在 Java 虛擬機器上編譯可執行程式碼。

FORTH 語言是個例外。這包括：

- 能夠執行 FORTH 語言中任何單字的解釋器
- 能夠擴展 FORTH 單字字典的編譯器。

FORTH 字是什麼？

FORTH 單字指定由 ASCII 字元組成並可用於解釋和/或編譯的任何字典表達式：單字可讓您列出 FORTH 字典中的所有單字。

某些 FORTH 單字只能在編譯中使用：例如 **if else then** 。

對於 FORTH 語言，基本原則是我們不創建應用程式。在 FORTH 中，我們正在擴展字典！您定義的每個新單字都會與 FORTH 啟動時預先定義的所有單字一樣成為 FORTH 字典的一部分。例：

```
: 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
;
```

我們建立兩個新單字： **typeToLoRa** 和 **typeToTerm** ，這將完成預先定義單字的字典。

一個字就是一個函數？

是和不是。事實上，一個單字可以是一個常數、一個變數、一個函數…在我們的例子中，有以下序列：

```
: typeToLoRa ...代碼... ;
```

在 C 語言中會有等價的：

```
void typeToLoRa() { ...程式碼... }
```

在 FORTH 語言中，語言和應用之間沒有限制。

在 FORTH 中，與 C 語言一樣，您可以使用新單字定義中已定義的任何單字。

是的，但是為什麼是 FORTH 而不是 C？

我正期待著這個問題。

在 C 語言中，一個函數只能透過主函數 **main()** 來存取。如果該功能整合了多個附加功能，則在程式發生故障時很難發現參數錯誤。

相反，使用 FORTH 可以透過解釋器執行任何預先定義或由您定義的單詞，而無需遍歷程式的主單字。

FORTH 解譯器可透過終端機類型程式以及 ESP32 卡與 PC 之間的 USB 連結在 ESP32 卡上立即存取。

用 FORTH 語言編寫的程式的編譯是在 ESP32 卡中進行的，而不是在 PC 上進行的。沒有上傳。例：

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


此定義透過複製/貼上的方式傳輸到終端中。 FORTH 解釋器/編譯器將解析流並編譯新單字 **>gray** 。

>gray 的定義中，我們看到序列 **dup 2/ xor** 。要測試此序列，只需在終端機中鍵入它即可。要運行 **>gray** ，只需在終端機中輸入該單詞，前面加上要轉換的數字。

FORTH 語言與 C 語言的比較

這是最不喜歡的部分。我不喜歡將 FORTH 語言與 C 語言進行比較。但由於幾乎所有開發人員都使用 C 語言，所以我將嘗試這個練習。

用 C 語言 **if()** 進行的測試：

```
if(j > 13){           // 如果接收到所有位
    rc5_ok = 1;        // 解碼過程正常
    detachInterrupt(0); // 禁止外部中斷 (INT0)
    return;
}
```

用 FORTH 語言的 **if** 進行測試（程式碼片段）：

```
var-j @ 13 >          \ 如果接收到所有位
  if
    1 rc5_ok !        \ 解碼過程正常
    di                \ 禁用外部中斷 (INT0)
    exit
  then
```

下面是 C 語言中暫存器的初始化：

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

FORTH 語言中的相同定義：

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

與 C 語言相比，FORTH 可以讓您做什麼

我們知道 FORTH 可以立即存取字典中的所有單詞，但不僅限於此。透過解釋器，我們還可以存取 ESP32 卡的整個記憶體。連接到安裝了 FlashForth 的 ARDUINO 板，然後只需鍵入：

```
hex here 100 dump
```

您應該在終端螢幕上找到它：

```
3FFEE964          DF DF 29 27 6F 59 2B 42 FA CF 9B 84
```

3FFEE970	39 4E 35 F7 78 FB D2 2C A0 AD 5A AF 7C 14 E3 52
3FFEE980	77 0C 67 CE 53 DE E9 9F 9A 49 AB F7 BC 64 AE E6
3FFEE990	3A DF 1C BB FE B7 C2 73 18 A6 A5 3F A4 68 B5 69
3FFEE9A0	F9 54 68 D9 4D 7C 96 4D 66 9A 02 BF 33 46 46 45
3FFEE9B0	45 39 33 33 2F 0D 08 18 BF 95 AF 87 AC D0 C7 5D
3FFEE9C0	F2 99 B6 43 DF 19 C9 74 10 BD 8C AE 5A 7F 13 F1
3FFEE9D0	9E 00 3D 6F 7F 74 2A 2B 52 2D F4 01 2D 7D B5 1C
3FFEE9E0	4A 88 88 B5 2D BE B1 38 57 79 B2 66 11 2D A1 76
3FFEE9F0	F6 68 1F 71 37 9E C1 82 43 A6 A4 9A 57 5D AC 9A
3FFEEA00	4C AD 03 F1 F8 AF 2E 1A B4 67 9C 71 25 98 E1 A0
3FFEEA10	E6 29 EE 2D EF 6F C7 06 10 E0 33 4A E1 57 58 60
3FFEEA20	08 74 C6 70 BD 70 FE 01 5D 9D 00 9E F7 B7 E0 CA
3FFEEA30	72 6E 49 16 0E 7C 3F 23 11 8D 66 55 EC F6 18 01
3FFEEA40	20 E7 48 63 D1 FB 56 77 3E 9A 53 7D B6 A7 A5 AB
3FFEEA50	EA 65 F8 21 3D BA 54 10 06 16 E6 9E 23 CA 87 25
3FFEEA60	E7 D7 C4 45

這對應於閃存的內容。

而 C 語言卻做不到這一點？

是的，但不像 FORTH 語言那樣簡單和互動。

讓我們來看另一個案例，凸顯 FORTH 語言非凡的緊湊性...

但為什麼是堆疊而不是變數呢？

堆疊是幾乎所有微控制器和微處理器上實現的機制。即使 C 語言也利用堆疊，但您無權存取它。

只有 FORTH 語言才能完全存取資料堆疊。例如，要進行加法，我們將兩個值堆疊起來，執行加法，然後顯示結果： **2 5 +** 。顯示 **7** 。

這有點不穩定，但是當您了解資料堆疊的機制時，您會非常欣賞它的強大效率。

資料堆疊允許資料在 FORTH 個字之間傳遞，比透過 C 語言或使用變數的任何其他語言中處理變數要快得多。

你確信嗎？

就我個人而言，我懷疑這一章是否會不可挽回地讓您轉向使用 FORTH 語言進程式設計。當您嘗試掌握 ESP32 卡時，您有兩種選擇：

- 使用 C 語言進程式設計並使用大量可用的程式庫。但您仍將無法使用這些函式庫的功能。將程式碼改編為 C 語言需要真正的 C 語言程式設計知識並掌握 ESP32 卡的架構。開發複雜的程式永遠是個問題。
- 嘗試第四次冒險，探索一個新的、令人興奮的世界。當然，這並不容易。您需要深入了解 ESP32 卡的架構、暫存器、暫存器標誌。作為回報，您將能夠獲得完全適合您的專案的程式設計。

有沒有用 FORTH 寫的专业應用程式？

哦是的！從哈伯太空望遠鏡開始，其某些組件是用 FORTH 語言編寫的。

德國 TGV ICE (Intercity Express) 使用 RTX2000 處理器透過功率半導體控制馬達。 RTX2000 處理器的機器語言是 FORTH 語言。

試圖降落在彗星上的菲萊探測器也使用了相同的 RTX2000 處理器。

如果我們將每個單字視為一個黑盒子，那麼為專業應用程式選擇 FORTH 語言就會變得很有趣。每個單字必須很簡單，因此具有相當短的定義並且依賴很少的參數。



在偵錯階段，測試該字處理的所有可能值變得容易。一旦變得完全可靠，這個詞就變成了一個黑盒子，也就是說，我們對其正常運作擁有無限信心的功能。從字到字，用 FORTH 比用任何其他程式語言更容易使複雜的程式變得可靠。

但如果我們缺乏嚴謹性，如果我們建造天然氣工廠，也很容易得到一個運作不佳的應用程序，甚至完全崩潰！

最後，可以用 FORTH 語言編寫您用任何人類語言定義的單字。然而，可用的字元僅限於 33 到 127 之間的 ASCII 字元集。以下是我們如何象徵性地重寫單字 **high** 和 **low**：

```
\ Active broche de port, ne changez pas les autres.
: __/ ( pinmask portadr -- )
  mset
;
\ Desactivez une broche de port, ne change pas les autres.
: \__ ( pinmask portadr -- )
  mclr
;
```

從此時起，要開啟 LED，您可以輸入：

```
_o_ __/ \ LED on
```

是的！序列 **_o_ __/** 採用 FORTH 語言！

對於 ESP32forth，以下是您可以使用的可以組成 FORTH 單字的所有字元：

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

良好的編程。

在 ESP32Forth 中使用數字

我們毫無問題地啟動了 ESP32Forth。現在我們將深入研究一些數位操作，以了解如何使用 FORTH 語言掌握微控制器。

像許多書籍一樣，我們可以從一個簡單的範例程式開始，例如閃爍 LED。例如這樣：

```
\ 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 ;
```

這段程式碼看似簡單，卻需要一個知識庫，例如記憶體位址、暫存器、二進位遮罩、十六進制數的概念。因此，我們將首先邀請您進行簡單的操作來解決這些基本概念。

FORTH 解譯器的數字

當 ESP32Forth 啟動時，TERA TERM 終端機視窗（或您選擇的任何其他終端程式）應指示 ESP32Forth 可用。按鍵盤上的 **ENTER** 鍵一次或兩次。ESP32Forth 回應並確認成功執行。

我們將測試兩個數字的輸入，這裡是 **25** 和 **33**。鍵入這些數字，然後按鍵盤上的 **ENTER** 鍵。

ESP32Forth 始終響應 **ok**。您剛剛在 ESP32Forth 語言

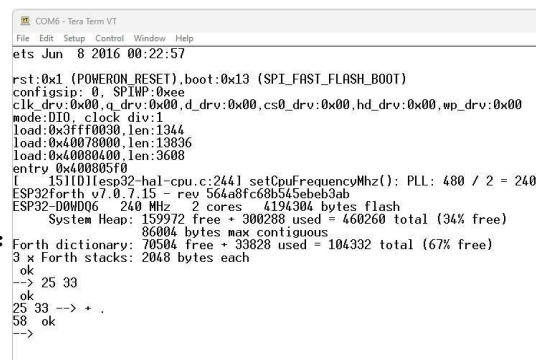
堆疊上堆疊了兩個數字。現在輸入 **+**。然後按 **ENTER** 鍵。

ESP32Forth 顯示結果：

此操作由 FORTH 解譯器處理。

ESP32Forth 與 FORTH 語言的所有版本一樣，有兩種狀態：

- **解釋器**：您剛剛執行兩個數字的簡單求和來測試的狀態；
- **編譯器**：允許定義新詞的狀態。稍後將進一步探討這方面。



The screenshot shows a terminal window titled 'COM6 - Tera Term VT'. The output displays the ESP32Forth boot sequence, including hardware initialization and system status reports. At the bottom, the user has entered the commands '25', '33', and '+', each followed by 'ok' responses from the interpreter.

輸入不同基數的數字

為了完全理解這些解釋，我們邀請您透過 TERA TERM 終端機視窗測試所有範例。

可以自然地輸入數字。在十進制中，它總是一個數字序列，例如：

```
-1234 5678 + .
```

此範例的結果將顯示 **4444**。FORTH 數字和單字必須至少用一個空格字元分隔。如果您每行鍵入一個數字或單字，則該範例將完美運行：

```
-1234
5678
+
.
```

如果要輸入十進制以外的值，可以在數字前面加上前綴：

- **\$** 符號表示該數字是十六進位值；

例：

```
255 .      \ 顯示 255
$ff .      \ 顯示 255
```

這些前綴的目的是避免在相似值的情況下出現任何解釋錯誤：

```
$0305
0305
```

如果未明確定義十六進制數基數，則它們不相等！

改變數值基數

ESP32Forth 有一些單字可讓您更改數字基數：

- **hex** 選擇十六進制數字基數；
- **binary** 進制選擇二進制數基；
- **decimal** 小數選擇十進制數字基數。

以數字基數輸入的任何數字都必須遵循該基數中數字的語法：

```
3E7F
```

如果您使用十進制，則會導致錯誤。

```
hex 3e7f
```

將在十六進制下完美工作。只要未選擇另一個數字基數，新的數字基數就保持有效：

```
hex
$0305
0305
```

是相等的數字！

一旦數字被放入數字基數的資料堆疊中，它的值就不再改變。例如，如果將值 **\$ff** 放入資料堆疊中，則該值（十進制為 **255**，或二進制為 **11111111**）如果我們傳回十進制，則不會改變：

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

冒著堅持的風險，十進制 **255** 與十六進制 **\$ff** 的值相同！

在本章開頭給出的範例中，我們定義了一個十六進位常數：

```
25 constant ledRED
```

如果我們輸入：

```
hex ledRED .
```

這將以十六進位形式顯示該常數的內容。基地的變更不會對 FORTH 程式的最終運作產生任何影響。

二進制和十六進制

現代二進制數系統是二進制代碼的基礎，由戈特弗里德·萊布尼茨 (Gottfried Leibniz) 於 1689 年發明，並出現在他 1703 年的文章《二進制算術解釋》中。

在萊布尼茲的文章中，僅使用字元 **0** 和 **1** 來描述所有數字：

```
: 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
```

有必要了解二進位編碼嗎？我會說是和否。**不**適合日常使用。**是的**，了解微控制器的程式設計和邏輯運算子的掌握。

喬治·布爾正式描述了邏輯。直到第一台電腦出現之前，他的工作被人們遺忘了。克勞德·香農 (Claude Shannon) 意識到該代數可以應用於電路的設計和分析。

布林代數專門處理 **0** 和 **1** 。

我們所有電腦和數位記憶體的基本元件都使用二進位編碼和布林代數。

最小的儲存單位是位元組。它是由 **8** 位元組成的空間。一個位元只能有兩種狀態：**0** 或 **1**。一個位元組可以儲存的最小值是 **00000000**，最大是 **11111111**。如果我們將一個位元組一分為二，我們將得到：

- 低四位，可以取值 **0000** 到 **1111** ；
- 四個最高有效位元可以採用這些相同值之一。

如果我們從 0 開始對 0000 到 1111 之間的所有組合進行編號，我們會得到 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
```

在每行的右側部分，我們顯示與左側部分相同的值，但以十六進位表示：**1101** 和 **D** 是相同的值！

出於實際原因，選擇十六進位表示法來表示計算中的數字。對於 4 位元位元組的最高或最低有效部分，十六進位表示的唯一組合將在 **0** 和 **F** 之間。這裡，字母 **A** 到 **F** 是十六進制數字！

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

00 到 **FF** 表示位元組內容的優點。在十進制中，應該使用 0 到 255。

FORTH 資料堆疊上數字的大小

ESP32forth 使用 32 位元記憶體大小的資料堆疊，即 4 個位元組（8 位元 x 4 = 32 位元）。可堆疊在 FORTH 堆疊上的最小十六進位值將為 **00000000**，最大將為 **FFFFFFFF**。任何嘗試堆疊更大的值都會導致該值被切割：

```
hex
abcdefabcdefabcdef . \ display: EFABCDEF
```

讓我們以 32 位元（4 位元組）十六進位格式堆疊最大可能值：

```
decimal
$ffffffff . \ display: -1
```

我看你很驚訝，但這個結果**很正常**！單字。以有符號形式顯示資料堆疊頂部的值。要顯示相同的無符號值，必須使用單字 **u.**：

```
$ffffffff u.    \ display:    4294967295
```

這是因為 **FORTH** 使用 **32** 位元來表示整數，最高有效位元用作符號：

- 若最高有效位為 **0**，則該數為正；
- 如果最高有效位為 **1**，則該數字為負數。

因此，如果您遵循正確的話，我們的十進制值 **1** 和 **-1** 將在堆疊上以二進位格式表示，形式如下：

```
binary
000000000000000000000000000000001  \ push 1 on stack
111111111111111111111111111111111  \ push -1 on stack
```

在這裡，我們將請我們的數學家萊布尼茨先生將這兩個數字以二進位形式相加。如果我們像在學校一樣，從右邊開始，你只需要遵守這個規則：二元的 **1 + 1 = 10**。我們將結果放在第三行：

```
000000000000000000000000000000001
111111111111111111111111111111111
10
```

下一步：

```
000000000000000000000000000000001
111111111111111111111111111111111
10
100
```

最後，我們將得到結果：

```
000000000000000000000000000000001
111111111111111111111111111111111
100000000000000000000000000000000
```

但由於該結果的第 **33** 個最高有效位為 **1**，因此知道整數格式嚴格限制為 **32** 位，因此最終結果為 **0**。令人驚訝嗎？但這就是每個數字時鐘的作用。隱藏營業時間。當達到 **59** 時，加 **1**，時鐘將顯示 **0**。

十進制算術規則，即 **-1 + 1 = 0**，在二進制邏輯中得到了完美的遵守！

記憶體存取和邏輯運算

資料堆疊絕不是資料儲存空間。它的尺寸也非常有限。並且棧是由很多單字共享的。參數的順序是基本的。錯誤可能會導致故障。讓我們以顯示記憶體空間內容的單字 **dump** 為例：

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

以粗體和紅色顯示，我們發現保留用於在分數變數中儲存值的四個位元組。讓我們在 **Score** 中儲存任何值：

```
decimal
```



```
1900 score !  
hex  
score 10 dump \ display:  
3FFEE90C                                6C 07 00 00  
3FFEE910                37 33 36 37 30 33 34 34 79 64 31 30
```

我們找到包含十進制值 **1900**、十六進位值 **0000076C** 的四個位元組。還感到驚訝嗎？因此，二進位編碼的效果及其微妙之處就是原因。在記憶體中，位元組從最低有效位元組開始儲存。恢復後，轉換機制是透明的：

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

讓我們回到讓 **LED** 閃爍的程式碼。提煉：

```
1 mLED RED      GPIO_ENABLE_REG regSet
```

此程式碼啟動與 LED 關聯的 GPIO 輸出。GPIO_ENABLE_REG 是一個常數，其內容是指向該 LED 的遮罩。我們不妨這樣寫：

```
1 25 lshift GPIO_ENABLE_REG !
```

這裡，字 **lshift** 執行向左邏輯移位 25 位元：

```
\ before shift: %000000000000000000000000000000001  
\ after shift: %0000001000000000000000000000000000000
```

提醒一下，GPIO 的²編號為 0 到 31。要啟動另一個 GPIO，例如 GPIO17，我們可以執行以下指令：

```
1 17 lshift GPIO_ENABLE_REG !
```

假設我們想要透過一條指令啟動 **GPIO 17** 和 **25**，我們將執行以下指令：

```
1 25 lshift
1 17 lshift or GPIO ENABLE REG !
```

我們做了什麼？以下是操作的詳細資訊：

```
\ 1 25 lshift \ %00000001000000000000000000000000
\ 1 17 lshift \ %00000001000000000100000000000000
\ or          \ %00000001000000000100000000000000
```

字 **or** 執行了將兩個偏移量組合成單一二進位遮罩的操作。

讓我們回到**分數變數**。我們想要隔離最低有效位元組。我們可以使用多種解決方案。一種解決方案使用帶有邏輯運算子 **and** 的**二進位遮罩**：

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

要隔離右側的第二個位元組：

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

2 通用輸入/輸出 = 一般輸入-輸出

在這裡，我們對變數的內容感到很有趣。要掌握像安裝在 **ESP32** 卡上的微控制器，其機制幾乎沒有什麼不同。最困難的部分是找到正確的暫存器。這將是另一章的主題。

總而言之，關於二進制邏輯和不同可能的數字編碼，還有很多東西需要學習。如果您測試過這裡給出的幾個範例，您肯定會明白 **FORTH** 是一種有趣的語言：

- 由於其解釋器，它允許互動式地進行大量測試，而無需透過上傳程式碼重新編譯；
- 一本字典，其中大部分單字都可以透過口譯員找到；
- 動態新增單字，然後立即測試它們。

最後，這不會破壞任何東西，**FORTH** 程式碼一旦編譯，肯定與 **C** 語言中的等效程式碼一樣有效率。

使用 ESP32Forth 的真正 32 位元 FORTH

ESP32Forth 是真正的 32 位元 FORTH。這是什麼意思？

FORTH 語言支援整數值的運算。這些值可以是文字值、記憶體位址、暫存器內容等。

資料堆疊上的值

當 ESP32Forth 啟動時，FORTH 解釋器可用。如果輸入任何數字，它將作為 32 位元整數放入堆疊中：

```
35
```

如果我們堆疊另一個值，它也會被堆疊。前一個值將被下推一位：

```
45
```

為了添加這兩個值，我們使用一個詞，這裡 **+**：

```
+
```

我們將兩個 32 位元整數值相加，並將結果放入堆疊中。為了顯示這個結果，我們將使用單字 **. **：

```
. \ 顯示 80
```

在 FORTH 語言中，我們可以將所有這些操作集中在一行中：

```
35 45 + . \ 顯示 80
```

與 C 語言不同，我們沒有定義 **int8** 或 **int16** 或 **int32** 型別。

對於 ESP32Forth，ASCII 字元將由 32 位元整數指定，但其值將受到 [32..256]。例：

```
67 emit \顯示 C
```

記憶體中的值

ESP32Forth 允許您定義常數和變數。它們的內容將始終採用 32 位元格式。但有些情況下這不一定適合我們。讓我們舉一個簡單的例子，定義摩斯電碼字母表。我們只需要幾個位元組：

- 1 定義莫爾斯電碼符號的數量
- 摩斯電碼的每個字母一個或多個位元組

```
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,
```

這裡我們只定義了 3 個字，**mA**、**mB** 和 **mC**。每個字中儲存幾個位元組。問題是：我們如何檢索這些單字中的信息？

這些字之一的執行會存入一個 32 位元值，對應到我們儲存摩斯電碼資訊的記憶體位址。我們將使用單字 **c@** 從每個字母中提取莫爾斯電碼：

```
mA c@ . \ 顯示 2
mB c@ . \ 顯示 4
```

像這樣提取的第一個位元組將用於管理循環以顯示字母的莫爾斯電碼：

```
: .morse ( addr -- )
  dup 1+ swap c@ 0 do
    dup i + c@ emit
  loop
  drop
;
mA .morse \ 顯示 .-
mB .morse \ 顯示 -...
mC .morse \ 顯示-.-。
```

當然還有很多更優雅的例子。在這裡，它展示了一種操作 8 位元值（我們的位元組）的方法，同時我們在 32 位元堆疊上使用這些位元組。

根據資料大小或類型進行文字處理

在所有其他語言中，我們都有一個通用詞，例如 **echo**（在 **PHP** 中），它可以顯示任何類型的資料。無論是整數、實數、字串，我們總是使用同一個字。**PHP** 語言範例：

```
$bread = "backed bread";
$price = 2.30;
echo $bread . " : " . $price;
// display   backed bread: 2.30
```

對於所有程式設計師來說，這種做事方式就是標準！那麼 **FORTH** 將如何在 **PHP** 中完成這個範例呢？

```
: bread s" backed bread" ;
: price s" 2.30" ;
bread type   s" : " type   price type
\ display   backed bread: 2.30
```

這裡，單字 **type** 告訴我們我們剛剛處理了一個字串。

PHP（或任何其他語言）具有通用函數和解析器，而 **FORTH** 使用單一資料類型進行補償，但採用了適應的處理方法，這些方法告訴我們所處理資料的性質。

這是 **FORTH** 的一個絕對簡單的例子，以 **HH:MM:SS** 格式顯示秒數：

```

: :##
  # 6 base !
  # decimal
  [char] : hold
;
: .hms ( n -- )
  <# :## :## # # #> type
;
4225 .hms \ 顯示: 01:10:25

```

我喜歡這個例子，因為到目前為止，**沒有其他程式語言**能夠如此優雅而簡潔地執行這種 HH:MM:SS 轉換。

您已經明白，FORTH 的秘密就在於它的詞彙。

結論

FORTH 沒有資料型別。所有資料都透過資料堆疊。堆疊中的每個位置始終是 32 位元整數！

這就是所有需要知道的。

超結構化和冗長語言（例如 C 或 Java）的純粹主義者肯定會大喊異端。在這裡，我將允許自己回答這些問題：為什麼需要輸入資料？

因為 FORTH 的強大之處就在於這種簡單性：具有非類型化格式和非常簡單的操作的單一資料堆疊。

我將向您展示許多其他程式語言無法做到的事情，定義新的定義詞：

```

: 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   .- -... -.-.

```

在這裡，莫爾斯電碼這個詞：已經成為一個定義詞，就像**常數**或**變數**一樣...

因為 FORTH 不只是一種程式語言。它是一種元語言，也就是說一種建構你自己的程式語言的語言...

評論和澄清

沒有 IDE³可以以結構化方式管理和呈現用 FORTH 語言編寫的程式碼。最糟的情況是你使用 ASCII 文字編輯器，最好的情況是使用真正的 IDE 和文字檔案：

- Windows 上的編輯或寫字板
- Linux 下編輯
- windows 下的 PsPad
- Netbeans ...

這是初學者可以寫的程式碼片段：

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

這段程式碼將被 ESP32forth 完美編譯。但是，如果將來需要在另一個應用程式中修改或重複使用它，它仍然可以理解嗎？

編寫可讀的 FORTH 程式碼

讓我們從要定義的單字的名稱開始，這裡是 **cycle.stop**。ESP32forth 允許您編寫很長的單字名稱。定義的字的大小對最終應用程式的效能沒有影響。因此，我們有一定的自由來寫下這些話：

- 就像 JavaScript 中的物件程式設計：**cycle.stop**
- Camel 方式編碼週期停止
- 的程式設計師來說
- **cs1** 程式碼的程式設計師

沒有規則。最主要的是您可以輕鬆地重新閱讀 FORTH 程式碼。然而，使用 FORTH 語言的電腦程式設計師有一定的習慣：

- 大寫字元常數 **MAX_LIGHT_TIME_NORMAL_CYCLE**
- 定義其他單字 **defPin:** 的單字，即單字後面跟著冒號；
- 位址轉換 word **>date**，這裡位址參數增加一定的值以指向對應的資料；
- 記憶體存儲字 **date@**或 **date!**
- 數據顯示字 **.date**

那麼用英語以外的語言命名第四個單字又如何呢？再次強調，只有一條規則：**完全自由**！但請注意，ESP32forth 不接受以拉丁字母以外的字母書寫的名稱。但是，您可以使用這些字母進行註釋：

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

3 整合開發環境=整合開發環境

```
...code... ;
```

或者

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

原始碼縮排

無論程式碼是兩行、十行或更多，一旦編譯，對程式碼的效能沒有影響。因此，您不妨以結構化的方式縮排程式碼：

- 控制結構的每個字一行 **if else then , begin while Repeat...**對於 **if** 這個詞，我們可以在它前面加上它將要處理的邏輯測試；
- 執行預定義字的一行，如有必要，前面可以是該字的參數。

例：

```
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
;

```

如果控制結構中處理的程式碼是稀疏的，則可以壓縮 **FORTH** 程式碼：

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

```

endof endcase 結構常出現這種情況：

```
: socketError ( -- )
  errno dup
  case
    2 of      ." No such file "      endof
    5 of      ." I/O error "         endof
    9 of      ." Bad file number "   endof
    22 of     ." Invalid argument "  endof
  endcase
  . quit
;

```

評論

與任何程式語言一樣，FORTH 語言允許在原始程式碼中添加註釋。新增註解對編譯原始碼後的應用程式的效能沒有影響。

在 FORTH 語言中，我們有兩個字來分隔註釋：

- 單字（後面必須至少跟一個空格字元。此註記由 字元完成）；
- \ 一詞後面必須至少跟一個空格字元。該單字後面是該單字與行尾之間任意大小的註解。

（一詞廣泛用於堆疊註解。範例：

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

堆疊評論

正如我們剛剛看到的，它們由（和）標記。它們的內容在編譯或執行期間對 FORTH 程式碼沒有影響。所以我們可以在（和）之間放置任何內容。至於堆疊註釋，我們將保持非常簡潔。--符號代表 FORTH 單字的動作。before 的指示-對應於執行該字之前放置在資料堆疊上的資料。--之後的指示對應於執行該字後留在資料堆疊上的資料。例子：

- words (--) 表示該字不處理資料棧上的任何資料；
- emit (c --) 表示該字將資料作為輸入處理，並且不會在資料堆疊上留下任何內容；
- bl (--32) 表示該字不處理任何輸入數據，將十進位值 32 留在數據堆疊上；

在字執行之前或之後處理的資料量沒有限制。提醒一下，（和）之間的指示僅供參考。

註解中堆疊參數的意義

首先，有必要進行一個小但非常重要的澄清。這是堆疊上資料的大小。對於 ESP32Forth，堆疊資料佔用 4 個位元組。所以這些是 32 位元格式的整數。然而，有些字以 8 位元格式處理資料。那我們在資料棧上放什麼呢？使用 ESP32Forth，它將始終是 32 位元資料！一個有 c 字的例子！：

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

這裡，參數 c 表示我們以 32 位元格式堆疊一個整數值，但其值將始終包含在區間[0..255]中。

標準參數始終為 n。如果有多個整數，我們將對它們進行編號：n1 n2 n3 等。

因此，我們可以像這樣寫前面的範例：

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

但它比以前的版本要明確得多。以下是您將在原始程式碼中看到的一些符號：

- **addr** 表示字面記憶體位址或由變數傳遞；
- **c** 表示區間[0..255]內的 8 位值
- **d** 表示雙精度值。
不與已經是 32 位元格式的 ESP32Forth 一起使用；
- **fl** 表示布林值，0 或非零；
- **n** 表示整數。ESP32Forth 的 32 位元有符號整數；
- **str** 表示字串。相當於 **addr len --**
- **u** 表示無符號整數

沒有什麼可以阻止我們更明確一點：

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

單字定義 單字註釋

定義詞使用 **create** 和 **does>**。對於這些單詞，建議像這樣編寫堆疊註解：

```
\ 定義 SSD1306 的指令或資料流
: 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
    sendDataToSSD1306
;
```

| 分為兩部分。：

- 左邊是執行定義詞時的動作部分，前綴為 **comp:**
- 右側是要定義的單字的操作部分，以 **exec:** 為前綴：

冒著堅持的風險，這不是一個標準。這些只是建議。

文字評論

它們由單字\表示，後面跟著至少一個空格字元和說明文字：

```
\ store at <WORD> addr length of datas compiled beetween
\ <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!  
;
```

這些註釋可以用原始碼編輯器支援的任何字母書寫：

```
\ 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!  
;
```

註釋在原始碼開頭

透過大量的程式設計實踐，您很快就會發現自己擁有數百甚至數千個原始檔案。為了避免文件選擇錯誤，強烈建議使用註釋標記每個來源文件的開頭：

```
\ *****  
\ 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 PETREMANN  
\   Author:        Marc PETREMANN  
\   GNU General Public License  
\ *****
```

所有這些資訊均由您自行決定。當您幾個月或幾年後再次查看文件的內容時，它們會變得非常有用。

最後，請毫不猶豫地用 FORTH 語言對原始檔進行註解和縮排。

診斷和調整工具

第一個工具涉及編譯或解釋警報：

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

這裡，**DDUP** 這個字不存在。此錯誤之後的任何編譯都將失敗。

反編譯器

在傳統的編譯器中，原始碼被轉換為包含編譯器所配備的函式庫的參考位址的可執行程式碼。要取得可執行程式碼，必須連結目標程式碼。程式設計師在任何時候都無法僅使用編譯器的資源來存取其庫中包含的可執行程式碼。

使用 ESP32Forth，開發人員可以反編譯他們的定義。要反編譯一個單詞，只要輸入 **see**，然後輸入要反編譯的單字：

```

: C>F ( 0C --- 0F) \ Conversion Celsius in Fahrenheit
  9 5 */ 32 +
;
see c>f
\ display:
: C>F
  9 5 */ 32 +
;

```

ESP32Forth 的 FORTH 字典中的許多單字都可以反編譯。

反編譯您的單字可以讓您偵測可能的編譯錯誤。

記憶體轉儲

有時希望能夠看到記憶體中的值。字元轉儲接受兩個參數：記憶體中的起始位址和要顯示的位元組數：

```

create myDATAS 01 c, 02 c, 03 c, 04 c,
hex
myDATAS 4 dump      \ displays:
3FFEE4EC                                     01 02 03 04

```

電池監視器

.s 隨時顯示資料堆疊的內容。以下是利用 .s 的 .DEBUG 一詞的定義：

```

variable debugStack

: debugOn ( -- )
  -1 debugStack !
;

: debugOff ( -- )
  0 debugStack !
;

: .DEBUG
  debugStack @
  if
    cr ." STACK: " .s
    key drop
  then
;

```

要使用 .DEBUG ，只需將其插入到要調試的單字中的重要位置即可：

```

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

```

do 循環中執行單字 **i** 後顯示資料堆疊的內容。我們啟動對焦並運行 **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
```

debugOn 啟用偵錯時，每次顯示資料堆疊內容都會暫停我們的 **do** 循環。執行 **debugOff** 以便 **myTEST** 字正常執行。

字典/堆疊/變數/常數

展開字典

Forth 屬於編織解釋語言類別。這意味著它可以解釋在控制台上鍵入的命令，以及編譯新的子程式和程式。

Forth 編譯器是語言的一部分，特殊單字用於建立新的字典條目（即單字）。最重要的是：**:**（開始一個新定義）和**;**（完成定義）。讓我們輸入以下命令來嘗試：

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

發生了什麼事？的動作是建立一個名為***+**的**新字典條目**，並從解釋模式切換到編譯模式。在編譯模式下，解釋器會搜尋單字，而不是執行它們，而是安裝指向其程式碼的指標。如果文字是數字，則 ESP32forth 不會將其壓入堆疊，而是在為新單字分配的字典空間中建構數字，並遵循每次執行該單字時將儲存的數字放入堆疊的特殊程式碼。因此，***+** 的執行動作是順序執行先前定義的字*****和**+**。

這個詞 **:** 很特別。它是一個立即字，即使系統處於編譯模式，它也總是被執行。什麼是**;**是雙重的。首先，它安裝將控制項返回到解釋器的下一個外部層級的程式碼，其次，它從編譯模式返回到解釋模式。現在試試你的新字：

```
decimal 5 6 7 *+ . \ 顯示 47
```

此範例說明了 Forth 中的兩個主要工作活動：將新單字新增到字典中，並在定義後立即嘗試它。

字典管理

單字 **forget** 後面跟著「要刪除」的單字將刪除自該單字以來您所做的所有字典條目：

```
: test1 ;  
: test2 ;  
: test3 ;  
忘記 test2 \ 從字典中刪除 test2 和 test3
```

堆疊和逆波蘭表示法

Forth 有一個明確可見的堆疊，用於在單字（命令）之間傳遞數字。使用 Forth 可以有效地迫使您從堆疊的角度進行思考。一開始這可能會很困難，但就像任何事情一樣，透過練習它會變得容易得多。

在 FORTH 中，這堆類似於一堆寫有數字的卡片。數字總是添加到堆疊頂部並從堆疊頂部刪除。

ESP32forth 整合了兩個堆疊：參數堆疊和回饋堆疊，每個堆疊由多個可容納 16 位元數字的單元組成。

第四條輸入線：

```
decimal 2 5 73 -16
```

保留參數堆疊不變

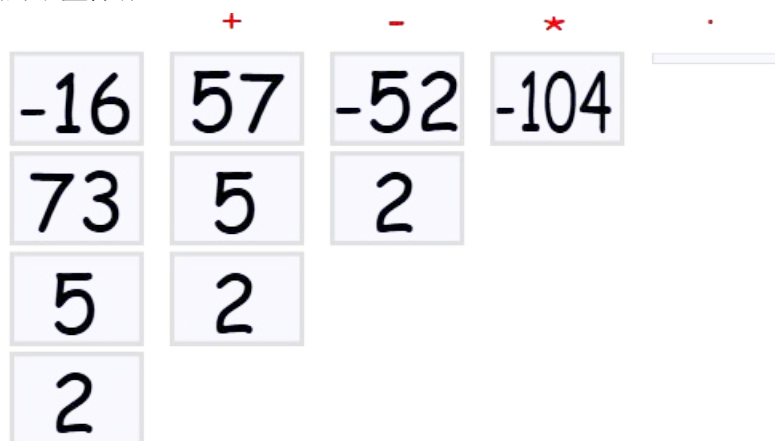
	細胞	內容	評論
0		-16	(TOS) 右上角
1		73	(NOS) 下一個
2		5	
3		2	

我們通常會在 **Forth** 資料結構（例如堆疊、陣列和表格）中使用從零開始的相對編號。請注意，當像這樣輸入數字序列時，最右邊的數字將成為 *TOS*，最左邊的數字位於堆疊的底部。

假設我們使用以下行跟隨原始輸入行

```
+ - * .
```

這些操作將產生連續的堆疊操作：



兩行之後，控制台顯示：

```
decimal 2 5 73 -16 \ 顯示: 2 5 73 -16 ok
+ - * .           \ 顯示: -104 ok
```

請注意，**ESP32forth** 在解釋每一行時會方便地顯示堆疊元素，並且 **-16** 的值顯示為 **32** 位元無符號整數。此外，這個字。消耗資料值**-104**，使堆疊為空。如果我們執行。在現在為空的堆疊上，外部解釋器將中止並出現堆疊指標錯誤 **STACK UNDERFLOW ERROR**。

先出現運算元，後跟運算子的程式表示法稱為逆波蘭表示法 (**RPN**)。

處理參數堆疊

作為一個基於堆疊的系統，**ESP32forth** 必須提供將數字放入堆疊、刪除它們以及重新排列它們的順序的方法。我們已經看到，只需鍵入數字即可將數字放入堆疊中。我們也可以將數字整合到 **FORTH** 單字的定義中。

drop 這個字從堆疊頂部刪除一個數字，從而將下一個數字放在頂部。單字**交換**交換前 **2** 個數字。 **dup** 複製頂部的數字，將所有其他數字向下推。 **rot** 旋轉前 **3** 個數字。這些行動如下所示。



返回堆疊及其用途

編譯新單字時，ESP32forth 會在調用單字和之前定義的單字之間建立鏈接，這些單字將由新單字的執行呼叫。這種連結機制在運行時使用 **rstack**。要呼叫的下一個字的位址被放置在返回堆疊中，以便噹噹前字執行完畢時，系統知道從哪裡移動到下一個字。由於單字可以嵌套，因此必須有一個這些返回位址的堆疊。

除了充當傳回位址的儲存庫之外，使用者還可以儲存和檢索返回堆疊，但必須小心操作，因為返回堆疊對於程式執行至關重要。如果您使用回電電池進行臨時存儲，則必須將其恢復到原始狀態，否則您可能會導致 ESP32forth 系統崩潰。儘管存在危險，但有時使用 **backstack** 作為臨時儲存可以降低程式碼的複雜性。

要儲存在堆疊上，請使用 **>r** 將參數堆疊的頂部移動到返回堆疊的頂部。要檢索值，**r>** 將堆疊頂部的值移回參數堆疊的頂部。為了簡單地從堆疊頂部刪除一個值，可以使用 **rdrop** 一詞。單字 **r@** 將堆疊頂部複製回參數堆疊。

記憶體使用情況

@ (fetch) 從記憶體中取出到堆疊，並透過字 **!** 從頂部儲存到記憶體。（瞎的）。**@** 期望堆疊上的地址並用其內容替換該地址。**!** 需要一個數字和一個地址來儲存它。它將數字放置在位址引用的記憶體位置中，同時消耗該過程中的兩個參數。

表示 8 位元（位元組）值的無符號數字可以放置在字元大小的字元中。使用 **c@** 和 **c!** 的儲存單元。

```
create testVar
  cell allot
  $f7 testVar c!
testVar c@ . \ 顯示 247
```

變數

變數是記憶體中的命名位置，可以將數字（例如計算的中間結果）儲存在堆疊之外。例如：

```
variable x
```

建立一個名為 **x** 的儲存位置，執行時將其儲存位置的位址保留在堆疊頂部：

```
x . \ 顯示位址
```

然後我們可以在這個地址收集或儲存：

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

常數

常數是程式運行時您不希望更改的數字。執行與常數關聯的字的結果是保留在堆疊上的資料的值。

```
\ 定義 VSPI 腳
19 constant VSPI_MISO
```



```

23 constant VSPI_MOSI
18 constant VSPI_SCLK
05 constant VSPI_CS

\ 設定 SPI 連接埠頻率
4000000 constant SPI_FREQ

\ 選擇 SPI 詞彙
only FORTH SPI also

\ 初始化 SPI 端口
: init.VSPI ( -- )
    VSPI_CS OUTPUT pinMode
    VSPI_SCLK VSPI_MISO VSPI_MOSI VSPI_CS SPI.begin
    SPI_FREQ SPI.setFrequency
;

```

偽常數值

用 **value** 定義的值是變數和常數的混合型別。我們設定並初始化一個值，然後像呼叫常數一樣呼叫它。我們也可以像更改變數一樣更改值。

```

decimal
13 value thirteen
thirteen .      \顯示: 13
47 to thirteen
thirteen .      \顯示: 47

```

單字 **to** 也適用於單字定義，將其後面的值替換為當前堆疊頂部的值。您需要注意的是，**to** 後面是由 **value** 定義的值，而不是其他內容。

記憶體分配的基本工具

創建和**分配**這兩個詞是保留記憶體空間並為其附加標籤的基本工具。例如，以下轉錄顯示了一個新的**圖形數組**字典條目：

```

create graphic-array ( --- addr )
    %00000000 c,
    %00000010 c,
    %00000100 c,
    %00001000 c,
    %00010000 c,
    %00100000 c,
    %01000000 c,
    %10000000 c,

```

執行時，**圖形數組**字將推送第一個條目的位址。

使用前面解釋的獲取和儲存字來存取分配給**圖形陣列**的記憶體。要計算分配給**圖形數組**的第三個位元組的位址，我們可以編寫**圖形數組** **graphic-array 2 +**，記住索引從 **0** 開始。

```

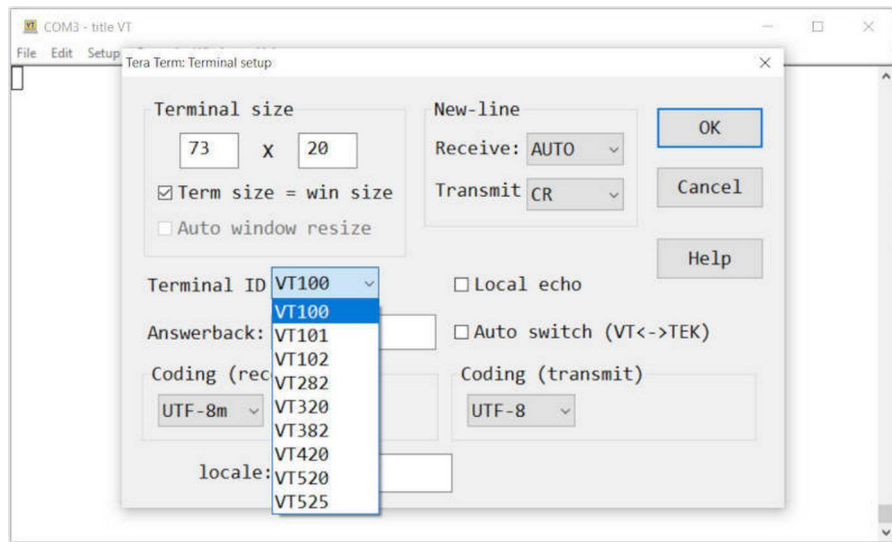
30 graphic-array 2 + c!
graphic-array 2 + c@ .      \ 顯示 30

```

終端機上的文字顏色和顯示位置

端子的 ANSI 編碼

如果您使用終端軟體與 ESP32forth 進行通信，則該終端很可能模擬 VT 類型終端或同等終端。此處，TeraTerm 配置為模擬 VT100 終端：



這些終端有兩個有趣的功能：

- 為頁面背景和要顯示的文字著色
- 定位顯示遊標

這兩個功能均由 ESC（轉義）序列控制。這就是 ESP32forth 中 **bg** 和 **fg** 的定義方式：

```
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" ;
```

單字 **normal**「正常」會覆蓋 **bg** 和 **fg** 定義的著色序列。

單字 **page** 頁面清除終端螢幕並將遊標定位在螢幕的左上角。

文字著色

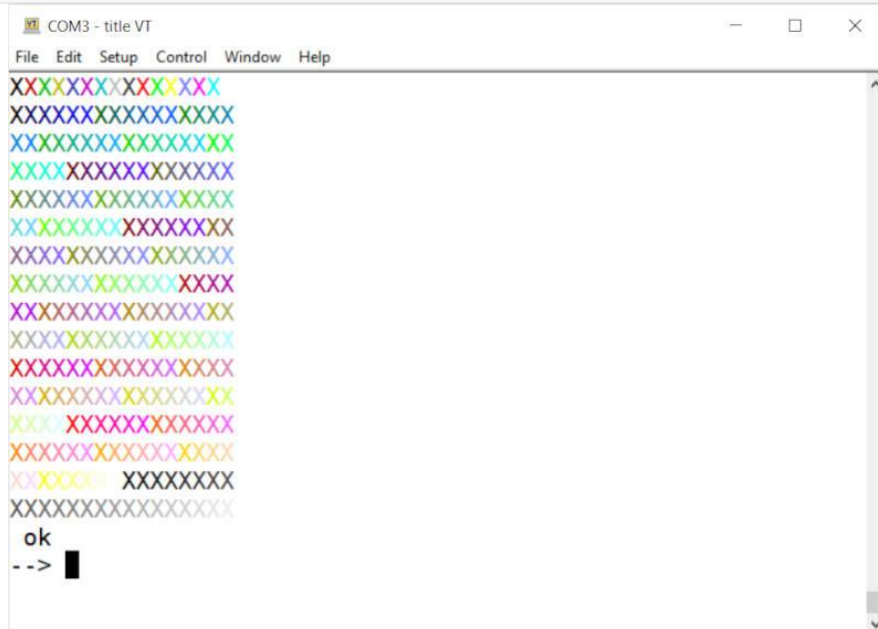
讓我們先看看如何為文字著色：

```
: testFG ( -- )
  page
```

```

16 0 do
  16 0 do
    j 16 * i + fg
    ." X"
  loop
cr
loop
normal
;

```



運行 **testFG** 會顯示以下內容：

要測試背景顏色，我們將按以下步驟進行：

```

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

```

執行 **testBG** 會顯示以下內容：



顯示位置

該終端是與 ESP32forth 通訊的最簡單的解決方案。使用 ANSI 轉義序列可以輕鬆改進資料的表示。

```
09 constant red
11 constant yellow
14 constant cyan
15 constant whyte
: box { x0 y0 xn yn color -- }
  color bg
  yn y0 - 1+ \ 決定高度
  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
;
```

運行 **3boxes** 顯示如下：



現在您可以建立簡單有效的介面，允許與 **ESP32forth** 編譯的 **FORTH** 定義進行互動。

ESP32Forth 的局部變數

介紹

FORTH 語言主要透過資料棧來處理資料。這種非常簡單的機制提供了無與倫比的性能。相反，追蹤資料流很快就會變得複雜。局部變數提供了一個有趣的選擇。

假堆疊註釋

如果您遵循第四個不同的範例，您將注意到由 (和) 構成的堆疊註解。例子：

```
\ 將兩個無符號值相加，將和和進位留在堆疊上
: um+ ( u1 u2 -- sum carry )
  \ here the definition
;
```

在這裡，註釋 (**u1 u2 -- sum carry**) 對 FORTH 程式碼的其餘部分絕對沒有任何作用。這是純粹的評論。

當準備複雜的定義時，解決方案是使用由 { 和 } 構成的局部變數。例：

```
: 2OVER { a b c d }
  a b c d a b
;
```

我們定義了四個局部變數 **a b c** 和 **d**。

單字 { 和 } 與單字 (和) 類似，但效果完全不同。位於 { 和 } 之間的程式碼是局部變數。唯一的限制：不要使用可能是 FORTH 字典中的 FORTH 單字的變數名稱。我們不妨這樣寫我們的例子：

```
: 2OVER { varA varB varC varD }
  varA varB varC varD varA varB
;
```

每個變數都會按照其在資料堆疊上的存放順序取得資料堆疊的值。這裡，**1** 進入 **varA**，**2** 進入 **varB**，以此類推：

```
--> 1 2 3 4
ok
1 2 3 4 --> 2over
ok
1 2 3 4 1 2 -->
```

我們的假堆疊註解可以這樣完成：

```
: 2OVER { varA varB varC varD -- varA varB varC varD varA varB }
```

-- 以下字元無效。唯一的一點是讓我們的假評論看起來像真正的堆疊評論。

對局部變數的操作

value 定義的偽變數完全相同。例：

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

的效果：

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

在此範例中， **var** 由 **value** 明確定義。

to 或 **+to** 為局部變數賦值，以增加局部變數的內容。在此範例中，我們在單字程式碼中加入一個初始化為零的局部變數 **result**：

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

不是比這個更具可讀性嗎？

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

um+ 的定義，它將兩個無符號整數相加，並將總和以及該總和的溢出值保留在資料堆疊上：

```
\ 將兩個無符號整數相加，將和和進位留在堆疊上
: um+ { u1 u2 -- 進位與 }
: 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
  ;
```

以下是一個更複雜的例子，使用局部變數重寫 **DUMP**：


```

\ 轉儲中的局部變數:
\ START_ADDR    \ 轉儲的首位址
\ END_ADDR      \ 轉儲的最後位址
\ OSTART_ADDR   \ 轉儲中循環的第一個位址
\ LINES         \ 轉儲循環的行數
\ myBASE        \ 目前數值基數
: 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

```

局部變數的使用極大地簡化了堆疊上的資料操作。程式碼更具可讀性。請注意，不需要預先聲明這些局部變量，在使用它們時指定它們就足夠了，例如：**base @ { myBASE }**。

>r 和 **r>** 一詞，否則您將面臨破壞局部變數管理的風險。只要看一下這個版本的 **DUMP** 的反編譯就可以明白這個警告的原因：

```

: 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 ;

```

ESP32forth 的資料結構

前言

ESP32forth 是 FORTH 語言的 32 位元版本。那些從 FORTH 一開始就練習的人都使用 16 位元版本進行程式設計。此資料大小由儲存在資料堆疊上的元素的大小決定。要找出元素的大小（以位元組為單位），您必須執行位元組。為 ESP32forth 運行這個字：

```
cell . \ affiche 4
```

值 4 表示放置在資料堆疊上的元素的大小為 4 個位元組，即 $4 \times 8 \text{ 位元} = 32 \text{ 位元}$ 。

對於 16 位元 FORTH 版本，cell 將堆疊值 2。同樣，如果使用 64 位元版本，cell 將堆疊值 8。

FORTH 中的表格

讓我們從相當簡單的結構開始：表。我們將只討論一維或二維數組。

一維 32 位元資料數組

這是最簡單的表格類型。要建立這種類型的表，我們使用單字 **create** 後跟要建立的表的名稱：

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

在此表中，我們儲存 6 個值：34、37....12。要檢索值，只需使用單字 **@**，透過將溫度堆疊的位址增加到所需的偏移量即可：

```
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
```

我們可以透過定義一個將計算該位址的字來將存取程式碼分解為所需的值：

```
: temp@ ( index -- value )
    cell * temperatures + @
;
```

```
0 temp@ . \ affiche 34
2 temp@ . \ affiche 42
```

你會注意到，對於這個表中儲存的 n 個值，這裡是 6 個值，存取索引必須始終在區間 $[0..n-1]$ 內。

表格定義字

以下是建立一維整數數組的字定義的方法：

```
: 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
```

在我們的範例中，我們儲存 0 到 255 之間的 6 個值。很容易建立陣列的變體來以更緊湊的方式管理我們的資料：

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

使用此變體，相同的值儲存在四倍的記憶體空間中。

在表中讀取和寫入

完全可以建立一個包含 n 個元素的空數組，並在該數組中寫入和讀取值：

```
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
```

在我們的範例中，陣列包含 6 個元素。使用 ESP32forth，有足夠的記憶體空間來處理更大的數組，例如 1,000 或 10,000 個元素。建立多維表很容易。二維數組的範例：

```
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'
```

mySCREEN 的二維表，它將是一個 16 行 63 列的虛擬螢幕。

只需保留一個記憶體空間即可使用，該記憶體空間是要使用的表尺寸 **X** 和 **Y** 的乘積。現在讓我們來看看如何管理這個二維數組：

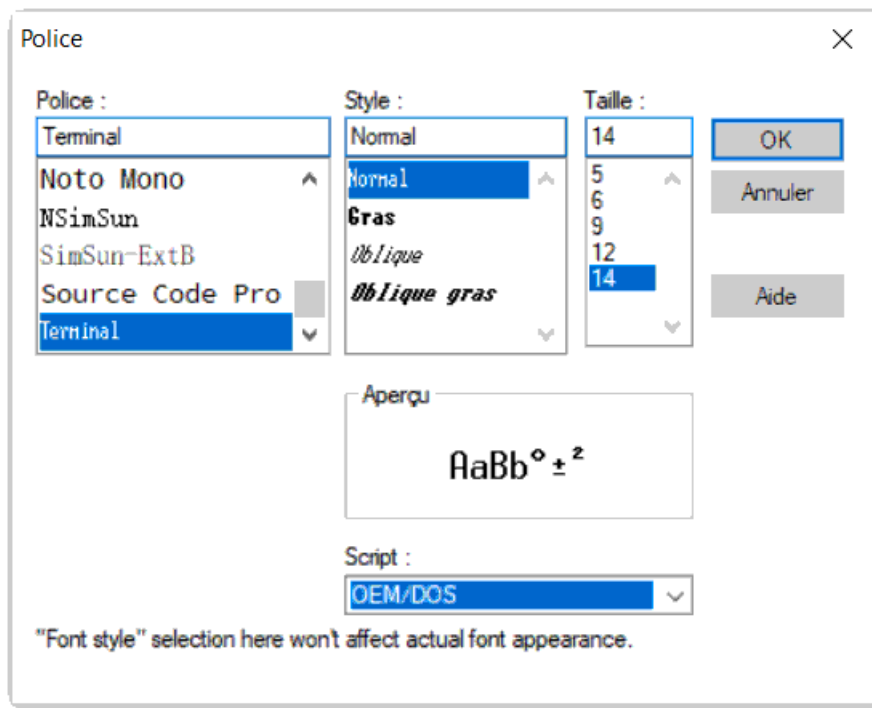
```
: xySCRaddr { x y -- addr }
  SCR_WIDTH y *
  x + mySCREEN +
;
: SCR@ ( x y -- c )
  xySCRaddr c@
;
: SCR! ( c x y -- )
  xySCRaddr c!
;
char X 15 5 SCR!    \ stocke char X à col 15 ligne 5
15 5 SCR@ emit      \ affiche X
```

管理虛擬螢幕的實際範例

在進一步討論管理虛擬螢幕的範例之前，讓我們看看如何修改 **TERA TERM** 終端機的字元集並顯示它。

啟動 **TERA** 術語：

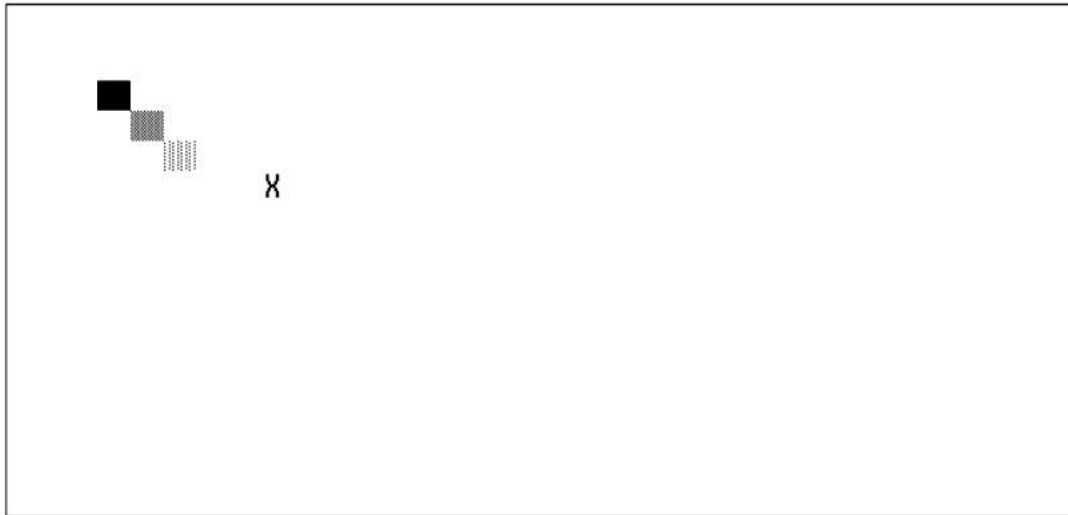
- 在功能表列中，按一下“設定”
- 選擇字體和字體...
- 配置以下字體：



顯示可用字元表的方法如下：

```
: 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
```

這是運行 **tableChars** 的結果：



在我們的虛擬螢幕範例中，我們展示了管理二維數組的具體應用。我們的虛擬螢幕可供書寫和閱讀。這裡我們在終端機視窗中顯示虛擬螢幕。這種顯示方式遠非有效率。但在真正的 **OLED** 螢幕上它可以更快。

複雜結構的管理

ESP32forth 具有結構詞彙表。該詞彙表的內容使得定義複雜的資料結構成為可能。

這是一個簡單的範例結構：

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

在這裡，我們定義了 **YMDHMS** 結構。此結構管理 **>year >month >day >hour >min** 和 **>sec** 指標。

YMDHMS 字的唯一目的是對複雜結構中的指標進行初始化和分組。以下是這些指標的使用方式：

```
建立日期時間
YMDHMS 分配

2022 日期時間 > 年！
03 日期時間 > 月份！
21 日期時間>日！
22 日期時間 > 小時！
36 日期時間 > 分鐘！
15 日期時間 > 秒！

: .日期（日期--）
>r
" 年: " r@ >年@ 。銘 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

```

我們定義了 **DateTime** 這個詞，它是一個由 6 個連續 32 位元單元組成的簡單表。對每個單元格的存取是透過相應的指標來實現的。我們可以使用 **i8** 一詞來重新定義 **YMDHMS 結構** 的分配空間以指向位元組：

```

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

```

在這個 cYMDHMS 結構中，我們將年份保留為 32 位元格式，並將所有其他值減少為 8 位元整數。我們在 .cDate 程式碼中看到，使用指標可以輕鬆存取複雜結構的每個元素...

精靈的定義

我們之前將虛擬螢幕定義為二維數組。此數組的維度由兩個常數定義。這個虛擬螢幕的定義提醒：

```

63 constant SCR_WIDTH
16 constant SCR_HEIGHT
create mySCREEN
    SCR_WIDTH SCR_HEIGHT * allot
mySCREEN SCR_WIDTH SCR_HEIGHT * bl fill

```

使用這種程式方法的缺點是尺寸是在常數中定義的，因此位於表格之外。將表格的尺寸嵌入表格中會更有趣。為此，我們將定義一個適合這種情況的結構：

```

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

```

為了定義軟體精靈，我們將非常簡單地分享這個定義：

```

: 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

```

以下是定義 5 x 7 位元組精靈的方法：

```

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,

```

要從終端機視窗中的 xy 位置顯示精靈，一個簡單的循環就足夠了：

```

: .sprite { xpos ypos sprAddr -- }
  sprAddr >height c@ 0 do
    xpos ypos at-xy
    sprAddr >width c@ i * \ calculate offset in sprite datas
    sprAddr >content + \ calc, real address for line n in 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

```

顯示精靈的結果：

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

The image shows three 3x3 pixel patterns arranged horizontally. The first pattern is black on a white background, forming a shape that resembles a '3'. The second pattern is red on a white background, forming a similar '3' shape. The third pattern is blue on a white background, also forming a similar '3' shape. Each pattern is composed of solid-colored pixels.

我希望本章的內容能給您一些有趣的想法並願意與您分享...

ESP32forth 的實數

如果我們用 FORTH 語言測試運算 **1 3 /**，結果將為 0。

這並不奇怪。基本上，ESP32forth 僅透過資料堆疊使用 32 位元整數。整數具有某些優點：

- 處理速度；
- 計算結果在迭代時沒有漂移風險；
- 幾乎適用於所有情況。

即使在三角計算中，我們也可以使用整數表。只需建立一個包含 90 個值的表，其中每個值對應於角度的正弦值乘以 1000。

但整數也有限制：

- 簡單除法計算的不可能結果，例如我們的 1/3 範例；
- 需要複雜的操作來應用物理公式。

從 7.0.6.5 版本開始，ESP32forth 包含了處理實數的運算子。

實數又稱浮點數。

真正的 ESP32forth

為了區分實數，它們必須以字母 “e” 結尾：

```
3          \ push 3 on the normal stack
3e         \ push 3 on the real stack
5.21e f.   \ display 5.210000
```

就是這個詞。它允許您顯示位於實數堆疊頂部的實數。

ESP32forth 的實數精度

set-precision 一詞可讓您指示小數點後顯示的小數位數。讓我們用常數 **pi** 來看看：

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

ESP32forth 處理實數的極限精度為小數點後六位：

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

如果我們將實數的顯示精度降低到 6 以下，計算仍然會以小數點後 6 位的精度進行。

實數常數和變數

實數常數用單字 **fconstant** 定義：

```
0.693147e fconstant ln2 \ natural logarithm of 2
```

實數變數用單字 **fvariable** 定義：

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

注意：所有實數都通過**實數棧**。對於實數變量，只有指向實數值的位址才會通過資料堆疊。

這個字！將實際值儲存在其記憶體位址指向的位址或變數中。執行實數變數會將記憶體位址放置在經典資料堆疊上。

字 **SF@**堆疊其記憶體位址指向的實際值。

實數的算術運算符

ESP32Forth 有四個算術運算子 **F+ F- F* F/**：

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

ESP32forth 還有這樣的話：

- **1/F** 計算實數的倒數；
- **fsqrt** 計算實數的平方根。

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

實數的數學運算符

ESP32forth 有幾個數學運算子：

- **F**** 計算實數 r_val 的 r_exp 次方
- **FATAN2** 從切線計算弧度角。
- **FCOS** ($r1$ -- $r2$) 計算以弧度表示的角度的餘弦。
- **FEXP** ($\ln-r$ -- r) 計算 $e^{EXP\ r}$ 對應的實數
- **FLN** (r -- $\ln-r$) 計算實數的自然對數。
- **FSIN** ($r1$ - $r2$) 計算以弧度表示的角度的正弦值。
- **FSINCOS** ($r1$ -- $r\cos\ r\sin$) 計算以弧度表示的角度的餘弦和正弦。

一些例子：

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

4.605170e FEXP F.    \ display 100.000018

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

實數上的邏輯運算符

ESP32forth 還允許您對真實資料進行邏輯測試：

- **F0<** (r -- fl) 測試實數是否小於零。
- **F0=** (r -- fl) 表示實數為零時為真。
- **f<** (r1 r2 -- fl) 若 $r1 < r2$ ，則 fl 為真。
- **f<=** (r1 r2 -- fl) 如果 $r1 \leq r2$ ，則 fl 為真。
- **f<>** (r1 r2 -- fl) 如果 $r1 \neq r2$ ，則 fl 為真。
- **f=** (r1 r2 -- fl) 若 $r1 = r2$ ，則 fl 為真。
- **f>** (r1 r2 -- fl) 如果 $r1 > r2$ ，則 fl 為真。
- **f>=** (r1 r2 -- fl) 如果 $r1 \geq r2$ ，則 fl 為真。

整數 ↔ 實數轉換

ESP32forth 有兩個字用於將整數轉換為實數，反之亦然：

- **F>S** (r -- n) 將實數轉換為整數。如果實數有小數部分，則將整數部分保留在資料堆疊上。
- **S>F** (n -- r: r) 將整數轉換為實數並將該實數傳送到實數堆疊。

例：

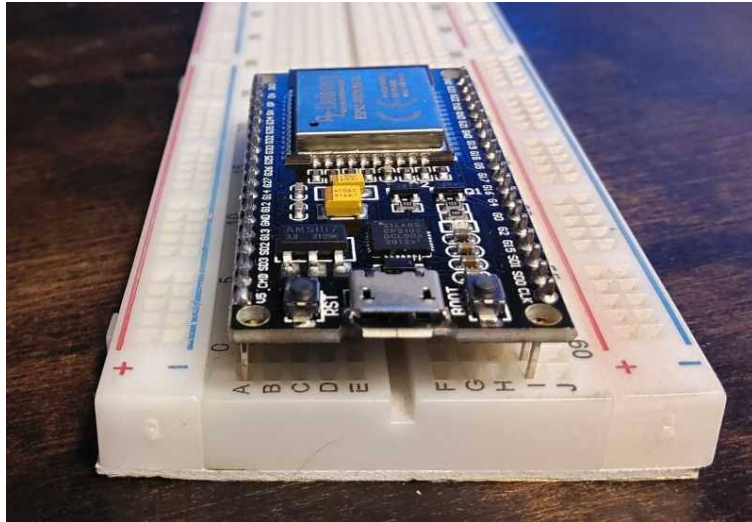
```
35 S>F
F.    \ display 35.000000

3.5e F>S .    \ display 3
```


將麵包板適應 ESP32 板

ESP32 測試板

您剛剛收到 ESP32 卡。第一個糟糕的驚喜是，這張卡在測試板上非常不適合：

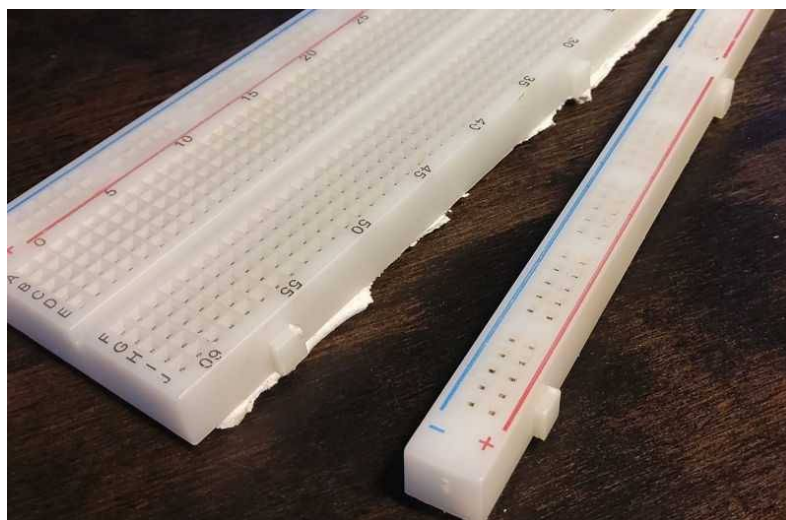


沒有專門適合 ESP32 板的麵包板。

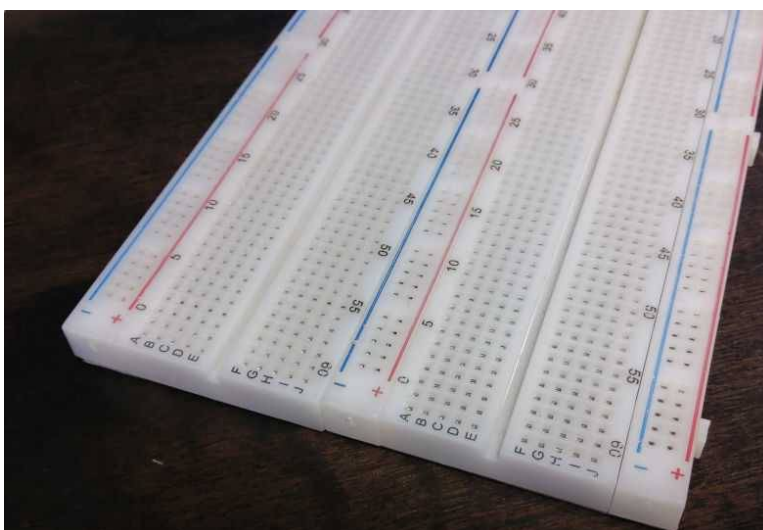
建造適合 ESP32 板的麵包板

我們將建立自己的測試板。為此，您必須有兩個相同的測試板。

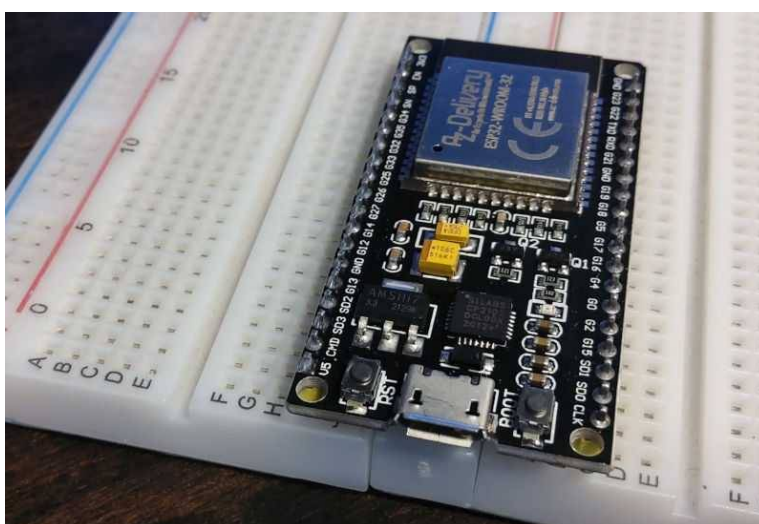
在其中一個板上，我們將拆下一條電源線。為此，請使用刀具從下方切割。您應該能夠像這樣分開這條電源線：



然後我們可以用這張地圖重新組裝整個地圖。測試板的側面有楔子將它們連接在一起：



就這樣吧！現在我們可以放置 ESP32 卡：



I/O 埠可以毫無困難地擴展。

RECORDFILE 和 FORTH 專案管理

本章專門討論一個關鍵元素：**RECORDFILE**。該字允許在 SPIFFS 檔案系統中快速儲存檔案。

Visual 或 **Editor** 操作原始檔之前仔細閱讀它。

以下逐步介紹如何儲存 **RECORDFILE** 的定義，然後有效地使用它。

將 RECORDFILE 儲存在 autoexec.fs 檔案中

autoexec.fs 檔案是否存在。如果此文件存在，則將解釋其內容。

RECORDFILE 的原始碼。這個定義是由鮑伯·愛德華茲提出的。複製此程式碼並將其貼上到終端視窗中進行編譯。此操作只需執行一次：

```
\ 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
;
;
```

編譯該字後，我們將了解如何繼續操作，以便該字可從 **autoexec.fs** 中永久可用。

在您的 PC 上，在專用於 ESP32Forth 的開發區域中，建立一個 **autoexec.fs** 檔案。

上面給出的 **RECORDFILE** 程式碼複製到此 **autoexec.fs** 檔案中。加入這兩行程式碼：

```

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>

```

再次複製此原始碼，包括紅色程式碼行。再次將此程式碼貼到終端機視窗中。將此程式碼傳輸到 ESP32 板。

與編譯程式碼的第一次操作不同，這次程式碼保存在 **/spiffs/autoexec.fs** 檔案中。

autoexec.fs 檔案是否已儲存，請執行 **ls**：

```
ls /spiffs/
```

autoexec.fs 檔案應該會出現在檔案清單中。若要檢查 **autoexec.fs** 的內容，請鍵入：

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

autoexec.fs 的內容。

使用 autoexec.fs 檔案的修改內容

重新啟動 ESP32forth。如果一切順利，**RECORDFILE** 現在可以在 ESP32forth 啟動時使用。跑的話。您應該在 FORTH 字典的第一個單字中找到 **RECORDFILE**：

```
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....
```

不要將 **autoexec.fs** 與其他定義混淆。我們將看到如何建立一個專案。

使用 ESP32forth 分解項目

ESP32forth 的 FORTH 開發專案已在您的 PC 上建立：

- 使用您選擇的文字編輯器或 IDE（例如 Netbeans）編輯原始程式碼；
- 有一個透過 USB 連接到 ESP32 卡的終端機；
- 在 ESP32 板上啟用 ESP32forth。

在 PC 上，以結構化方式運作。以下解釋僅是建議。

首先定義所有 ESP32forth 開發的通用工作目錄。例如，名為 **ESP32forthdevelopments** 的資料夾。

然後，在此資料夾中建立兩個附加資料夾：

- **_我的項目**，旨在容納您的所有項目；
- **_sandbox**，用於接收所有要測試的小程序，沒有特定用途；
- 旨在容納所有普遍感興趣的來源文件的工具。這些是經過測試的文件，不需要適應；
- 用於任何類型文檔的文檔。

範例專案

我將使用 TEMPVS FVGIT 原始碼作為範例專案。完整的源代碼可以在這裡找到：

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

要建立的第一個檔案稱為 **main.fs**。該檔案必須寫入 TEMPVS FVGIT 資料夾中：

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

再次強調，這些只是建議。主要興趣是將單一項目的所有組件組合在一起。 **main.fs** 文件的內容：

```
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>
```

我們找到紅色的單字 **RECORDFILE**。要將 **main.fs** 中的程式碼儲存到 ESP32 板上的 SPIFFS 檔案系統，只需複製此原始程式碼並使用終端程式將其傳遞到 ESP32forth 即可。

藍色部分，在上面的程式碼中，**main.fs** 的內容呼叫 **strings.fs** 檔案。該文件的源代碼來自 **Tools** 資料夾。它是 **strings.fs** 的副本，然後修改如下：

```
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>
```

複製並傳遞此原始程式碼會在 ESP32 板上的 SPIFFS 檔案系統中建立 **strings.fs** 檔案。

在這個階段，我們開始在 ESP32 卡上有幾個檔案。要編譯所有傳輸的文件，我們只需執行：

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

除了實體空間限制外，ESP32 卡上可以儲存的檔案沒有任何限制。SPIFFS 檔案系統中的可用空間超過 1MB 的記錄空間。

如果您需要修改通用軟體元件的內容，請始終在該元件的來源檔案的副本上執行此操作。請記住對這些修改進行版本和日期。

對於該專案中的每個文件，我們整合了 **RECORDFILE** 及其 **<EOF>** 終止符。

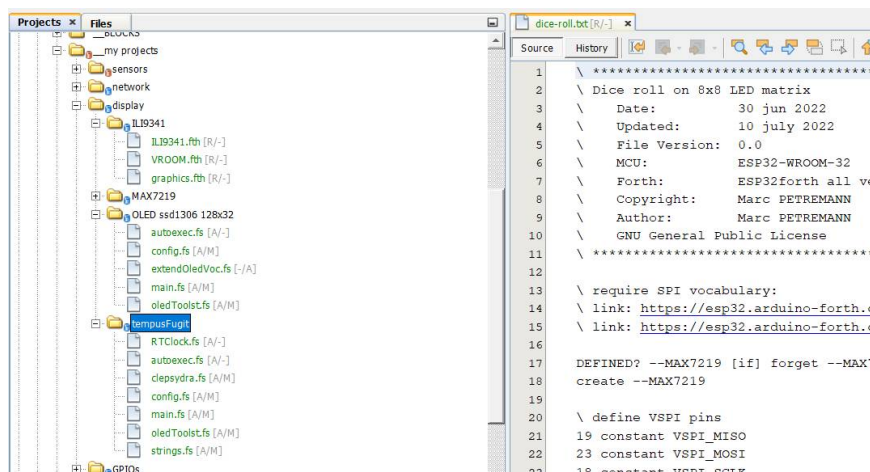


Figure 7: 使用 Netbeans IDE 建置專案

在每個專案中，我們找到 **main.fs** 和 **config.fs** 檔案。但他們的內容會根據每個項目進行調整。對於特定項目，所有 **fs** 擴充檔案都會載入到 SPIFFS 檔案系統中的 ESP32 板上。編譯他們的內容的速度非常快。但最重要的是，這些文件的內容在 ESP32 卡的兩次重啟之間得以保留。FORTH 出現最輕微的阻斷時，可以輕鬆重新啟動卡片並找到項目的所有單字定義，而無需透過終端進行新的傳輸。

黑盒子的概念

這是一個古老的概念，可以追溯到我們主要在微控制器卡上使用彙編器進行開發的時候。我們還在物件程式設計中的類別中發現了它。在「黑盒子」的概念中，我們必須將子程式、函數、方法視為黑盒子。我們知道我們在那裡放了什麼。我們知道它會產生什麼或這個盒子是如何運作的，但我們不擔心它的內部功能。我們信任那些對「黑盒子」進行程式設計的人。

在 FORTH 語言中，一個字有一個定義。當您透過堆疊傳遞參數時，最終只有定義的設計者必須確保定義的正常運作。為了確保定義的正常運行，強烈建議不要定義太長。

我標記已驗證程式碼的技巧就是在定義之前新增註解行。未驗證程式碼範例：

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

程式碼將在沙箱或專案文件中進行測試。沒關係。使用不同的值進行測試後，我修改原始程式碼：

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

如果您懷疑程式碼的可靠性，可以定義一個 **tests.fs** 檔案。

該文件僅用於進行電池單元測試。請參閱斷言（執行這些測試的單字的定義，定義在此處可見：

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

tests.fs 檔案中的測試範例：

```
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(會產生警報。

fpi* 這樣簡單的定義，如果我們沒有使 **f*** 一詞變得可靠，則可能需要進行一系列測試。我們將這些測試整合到 **main.fs** 檔案中：

```
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>
```

tests.fs 檔案的內容也必須傳輸到 ESP32 卡。

這樣，完整的編譯週期還包括一系列測試。測試並不能保證程式碼是可靠的。只有當我們必須修改部分應用程式程式碼時，它們才可以偵測可能的副作用。

總之，我建議透過系統地整合這些文件來分割您的程式碼：

- **main.fs** 這是主文件。通常，無論專案的名稱是什麼，您都可以透過簡單執行 **include /spiffs/main.fs** 來編譯它
- **config.fs** 包含全域設定參數，例如 WiFi 存取密碼；
- **test.fs** 包含一組測試。如果您不進行任何測試，則無需建立此文件。

除未由 **ESP32forth** 處理的檔案外，所有其他檔案都將具有副檔名 **fs** 。

透過遵循這幾個準則，您將可以更輕鬆地管理複雜的應用程式。每次編譯可靠的程式碼部分並將其保存在 **SPIFFS** 檔案系統的檔案中時，節省時間是採用 **RECORDFILE** 的主要理由。

新增 SPI 庫

ESP32forth 中並未原生實作 SPI 函式庫。要安裝它，您必須先建立 **spi.h** 文件，該文件必須安裝在與包含 **ESP42forth.ino** 文件的資料夾相同的資料夾中。

spi.h 文件內容（C 語言）：

```
# 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); DROPn(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); DROPn(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); DROPn(3)) \
    XV(spi, "SPI.transferBits", SPI_TRANSFER_BITES, SPI.transferBits((uint32_t) n2, (uint32_t *) n1, (uint8_t) n0); DROPn(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); DROPn(2)) \
    XV(spi, "SPI.writePixels", SPI_WRITE_PIXELS, SPI.writePixels((const void *) n1, (uint32_t) n0); DROPn(2)) \
    XV(spi, "SPI.writePattern", SPI_WRITE_PATTERN, SPI.writePattern((const uint8_t *) n2, (uint8_t) n1, (uint32_t) n0); DROPn(3))

const char spi_source[] = R"""(
vocabulary spi    spi definitions
transfer spi-builtins
forth definitions
)""";
```

完整文件也可以在這裡找到：<https://github.com/MPETREMANNN11/ESP32forth/blob/main/optional/spi.h>

ESP32forth.ino 檔案的更改

ESP32forth.ino 檔案進行一些更改，則 **spi.h** 檔案的內容無法整合到 **ESP32forth** 中。以下是對此文件進行的一些修改。這些變更是在版本 **7.0.7.15** 上進行的，但應該適用於其他最近或未來的版本。

第一次修改

新增紅色代碼：

```
#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
```

第二次修改

這段程式碼後面加上紅色：

```
// 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
```

第三次修改

添加紅色：

```
#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
```

第四次修改

添加紅色：

```
internals DEFINED? oled-source [IF]
  oled-source evaluate
[THEN] forth

internals DEFINED? spi-source [IF]
  spi-source evaluate
[THEN] forth
```

如果您仔細遵循這些說明，您將能夠使用 ARDUINO IDE 編譯 ESP32forth 並將其上傳到 ESP32 開發板。完成這些操作後，啟動終端。您需要找到 ESP32forth 歡迎提示。類型：

大三角帆列表

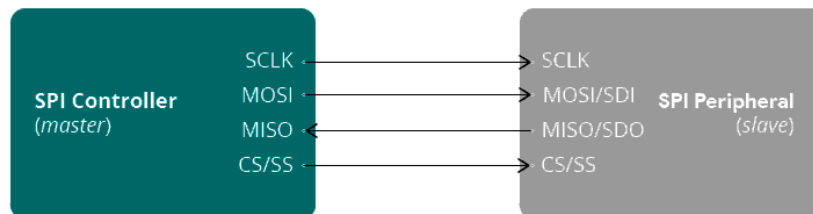
spi 詞彙表中定義的單字：

```
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.writel6
SPI.write32 SPI.writeBytes SPI.writePixels SPI.writePattern spi-builtins
```

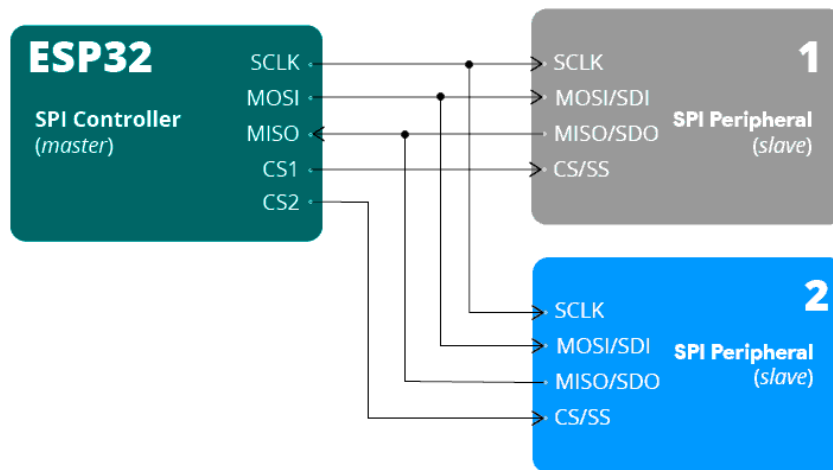
現在您可以透過 SPI 連接埠驅動擴展，例如 MAX7219 LED 顯示器。

與 MAX7219 顯示模組通信

在 SPI 通訊中，總有一個控制外設的主機（也稱為從機）。資料可以同時發送和接收。這意味著主機可以向從機發送數據，並且從機可以同時向主機發送數據。



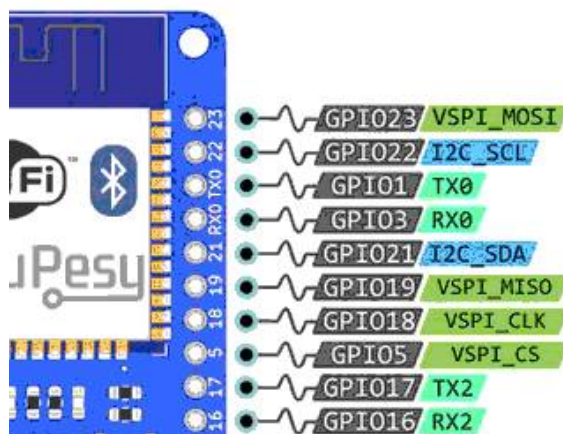
你可以有幾個奴隸。從設備可以是感測器、顯示器、microSD 卡等，或其他微控制器。這意味著您可以將 ESP32 連接到多個裝置。



透過將 CS1 或 CS2 選擇器設定為低電位來選擇從機。有多少從屬設備需要管理，就需要多少個 CS 選擇器。

找到 ESP32 板上的 SPI 端口

ESP32 板上有兩個 SPI 連接埠：HSPI 和 VSPI。我們將管理的 SPI 連接埠是引腳前綴為 VSPI 的連接埠：



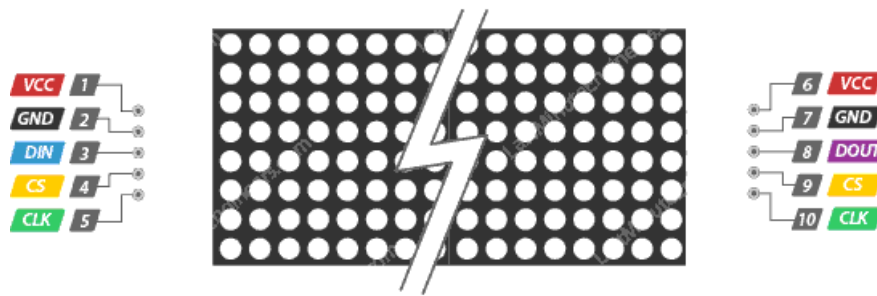
因此，使用 ESP32forth，我們可以定義指向這些 VSPI 引腳的常數：

```
\ 定義 VSPI 腳
19 constant VSPI_MISO
23 constant VSPI_MOSI
18 constant VSPI_SCLK
05 constant VSPI_CS
```

為了與 MAX7219 顯示模組通信，我們只需連接 VSPI_MOSI、VSPI_SCLK 和 VSPI_CS 引腳。

MAX7219 顯示模組上的 SPI 連接器

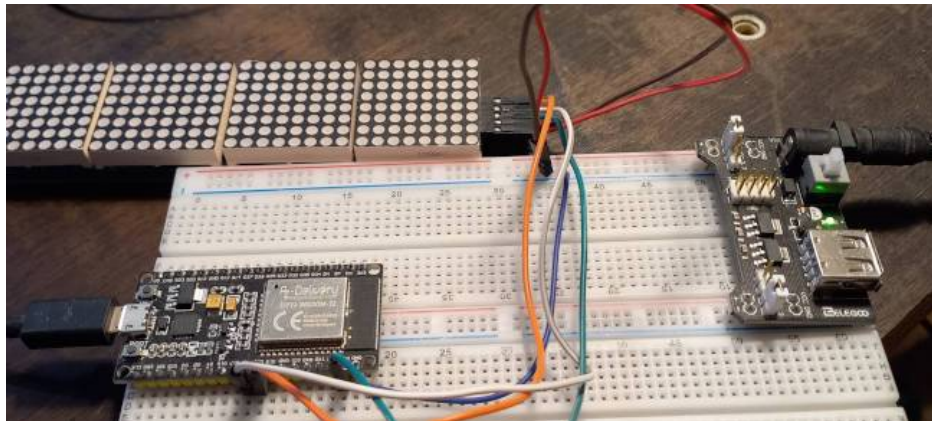
以下是 MAX7219 模組上的 SPI 連接埠連接器圖：



MAX7219 模組與 ESP32 卡之間的連接：

MAX7219		ESP32
DIN	<---->	VSPI_MOSI
CS	<---->	VSPI_CS
CLK	<---->	VSPI_SCLK

VCC 和 GND 連接器連接到外部電源：



此外部電源的 GND 部分與 ESP32 卡的 GND 引腳共用。

SPI 埠軟體層

所有用於管理 SPI 連接埠的單字都已在 **spi** 詞彙表中可用。

唯一需要定義的是 SPI 埠的初始化：

```
\ 定義 SPI 連接埠頻率
4000000 constant SPI_FREQ

\選擇 SPI 詞彙
only FORTH SPI also

\ 初始化 SPI 端口
: 初始化.VSPI ( -- )
: init.VSPI ( -- )
  VSPI_CS OUTPUT pinMode
  VSPI_SCLK VSPI_MISO VSPI_MOSI VSPI_CS SPI.begin
  SPI_FREQ SPI.setFrequency
;
```

現在我們可以使用 MAX7219 顯示模組了。

安裝 HTTP 用戶端

編輯 ESP32forth.ino 文件

ESP32Forth 以原始檔案形式提供，由 C 語言編寫，該檔案必須使用 ARDUINO IDE 或任何其他與 ARDUINO 開發環境相容的 C 編譯器進行編譯。

以下是要修改的程式碼部分。第一部分修改：

```
#define ENABLE_SD_SUPPORT
#define ENABLE_SPI_FLASH_SUPPORT
#define ENABLE_HTTP_SUPPORT
// #define ENABLE_HTTPS_SUPPORT
```

第二部分修改：

```
// .....
#define VOCABULARY_LIST \
  V(forth) V(internals) \
  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)
```

第三部分修改：

```
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
```

第四部分修改：

```
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

```

ESP32forth.ino 檔案後，您可以對其進行編譯並將其上傳到 ESP32 開發板。如果一切順利，你應該會有一個新的 **http** 詞彙：

```

http
vlist      \ displays :
HTTP.begin HTTP.doGet HTTP.getPayload HTTP.end http-builtins

```

HTTP 用戶端測試

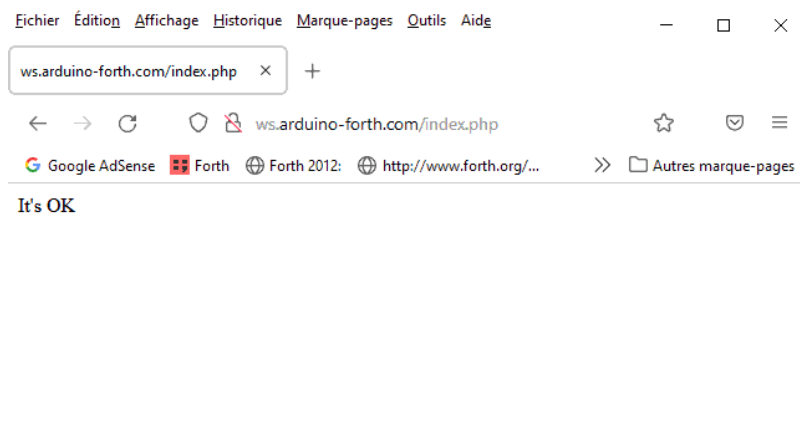
為了測試我們的 HTTP 用戶端，我們可以透過查詢任何 Web 伺服器來完成。但對於我們稍後要考慮的，您需要有一個個人網頁伺服器。在此伺服器上，我們建立一個子網域：

- 我們的伺服器是 arduino-forth.com
- **ws** 子網域
- 我們使用 URL <http://ws.arduino-forth.com> 來訪問該子網域

正在建立此子網域，它不包含任何要執行的腳本。我們創建 **index.php** 頁面並將以下程式碼放在那裡：

```
It's OK
```

要檢查我們的子網域是否正常運行，只需從我們最喜歡的網頁瀏覽器中查詢它：



在我們最喜歡的網頁瀏覽器中顯示「**It's OK**」文字。現在讓我們看看如何從 ESP32Forth 執行相同的伺服器查詢...

以下是快速編寫的用於執行 HTTP 客戶端測試的 FORTH 程式碼：

```
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
;

```

myWiFiConnect 來啟動 Wifi 連接，然後運行：

```

--> myWiFiConnect
192.168.1.23
MDNS started
ok
--> run

Get results: 200
8
It's OK
ok

```

我們的 HTTP 用戶端完美地查詢了 Web 伺服器，顯示了與從 Web 瀏覽器檢索到的文字相同的文字。

這個小小的成功測試為巨大的可能性開闢了道路。

ESP32forth 詞彙詳細內容

ESP32forth 提供了大量詞彙：

- **FORTH** 是主要詞彙；
- 某些詞彙用於 ESP32Forth 的內部機制，例如 **insides** 、 **asm...**
- 許多詞彙允許管理特定連接埠或配件，例如 **藍牙**、 **oled** 、 **spi** 、 **wifi** 、 **wire...**

在這裡您將找到這些不同詞彙表中定義的所有單字的清單。有些單字帶有彩色連結：

align 是一個普通的 FORTH 字；

CONSTANT 是定義詞；

begin 標記控制結構；

key 是延遲執行字；

LED 常數、變數或值定義的詞；

registers 標記一個詞彙。

FORTH 詞彙依字母順序顯示。對於其他詞彙表，單字會以其顯示順序呈現。

Version v 7.0.7.15

FORTH

=	-rot	└	:	:	:noname	!
?	?do	?dup	└	└"	.s	└
(local)	[[']	[char]	[ELSE]	[IF]	[THEN]
l	f	f	}transfer	@	*	*/
*/MOD	/	/mod	#	#!	#>	#fs
#s	#tib	+	+!	+loop	+to	<
<#	<=	<>	=	>	>=	>BODY
>flags	>flags&	>in	>link	>link&	>name	>params
>R	>size	0<	0<>	0=	1-	1/F
1+	2!	2@	2*	2/	2drop	2dup
4*	4/	abort	abort"	abs	accept	adc
afliteral	aft	again	ahead	align	aligned	allocate
allot	also	analogRead	AND	ansi	ARSHIFT	asm
assert	at-xy	base	begin	bq	BIN	binary
bl	blank	block	block-fid	block-id	buffer	bye
c,	C!	C@	CASE	cat	catch	CELL
cell/	cell+	cells	char	CLOSE-DIR	CLOSE-FILE	cmove
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<	F<=	F<>	F=	F>	F>=
F>S	F0<	F0=	FABS	FATAN2	fconstant	FCOS
fdepth	FDROP	FDUP	FEXP	fq	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!
FP@	fp0	free	freq	FROT	FSIN	FSINCOS
FSQRT	FSWAP	fvariable	handler	here	hex	HIGH
hld	hold	httpd	I	if	IMMEDIATE	include
included	included?	INPUT	internals	invert	is	J
K	key	key?	L!	latestxt	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>	r
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 disasm1 matchit address istep sextend m. m@ for-ops op >operands
>mask >pattern >length >xt op-snap opcodes coden, names operand l 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 READDIR CALLCODE 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  
DONEXT 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\_NEGEDGE #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.LW, 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, BEQ, 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 iiii 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  
SO\_L\_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
```

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  
typer 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.beginTransaction 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, RDTLB1, RDTLB0, 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.T,  
MOVSP, MOV.T.S, MOV.F.S, MOV.GEZ.S, MOV.LTZ.S, MOV.NEZ.S, MOV.EQZ.S, U.L.S, O.L.S,  
U.LT.S, O.LT.S, U.EQ.S, O.EQ.S, U.N.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 [MOV.F](#), 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, S32CI, [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, DPFR, 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 iiii [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](#)

資源

用英語

- **ESP32forth** 頁面由 ESP32forth 的創建者 Brad NELSON 維護。您將在那裡找到所有版本（ESP32、Windows、Web、Linux...）
<https://esp32forth.appspot.com/ESP32forth.html>

-

法語

- **ESP32 Forth** 網站有兩種語言（法語、英語），有許多範例
<https://esp32.arduino-forth.com/>

GitHub

- 尤福斯 資源由 Brad NELSON 維護。包含 ESP32forth 的所有 Forth 和 C 語言原始檔
<https://github.com/flagxor/ueforth>
- **ESP32forth** ESP32forth 的原始碼和文件。 Marc PETREMANN 維護的資源
<https://github.com/MPETREMANN11/ESP32forth>
- **ESP32forthStation** 資源由 Ulrich HOFFMAN 維護。帶有 LillyGo TTGO VGA32 單板計算機和 ESP32forth 的獨立 Forth 計算機。
<https://github.com/uho/ESP32forthStation>
- **ESP32Forth** FJ RUSSO 維護的資源
<https://github.com/FJRusso53/ESP32Forth>
- **esp32forth** 插件 Peter FORTH 維護的資源
<https://github.com/PeterForth/esp32forth-addons>
- **Esp32forth-org** Forth2020 和 ES32forth 組成員的程式碼儲存庫
<https://github.com/Esp32forth-org>
-

詞彙索引

1/F.....	67	httpd.....	90	spi.....	92
and.....	33	insides.....	90	SPI.....	78
asm.....	89	internals.....	90	SPIFFS.....	92
binary.....	29	interrupts.....	91	streams.....	92
bluetooth.....	89	ledc.....	91	struct.....	61
decimal.....	29	normal.....	49	structures.....	61, 63, 92
drop.....	46	oled.....	91	tasks.....	92
dump.....	43	page.....	49	telnetd.....	93
dup.....	46	RECORDFILE.....	72	to.....	48, 54
editor.....	90	registers.....	91	u.....	32
ESP.....	90	riscv.....	91	value.....	48
F-.....	67	rot.....	46	variable.....	47
F*.....	67	rtos.....	92	visual.....	93
F/.....	67	SD.....	92	web-interface.....	93
F+.....	67	SD_MMC.....	92	WiFi.....	93
fconstant.....	67	see.....	43	Wire.....	93
FORTH.....	88	Serial.....	92	xtensa.....	93
fsqrt.....	67	set- precision.....	66	.s.....	43
fvariable.....	67	shift.....	33	+to.....	54
hex.....	29	sockets.....	92		