

El gran libro por ESP32forth

versión 1.6 - 7. enero 2024



Autor

- Marc PETREMANN petremann@arduino-forth.com

Colaboradores

- Vaclav POSSELT
- Bob EDWARDS

Índice

Autor.....	1
Colaboradores.....	1
Introducción.....	7
Ayuda de traducción.....	7
Descubrimiento de la tarjeta ESP32.....	8
Presentación.....	8
Puntos fuertes.....	8
Entradas/salidas GPIO en ESP32.....	9
Periféricos ESP32.....	11
Las diferentes tarjetas ESP32.....	12
Instalación final de ESP32forth.....	13
La tarjeta ESP32 Wroom 32.....	13
Placa de conector.....	14
La placa ESP32 Wrover.....	15
Placa de conector.....	15
La tarjeta ESP32 S3.....	16
Placa de conector.....	17
Instalar ESP32Forth.....	18
Descargar ESP32forth.....	18
Compilando e instalando ESP32forth.....	18
Configuraciones para ESP32 WROOM.....	20
Iniciar la compilación.....	21
Solucionar el error de conexión de carga.....	22
¿Por qué programar en lenguaje FORTH en ESP32?.....	24
Preámbulo.....	24
Límites entre lenguaje y aplicación.....	24
¿Qué es una CUARTA palabra?.....	25
¿Una palabra es una función?.....	25
Lenguaje FORTH comparado con el lenguaje C.....	26
Qué le permite hacer FORTH en comparación con el lenguaje C.....	27
Pero ¿por qué una pila en lugar de variables?.....	28
¿Estás convencido?.....	28
¿Hay alguna solicitud profesional escrita en FORTH?.....	28
Usando números con ESP32Forth.....	31
Números con el intérprete FORTH.....	31
Ingresar números con diferentes bases numéricas.....	32
Cambio de base numérica.....	32
Binario y hexadecimal.....	33
Tamaño de los números en la pila de datos FORTH.....	35
Acceso a memoria y operaciones lógicas.....	37
Un verdadero FORTH de 32 bits con ESP32Forth.....	39

Valores en la pila de datos.....	39
Valores en la memoria.....	39
Procesamiento de textos dependiendo del tamaño o tipo de datos.....	40
Conclusión.....	41
Comentarios y aclaraciones.....	43
Escribir código ADELANTE legible.....	43
Sangría del código fuente.....	44
Los comentarios.....	45
Comentarios de pila.....	45
Significado de los parámetros de la pila en los comentarios.....	46
Definición de palabras Comentarios de palabras.....	47
Comentarios textuales.....	47
Comentario al principio del código fuente.....	48
Herramientas de diagnóstico y ajuste.....	48
El descompilador.....	48
Volcado de memoria.....	49
Monitor de pila.....	49
Diccionario / Pila / Variables / Constantes.....	51
Ampliar diccionario.....	51
Gestión de diccionarios.....	51
Pilas y notación polaca inversa.....	52
Manejo de la pila de parámetros.....	53
La pila de retorno y sus usos.....	53
Uso de memoria.....	54
Variables.....	54
Constantes.....	55
Valores pseudoconstantes.....	55
Herramientas básicas para la asignación de memoria.....	55
Colores de texto y posición de visualización en el terminal.....	57
Codificación ANSI de terminales.....	57
Coloración de texto.....	58
Posición de visualización.....	59
Variables locales con ESP32Forth.....	61
Introducción.....	61
El comentario de la pila falsa.....	61
Acción sobre variables locales.....	62
Estructuras de datos para ESP32forth.....	65
Preámbulo.....	65
Tablas en FORTH.....	65
Matriz de datos unidimensional de 32 bits.....	65
Palabras de definición de tabla.....	66
Leer y escribir en una tabla.....	66
Ejemplo práctico de gestión de una pantalla virtual.....	67
Gestión de estructuras complejas.....	70
Definición de sprites.....	72

Instalación de la biblioteca OLED para SSD1306.....	75
Números reales con ESP32 en adelante.....	78
Los reales con ESP32 en adelante.....	78
Precisión de números reales con ESP32forth.....	78
Constantes y variables reales.....	79
Operadores aritméticos en números reales.....	79
Operadores matemáticos sobre números reales.....	80
Operadores lógicos en números reales.....	80
Entero ↔ transformaciones reales.....	80
Mostrar números y cadenas de caracteres.....	82
Cambio de base numérica.....	82
Definición de nuevos formatos de visualización.....	83
Mostrar caracteres y cadenas de caracteres.....	85
Variables de cadena.....	87
Código de palabra de gestión de variables de texto.....	88
Agregar carácter a una variable alfanumérica.....	90
Vocabularios con ESP32forth.....	91
Lista de vocabularios.....	91
Vocabularios esenciales.....	92
Lista de contenidos de vocabulario.....	92
Usando palabras de vocabulario.....	92
Encadenamiento de vocabularios.....	93
Palabras de acción retrasada.....	95
Definición y uso de palabras con defer.....	96
Establecer una referencia directa.....	96
Dependencia del contexto operativo.....	97
Un caso práctico.....	98
Palabras de creación de palabras.....	101
Usando does>.....	101
Ejemplo de gestión del color.....	102
Ejemplo, escribir en pinyin.....	103
Adaptar placas de pruebas a la placa ESP32.....	105
Placas de prueba para ESP32.....	105
Construya una placa de pruebas adecuada para la placa ESP32.....	105
Alimentando la placa ESP32.....	107
Elección de la fuente de energía.....	107
Alimentado por conector mini-USB.....	107
Alimentación mediante pin de 5V.....	107
Inicio automático de un programa.....	109
Instale y use la terminal Tera Term en Windows.....	111
Instalar Tera Term.....	111
Configuración de Tera Term.....	111
Usando el término Tera.....	114
Compilar código fuente en lenguaje Forth.....	115

Acceder a ESP32Forth por TELNET.....	117
Cambiar el nombre DNS de la placa ESP32.....	117
Conexión a placas ESP32 por su nombre de host.....	118
Gestión de proyectos RECORDFILE y FORTH.....	121
Guarda RECORDFILE en el archivo autoexec.fs.....	121
Utilice contenidos modificados del archivo autoexec.fs.....	123
Desglosando un proyecto con ESP32forth.....	123
Proyecto de ejemplo.....	124
La noción de caja negra.....	126
Instalación de la biblioteca OLED para SSD1306.....	129
Instalación del cliente HTTP.....	131
Editando el archivo ESP32forth.ino.....	131
Prueba de cliente HTTP.....	132
El generador de números aleatorios.....	135
Característica.....	135
Procedimiento de programación.....	136
Función RND en ensamblador XTENSA.....	136
Contenido detallado de los vocabularios ESP32forth.....	138
Version v 7.0.7.15.....	138
FORTH.....	138
asm.....	139
bluetooth.....	140
editor.....	140
ESP.....	140
httpd.....	140
insides.....	140
internals.....	140
interrupts.....	141
ledc.....	141
oled.....	141
registers.....	141
riscv.....	141
rtos.....	142
SD.....	142
SD_MMC.....	142
Serial.....	142
sockets.....	142
spi.....	142
SPIFFS.....	142
streams.....	142
structures.....	143
tasks.....	143
telnetd.....	143
visual.....	143
web-interface.....	143
WiFi.....	143

Wire.....	143
xtensa.....	143
Annexe A – Sommaire des registres.....	144
GPIO registers.....	144
Recursos.....	148
En inglés.....	148
En francés.....	148
GitHub.....	148

Introducción

Desde 2019 he gestionado varios sitios web dedicados al desarrollo del lenguaje FORTH para placas ARDUINO y ESP32, así como la versión web eForth :

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

Estos sitios están disponibles en dos idiomas, francés e inglés. Cada año pago por el alojamiento del sitio principal. **arduino-forth.com**.

Tarde o temprano –y lo más tarde posible– sucederá que ya no podré garantizar la sostenibilidad de estos lugares. La consecuencia será que la información difundida por estos sitios desaparecerá.

Este libro es la recopilación del contenido de mis sitios web. Se distribuye gratuitamente desde un repositorio de Github. Este método de distribución permitirá una mayor sostenibilidad que los sitios web.

De paso, si algunos lectores de estas páginas desean hacer su aporte, son bienvenidos. :

- para sugerir capítulos ;
- para informar errores o sugerir cambios ;
- para ayudar con la traducción...

Ayuda de traducción

Google Translate te permite traducir textos fácilmente, pero con errores. Por eso pido ayuda para corregir las traducciones.

En la práctica, proporciono los capítulos ya traducidos en formato LibreOffice. Si desea ayudar con estas traducciones, su función será simplemente corregir y devolver estas traducciones.

Corregir un capítulo lleva poco tiempo, de una a unas pocas horas.

Para contactar conmigo : petremann@arduino-forth.com

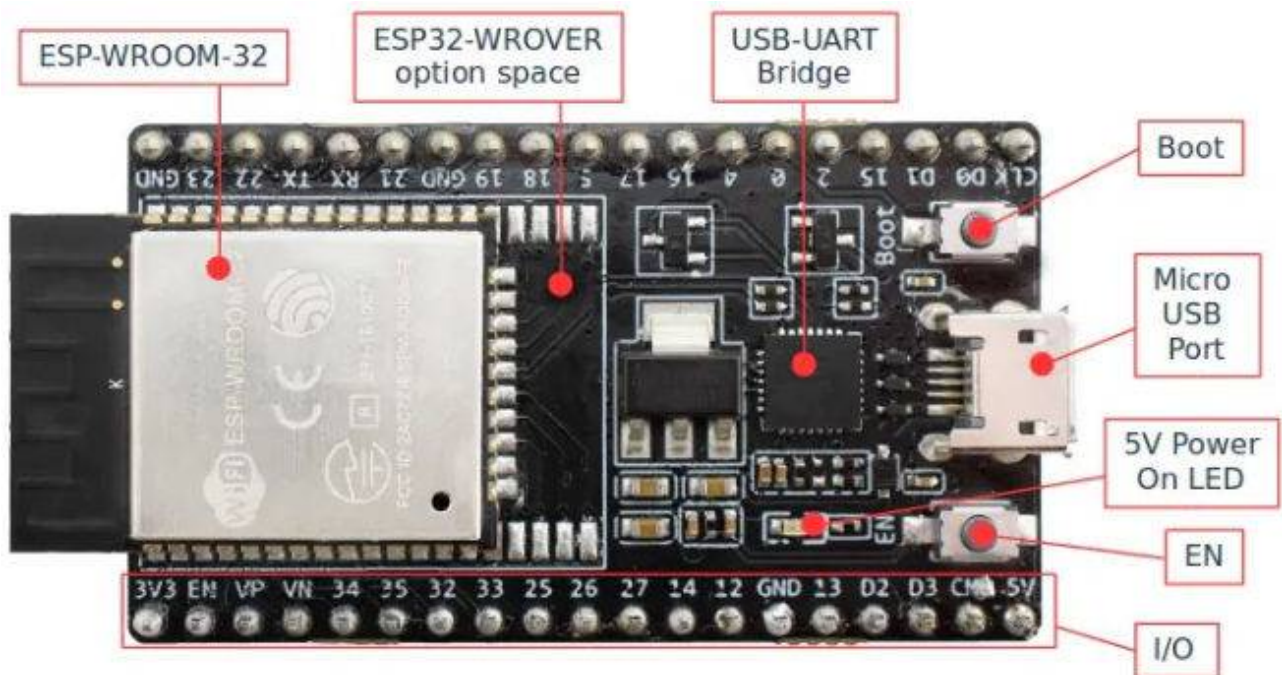
Descubrimiento de la tarjeta ESP32

Presentación

La placa ESP32 no es una placa ARDUINO. Sin embargo, las herramientas de desarrollo aprovechan ciertos elementos del ecosistema ARDUINO, como ARDUINO IDE.

Puntos fuertes

En cuanto al número de puertos disponibles, la tarjeta ESP32 se sitúa entre un ARDUINO



NANO y un ARDUINO UNO. El modelo básico tiene 38 conectores :

Los dispositivos ESP32 incluyen :

- 18 canales de convertidor analógico a digital. (ADC)
- 3 interfaces SPI
- 3 interfaces UART
- 2 interfaces I2C
- 16 canales de salida PWM
- 2 convertidores de digital a analógico (DAC)
- 2 interfaces I2S

- 10 GPIO de detección capacitiva

La funcionalidad ADC (convertidor analógico a digital) y DAC (convertidor digital a analógico) están asignadas a pines estáticos específicos. Sin embargo, puedes decidir qué pines son UART, I2C, SPI, PWM, etc. Sólo necesitas asignarlos en el código. Esto es posible gracias a la función de multiplexación del chip ESP32.

La mayoría de los conectores tienen múltiples usos.

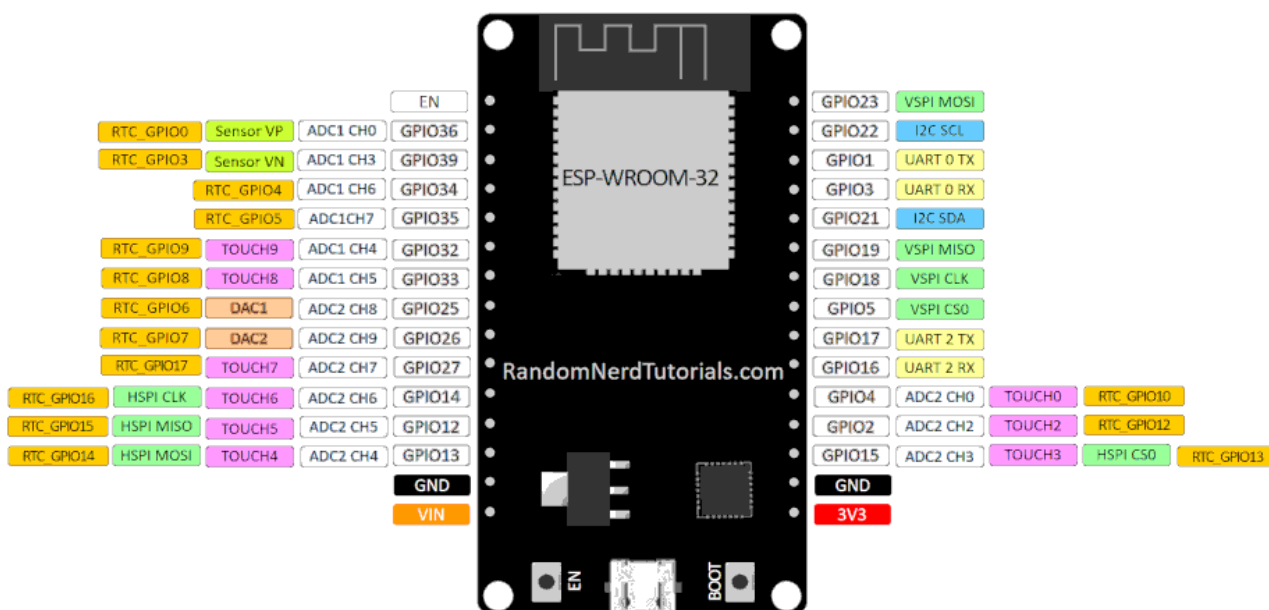
Pero lo que distingue a la placa ESP32 es que está equipada de serie con soporte WiFi y Bluetooth, algo que las placas ARDUINO sólo ofrecen en forma de extensiones.

Entradas/salidas GPIO en ESP32

Aquí, en foto, la tarjeta ESP32 desde la que explicaremos el papel de las diferentes entradas/salidas GPIO :



La posición y la cantidad de E/S GPIO pueden cambiar según la marca de la tarjeta. Si este es el caso, sólo son auténticas las indicaciones que aparecen en el mapa físico. En la foto, fila inferior, de izquierda a derecha : CLK, SD0, SD1, G15, G2, G0, G4, G16.....G22, G23, GND.



En este diagrama, vemos que la fila inferior comienza con 3V3 mientras que en la foto, esta E/S está al final de la fila superior. Por lo tanto, es muy importante no confiar en el diagrama y, en su lugar, verificar la correcta conexión de los periféricos y componentes en la tarjeta física ESP32.

Las placas de desarrollo basadas en un ESP32 generalmente tienen 33 pines aparte de los de la fuente de alimentación. Algunos pines GPIO tienen funciones un tanto particulares :

GPIO	Posibles nombres
6	SCK/CLK
7	SCK/CLK
8	SDO/SD0
9	SDI/SD1
10	SHD/SD2
11	CSC/CMD

Si tu tarjeta ESP32 tiene E/S GPIO6, GPIO7, GPIO8, GPIO9, GPIO10, GPIO11, definitivamente no debes usarlas porque están conectadas a la memoria flash del ESP32. Si los usas el ESP32 no funcionará.

Las E/S GPIO1(TX0) y GPIO3(RX0) se utilizan para comunicarse con la computadora en UART a través del puerto USB. Si los utilizas, ya no podrás comunicarte con la tarjeta.

GPIO36(VP), GPIO39(VN), GPIO34, GPIO35 I/O se pueden utilizar solo como entrada. Tampoco tienen resistencias pullup y pulldown internas incorporadas.

El terminal EN le permite controlar el estado de encendido del ESP32 a través de un cable externo. Está conectado al botón EN de la tarjeta. Cuando el ESP32 está encendido, está a 3,3 V. Si conectamos este pin a tierra el ESP32 se apaga. Puedes usarlo cuando el ESP32 está en una caja y quieres poder encenderlo/apagarlo con un interruptor.

Periféricos ESP32

Para interactuar con módulos, sensores o circuitos electrónicos, el ESP32, como cualquier microcontrolador, dispone de multitud de periféricos. Hay más que en una placa Arduino clásica.

ESP32 tiene los siguientes periféricos:

- 3 interfaces UART
- 2 interfaces I2C
- 3 interfaces SPI
- 16 salidas PWM
- 10 sensores capacitivos
- 18 entradas analógicas (ADC)
- 2 salidas DAC

ESP32 ya utiliza algunos periféricos durante su funcionamiento básico. Por lo tanto, hay menos interfaces posibles para cada dispositivo.

Las diferentes tarjetas ESP32

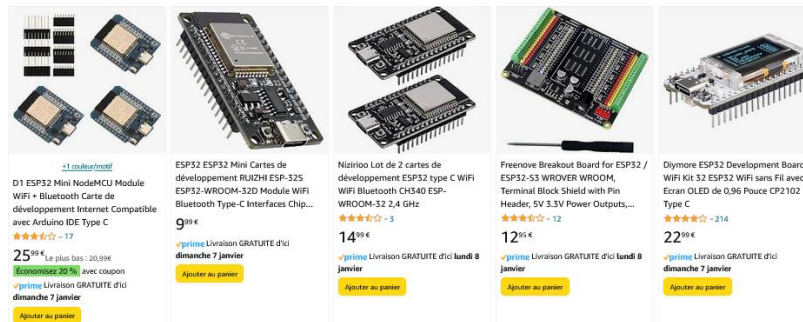


Figure 1: un grand choix de cartes ESP32 sur AMAZON

Si visita un sitio de ventas en línea para solicitar una tarjeta ESP32, puede terminar con una gran variedad de tarjetas.

Por tanto, surgen varias preguntas para orientar la elección:

- ¿Qué placa puede alojar ESP32 en adelante?
- ¿Qué tarjetas se adaptan mejor a mis proyectos?
- ¿Cuál es mi presupuesto para un proyecto determinado?

Si el precio de una tarjeta ESP32 normal sigue siendo asequible, ciertas variantes pueden ver sus precios dispararse. Si tu objetivo es realizar primero pequeños experimentos, empieza con una placa ESP32 sencilla. Para experimentar adecuadamente, necesitará:

- placas de prueba, tome al menos 10. Permita dos placas de prueba por tarjeta ESP32;
- conectores flexibles tipo dupont;
- LED, resistencias, etc.
- periféricos: pantalla OLED, LCD, relés, motores síncronos o ordinarios, servomotores, etc.
- Cable USB que conecta el PC y la tarjeta ESP32. Se recomienda un concentrador USB. En caso de inyección accidental de corriente en el puerto USB, será el puerto USB del hub el que se dañará antes que el puerto USB del PC;

Por unos cincuenta euros (o dólares estadounidenses), hay kits listos para usar, que incluyen una tarjeta ESP32 y periféricos y componentes.

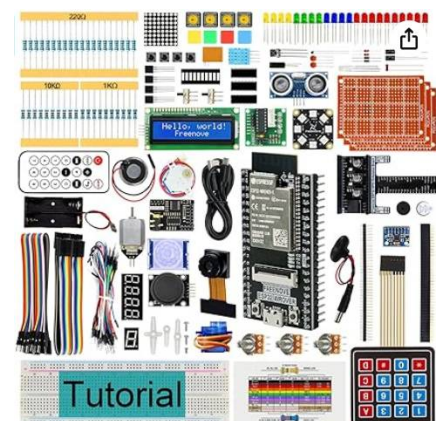


Figure 2: kit ESP32

Ningún kit está completo. Si estás realizando experimentos lo ideal es pedir un kit, luego varias tarjetas ESP32 (al menos 4) una serie de placas de prueba, correas planas, una fuente de alimentación por batería, etc....

Antes de embarcarse en proyectos ambiciosos, como control remoto mediante red 3G/4G/5G, análisis de vídeo, etc. Comience con experimentos simples, en C o ADELANTE.

Instalación final de ESP32forth

NO ! ¡La instalación de ESP32 en adelante en una tarjeta ESP32 **no es permanente** ! Si ha instalado ESP32Forth en una o más placas ESP32, puede descargar fácilmente el código binario desde cualquier fuente C, después de compilar el contenido de un archivo de extensión **ino** , a sus placas ESP32 y abandonar la programación FORTH.

Pero, a riesgo de publicitar ESP32, muchos "creadores" han tomado la decisión definitiva de programar en el lenguaje FORTH. Solo un ejemplo, el canal de YouTube de **0033mer** : <https://www.youtube.com/@0033mer>

Es uno de los colaboradores de FORTH más prolíficos en Youtube. Aunque usa muy poco ESP32, la mayoría de sus contribuciones usan el lenguaje FORTH.

Programar en lenguaje FORTH requiere esfuerzo intelectual. Este esfuerzo no es en vano, porque da lugar a ciertas buenas prácticas que pueden utilizarse en otros lenguajes de programación.

FORTH es el único lenguaje de programación instalable en una tarjeta electrónica y que integra un intérprete, un compilador, un ensamblador, un sistema de archivos SPIFFS, todo ello con un importante espacio de desarrollo.

La tarjeta ESP32 es una de las pocas tarjetas que también tiene capacidades de comunicación en serie (puerto UART0 a través del conector USB), a través de WiFi o Bluetooth. La mayoría de tarjetas también cuentan con numerosos puertos GPIO versátiles: lógicos, analógicos, PWM, UART, SPI, entrada y salida I2C, etc. Y todo ello con uno o dos procesadores a casi 160Mhz, o 10 veces más rápido que en una tarjeta ARDUINO normal. .

Y finalmente, la mayoría de las bibliotecas C para ARDUINO se pueden usar en ESP32. Algunos son accesibles desde ESP32 en adelante.

La tarjeta ESP32 Wroom 32

ESP32 Wroom es la última incorporación a la familia de tarjetas ESP de Espressif. Se trata de una gama de placas de desarrollo especialmente de moda, porque su bajo precio, bajo consumo y reducido tamaño las convierten en un producto ideal para llevar a cabo pequeños proyectos de IoT.

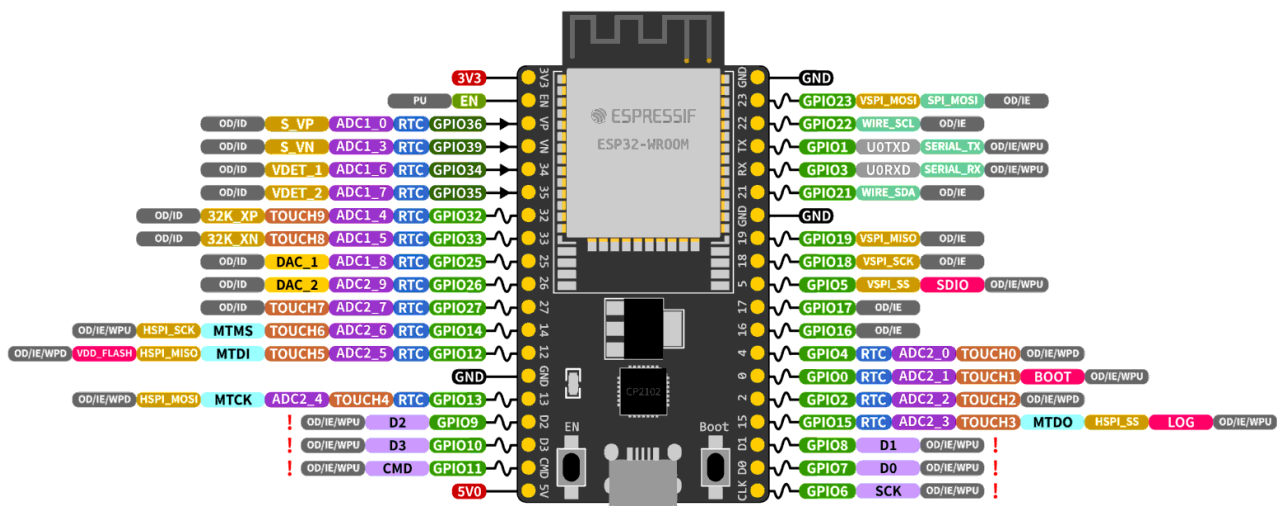


- Interfaz USB a UART CP2102

Si compila ESP32 en adelante para esta placa, estos son los parámetros a considerar en ARDUINO IDE: HERRAMIENTAS → PLACA → ESP32 → Módulo de desarrollo ESP32:

- PSRAM:** Desactivar

Placa de conector



La placa ESP32 Wrover

Los módulos MCU ESP32-WROVER de Espressif Systems son módulos MCU Wi-Fi/BT/BLE potentes y genéricos que se dirigen a una amplia gama de aplicaciones.

Estos módulos están dirigidos a aplicaciones que van desde redes de sensores de bajo consumo hasta las tareas más exigentes, como codificación de voz, transmisión de música y decodificación de MP3.

El módulo ESP32-WROVER usa una antena PCB mientras que el ESP32-WROVER-I usa una antena IPEX.

Estos módulos tienen una Flash SPI externa de 4 MB, una PSRAM externa de 4 MB y una PSRAM SPI de 32 Mbit.

Si compila ESP32 en adelante para esta placa, estos son los parámetros a considerar en ARDUINO IDE: HERRAMIENTAS → PLACA → ESP32 → Módulo de desarrollo ESP32:

- **Junta:** Esquema de partición del módulo de desarrollo ESP32
 - : Sin OTA (Aplicación de 2 M, SPIFFS de 2 M) ← Velocidad de carga **no predeterminada**
 - : 921600
 - Frecuencia de CPU:** 240 MHz
 - Frecuencia de flash:** 80 MHz
 - Modo de flash:** QIO
 - Tamaño de flash:** 4 MB (32 Mb)
 - Nivel de depuración del núcleo:** Ninguno
 - PSRAM:** **Activado**

Placa de conector

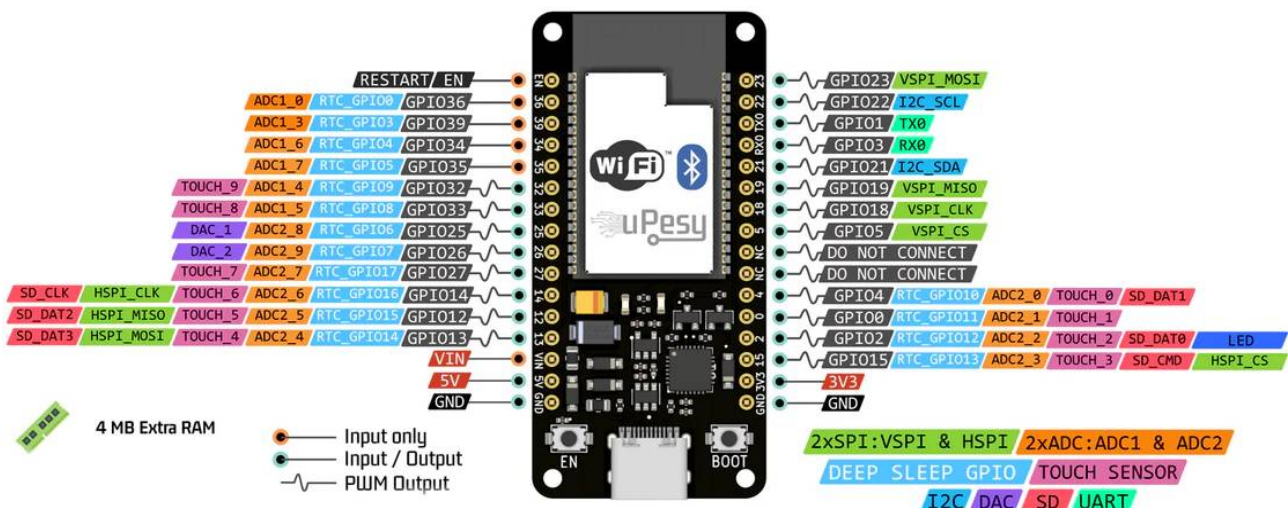


Figure 4: carte ESP32 Wrover

La tarjeta ESP32 S3

ESP32-S3 es una placa de desarrollo basada en un microcontrolador Espressif ESP32-S3-WROOM-2 con interfaces WiFi y Bluetooth Low Energy.

- Microprocesador Xtensa LX7 de doble núcleo de 32 bits
Memoria PSRAM: 8 MB
Memoria SRAM: 512 KB
Memoria ROM: 384 KB
Memoria SRAM (RTC): 16 KB SPI
Memoria FLASH: 32 MB
Interfaz WiFi: 802.11 b/g/n 2.4 GHz
BLE 5 interfaz de malla

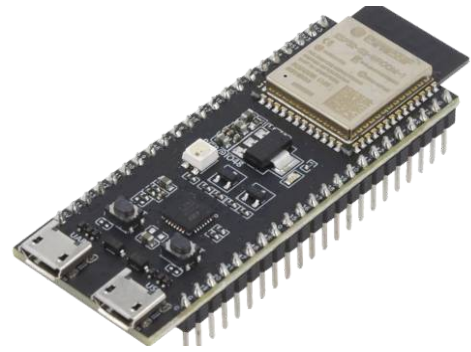
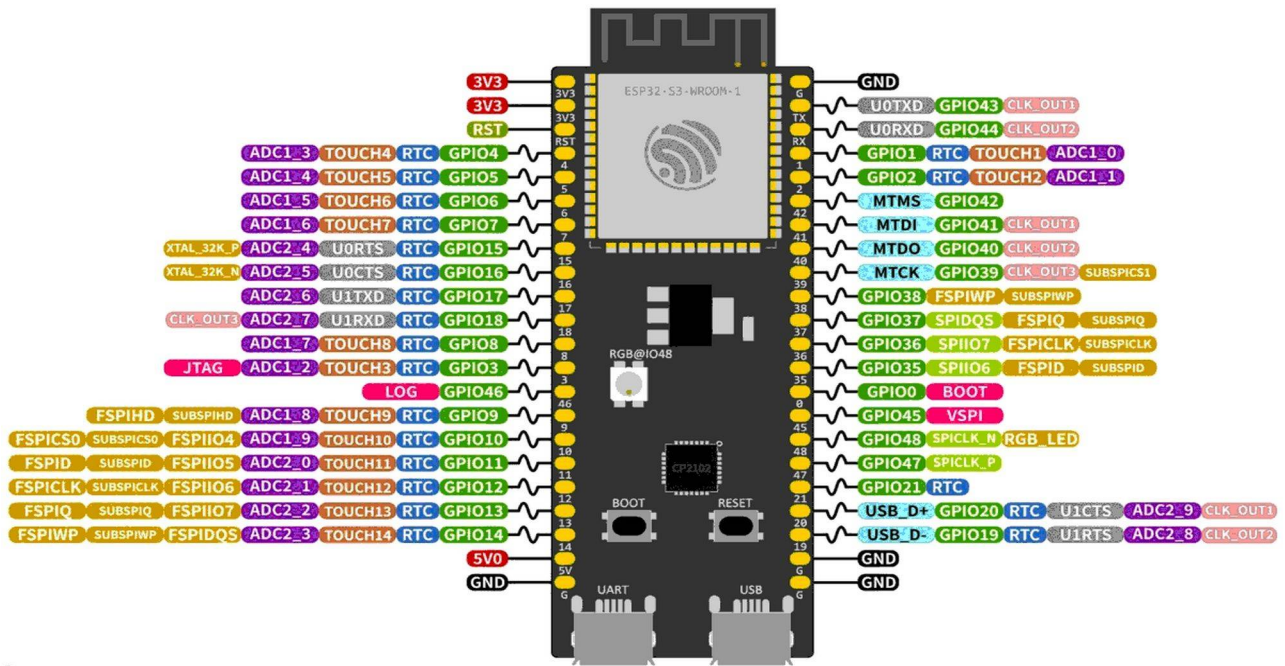


Figure 5: carte ESP32 S3

Si compila ESP32 en adelante para esta placa, estos son los parámetros a considerar en ARDUINO IDE: HERRAMIENTAS → PLACA → ESP32 → Módulo de desarrollo ESP32:

- **Junta:** Esquema de partición del módulo de desarrollo ESP32S2
: Sin OTA (2M APP, 2M SPIFFS) ← Velocidad de carga **no predeterminada**
: 921600
USB CDC Al arrancar: Deshabilitado
Firmware USB MSC Al arrancar: Deshabilitado
USB DFU Al arrancar: Deshabilitado
Modo de carga: UART0
Frecuencia de CPU: 240MHz
Frecuencia de flash: 80MHz
Modo de flash : Tamaño de flash QIO
: 4 MB (32 Mb)
Nivel de depuración del núcleo: Ninguno
PSRAM: Activado

Placa de conector



Instalar ESP32Forth

Descargar ESP32forth

El primer paso consiste en recuperar el código fuente, en lenguaje C, de ESP32forth.

Preferiblemente utilice la versión más reciente:

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

Contenido del archivo descargado:

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

Compilando e instalando ESP32forth

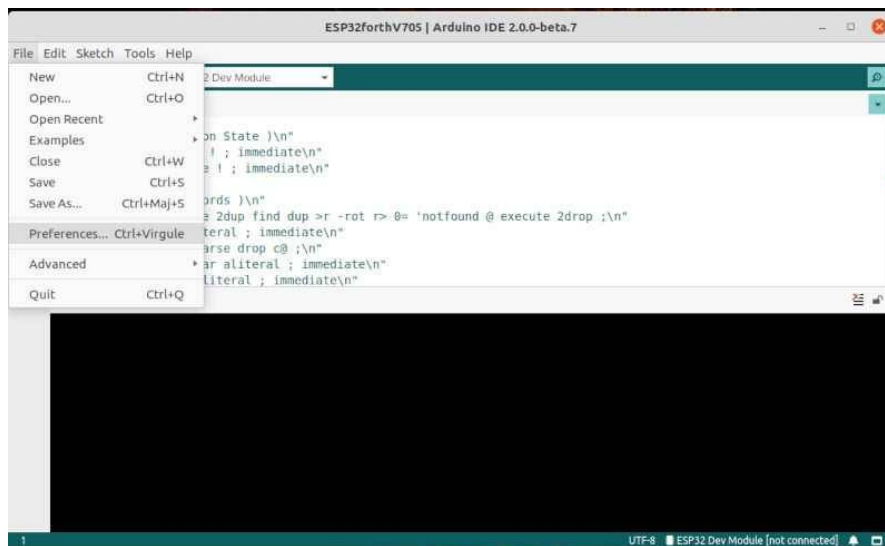
archivo **esp32forth.ino** en un directorio de trabajo. El directorio opcional contiene archivos que permiten la extensión de ESP32 en adelante. Para nuestra primera compilación y carga de ESP32 en adelante, estos archivos no son necesarios.

Para compilar ESP32 en adelante, debe tener ARDUINO IDE ya instalado en su computadora:

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

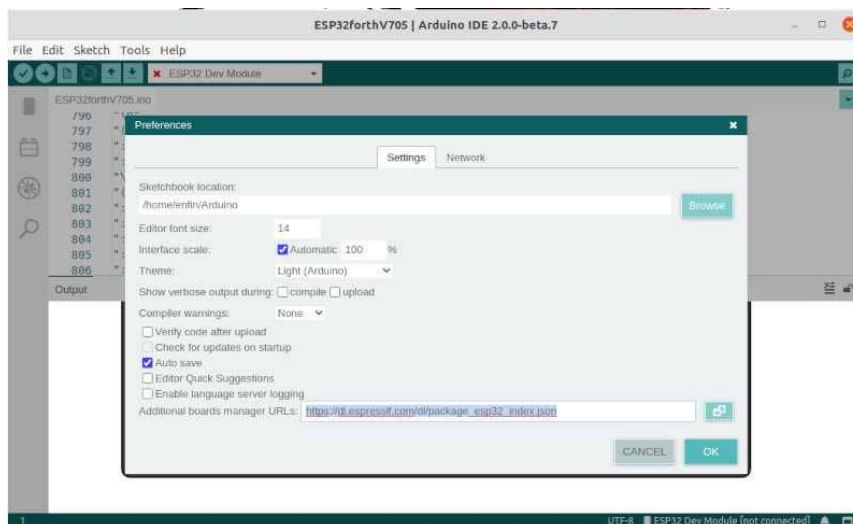
Una vez instalado ARDUINO IDE, ejecútelo. ARDUINO IDE está abierto, aquí la versión 2.0¹. Haga clic en *file* y seleccione *Preferences* :

¹ Nota sobre las siguientes versiones de ESP32: la llamada versión estable 7.0.6.19 necesita para una compilación correcta las bibliotecas de la placa Espressif 1.0.6, la versión reciente 7.0.7.15 necesita las bibliotecas 2.0.x.

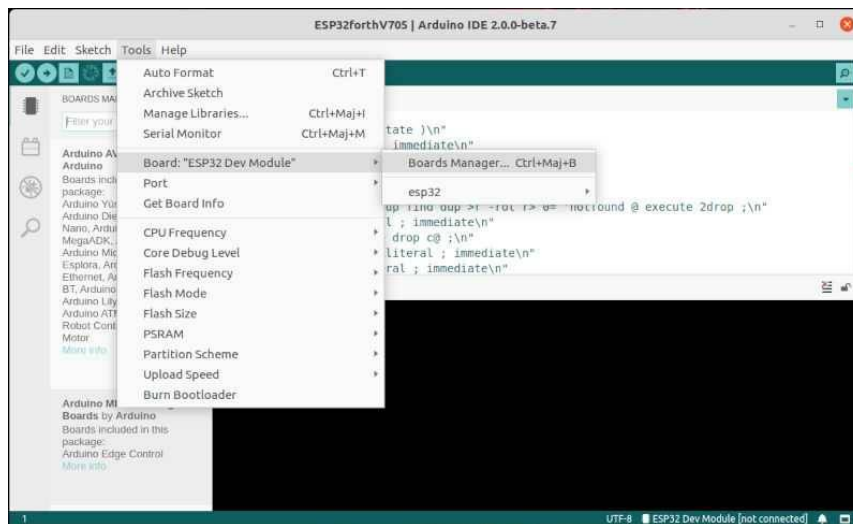


En la ventana que aparece, vaya al cuadro de entrada marcado *Additional boards manager URLs*: e ingrese esta línea:

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



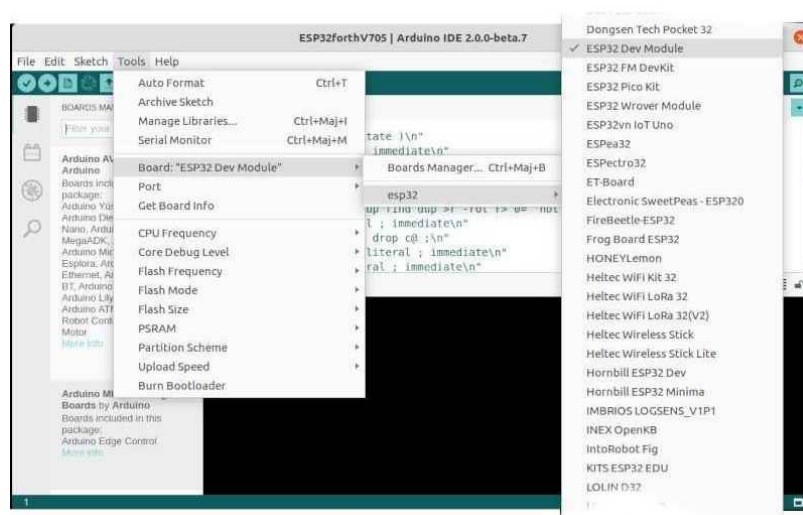
A continuación, haga clic en *Tools* y seleccione *Board* :.



Esta selección debería ofrecerle la instalación de paquetes para ESP32. Acepta esta instalación.

Entonces deberías poder acceder a la selección de tarjetas ESP32:

Selección de placa **ESP32 Dev Module** :



Configuraciones para ESP32 WROOM

Aquí están las otras configuraciones necesarias antes de compilar ESP32 en adelante. Accede a la configuración haciendo clic nuevamente en *Tools* :

```
-- TOOLS-----+-- BOARD      -----+-- ESP32  -----+-- ESP32 Dev Module
+-- Port: -----+-- COMx
|
+-- CPU Frequency -----+-- 240 Mhz
+-- Core Debug Level  -----+-- None
```

```

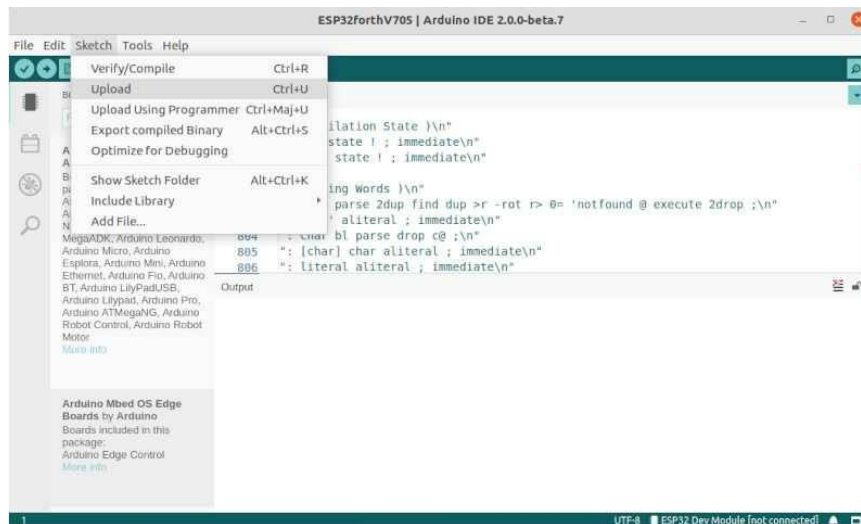
+-- Erase All Flash...-----+-- Disabled
+-- Events Run On -----+-- Core 1
+-- Flash Frequency -----+-- 80 Mhz
+-- Flash Mode -----+-- QIO
+-- Flash Size -----+-- 4MB
+-- JTAG Adapter -----+-- FTDI Adapter
+-- Arduino Runs on -----+-- Core 1
+-- PSRAM -----+-- Disabled
+-- Partition Scheme -----+-- Default 4MB with SPIFFS
+-- Upload Speed -----+-- 921600

```

Iniciar la compilación

Todo lo que queda es compilar ESP32. Cargue el código fuente mediante *File y Open*.

Se supone que su placa ESP32 está conectada a un puerto USB. Inicie la compilación haciendo clic en *Sketch* y seleccionando Upload :



Si todo va correctamente, deberías transferir el código binario automáticamente a la placa ESP32. Si la compilación se realiza sin errores, pero hay un error de transferencia, vuelva a compilar el archivo **esp32forth.ino** . En el momento de la transferencia, presione el botón marcado **BOOT** en la placa ESP32. Esto debería hacer que la tarjeta esté disponible para transferir el código binario ESP32forth.

Instalación y configuración de ARDUINO IDE en vídeo:

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

Solucionar el error de conexión de carga

Aprenda cómo solucionar el error fatal que ocurrió: "Failed to connect to ESP32: Timed out waiting for packet header" al intentar cargar un nuevo código a su tarjeta ESP32 de una vez por todas.

Algunas placas de desarrollo ESP32 (lea Las mejores placas ESP32) no ingresan al modo flash/carga automáticamente al descargar código nuevo.

Esto significa que cuando intentas cargar un nuevo boceto en tu placa ESP32, ARDUINO IDE no se conecta a tu placa y recibes el siguiente mensaje de error:

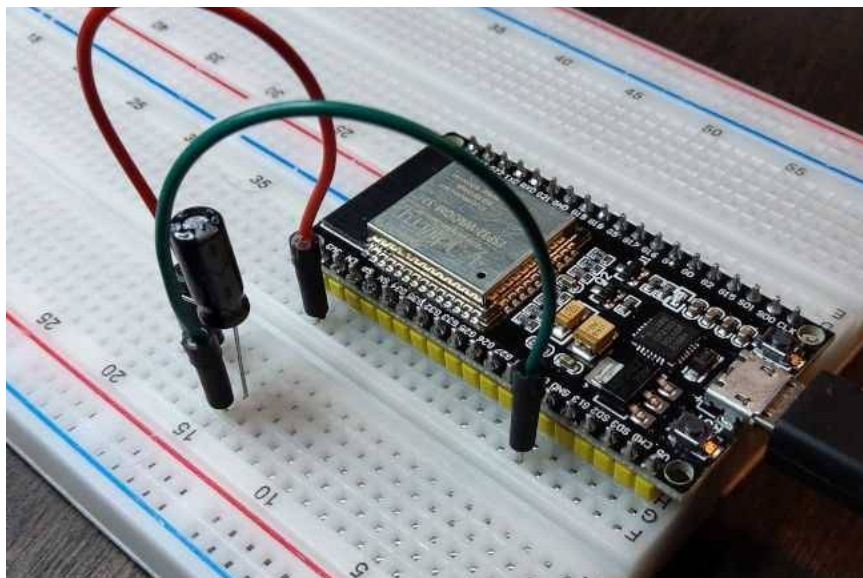


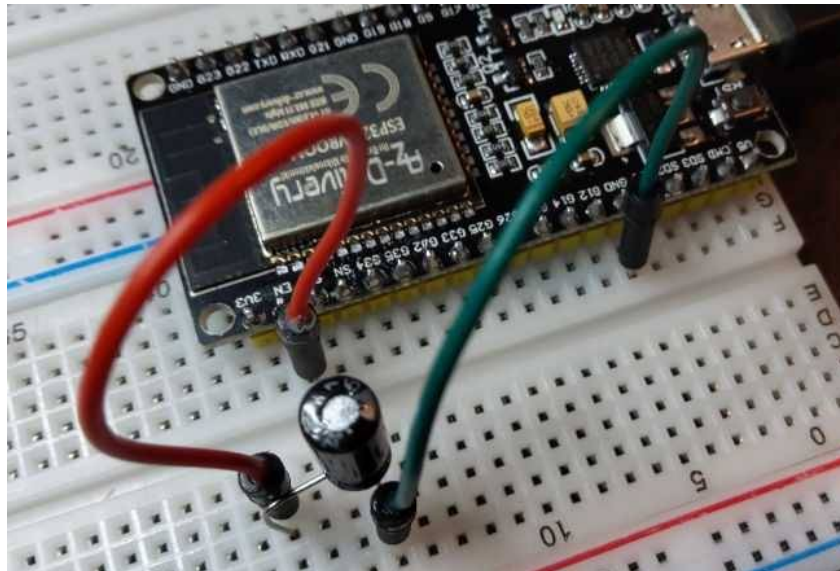
```
Blink
7 | it is attached to digital pin 13, on MKR1000 on pin 6. LED BUILTIN is set to

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

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

Para hacer que la placa ESP32 cambie automáticamente al modo flash/descarga, podemos conectar un condensador electrolítico de 10uF entre el pin EN y GND:





Esta manipulación sólo es necesaria si estás en la fase de carga de ESP32 adelante desde ARDUINO IDE. Una vez que ESP32forth está instalado en la placa ESP32, el uso de este condensador ya no es necesario.

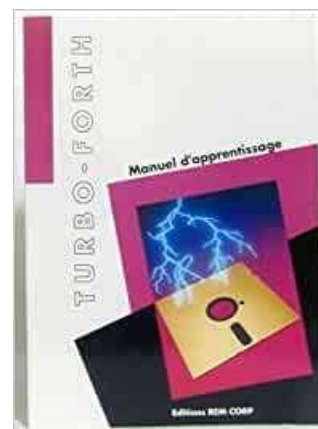
¿Por qué programar en lenguaje FORTH en ESP32?

Preámbulo

Llevo programando en FORTH desde 1983. Dejé de programar en FORTH en 1996. Pero nunca he dejado de seguir la evolución de este lenguaje. Reanudé la programación en 2019 en ARDUINO con FlashForth y luego ESP32forth.

Soy coautor de varios libros sobre el idioma FORTH:

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



Programar en el lenguaje FORTH siempre fue un hobby hasta que en 1992 me contactó el gerente de una empresa que trabajaba como subcontratista para la industria del automóvil. Tenían inquietudes por el desarrollo de software en lenguaje C. Necesitaban encargar un autómatas industrial.

Los dos diseñadores de software de esta empresa programaron en lenguaje C: TURBO-C de Borland para ser precisos. Y su código no podía ser lo suficientemente compacto y rápido como para caber en los 64 kilobytes de memoria RAM. Corría el año 1992 y no existían las ampliaciones de tipo memoria flash. ¡En estos 64 KB de RAM teníamos que meter MS-DOS 3.0 y la aplicación!

Durante un mes, los desarrolladores del lenguaje C habían estado dando vuelta al problema en todas direcciones, incluso aplicando ingeniería inversa con SOURCER (un desensamblador) para eliminar partes no esenciales del código ejecutable.

Analiqué el problema que se me presentó. Partiendo de cero, creé, solo, en una semana, un prototipo perfectamente operativo y que cumplía con las especificaciones. Durante tres años, de 1992 a 1995, creé numerosas versiones de esta aplicación que se utilizó en las líneas de montaje de varios fabricantes de automóviles.

Límites entre lenguaje y aplicación

Todos los lenguajes de programación se comparten de la siguiente manera:

- un intérprete y código fuente ejecutable: BASIC, PHP, MySQL, JavaScript, etc... La aplicación está contenida en uno o más archivos que serán interpretados cuando sea necesario. El sistema debe alojar permanentemente al intérprete que ejecuta el código fuente;
- un compilador y/o ensamblador: C, Java, etc. Algunos compiladores generan código nativo, es decir ejecutable específicamente sobre un sistema. Otros, como Java, compilan código ejecutable en una máquina Java virtual.

El lenguaje FORTH es una excepción. Integra:

- un intérprete capaz de ejecutar cualquier palabra en el CUARTO idioma
- un compilador capaz de ampliar el diccionario de CUARTAS palabras

¿Qué es una CUARTA palabra?

Una FORTH palabra designa cualquier expresión de diccionario compuesta por caracteres ASCII y utilizable en interpretación y/o compilación: palabras le permite enumerar todas las palabras en el FORTH diccionario.

Ciertas palabras FORTH solo se pueden usar en la compilación: **if else then** por ejemplo.

Con el lenguaje FORTH, el principio esencial es que no creamos una aplicación. ¡En ADELANTE, ampliamos el diccionario ! Cada nueva palabra que defina será una parte tan importante del diccionario FORTH como todas las palabras predefinidas cuando se inicie FORTH. Ejemplo :

```
: 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
;
```

Creamos dos nuevas palabras: **typeToLoRa** y **typeToTerm** que completarán el diccionario de palabras predefinidas.

¿Una palabra es una función?

Si y no. De hecho, una palabra puede ser una constante, una variable, una función... Aquí, en nuestro ejemplo, la siguiente secuencia:

```
: typeToLoRa ...código... ;
```

tendría su equivalente en lenguaje C:

```
void typeToLoRa() { ...código... }
```

En FORTH idioma, no hay límite entre el idioma y la aplicación.

En FORTH, como en el lenguaje C, puede utilizar cualquier palabra ya definida en la definición de una nueva palabra.

Sí, pero entonces ¿por qué FORTH en lugar de C?

Estaba esperando esta pregunta.

En lenguaje C, solo se puede acceder a una función a través de la función principal **main()** . Si esta función integra varias funciones adicionales, resulta difícil encontrar un error de parámetro en caso de un mal funcionamiento del programa.

Por el contrario, con FORTH es posible ejecutar - a través del intérprete - cualquier palabra predefinida o definida por usted, sin tener que pasar por la palabra principal del programa.

Se puede acceder inmediatamente al intérprete FORTH en la tarjeta ESP32 a través de un programa tipo terminal y un enlace USB entre la tarjeta ESP32 y la PC.

La compilación de programas escritos en lenguaje FORTH se realiza en la tarjeta ESP32 y no en el PC. No hay carga. Ejemplo :

```
: >gray (n -- n')
  dup 2/ xor \ n' = n xor ( 1 desplazamiento lógico a la derecha )
;
```

Esta definición se transmite copiando/pegando en el terminal. El intérprete/compilador FORTH analizará la secuencia y compilará la nueva palabra **>gray** .

En la definición de **>gray** , vemos la secuencia **dup 2/ xor** . Para probar esta secuencia, simplemente escríbala en la terminal. Para ejecutar **>gray** , simplemente escriba esta palabra en la terminal, precedida por el número a transformar.

Lenguaje FORTH comparado con el lenguaje C

Esta es la parte que menos me gusta. No me gusta comparar el lenguaje FORTH con el lenguaje C. Pero como casi todos los desarrolladores usan el lenguaje C, voy a probar el ejercicio.

Aquí hay una prueba con **if()** en lenguaje C:

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

Pruebe con **if** en el lenguaje FORTH (fragmento de código):

```

var-j @ 13 >          \ Si tous les bits sont recus
  if
    1 rc5_ok !      \ Le processus de decodage est OK
    di              \ Desactiver l'interruption externe (INT0)
    exit
  then

```

Aquí está la inicialización de registros en lenguaje C :

```

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

```

La misma definición en CUARTO idioma:

```

: 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
;

```

Qué le permite hacer FORTH en comparación con el lenguaje C

Entendemos que FORTH da acceso inmediatamente a todas las palabras del diccionario, pero no sólo eso. A través del intérprete también accedemos a toda la memoria de la tarjeta ESP32. Conéctese a la placa ARDUINO que tiene instalado FlashForth, luego simplemente escriba:

```
hex here 100 dump
```

Deberías encontrar esto en la pantalla del terminal :

```

3FFEE964          DF DF 29 27 6F 59 2B 42 FA CF 9B 84
3FFEE970 39 4E 35 F7 78 FB D2 2C A0 AD 5A AF 7C 14 E3 52
3FFEE980 77 0C 67 CE 53 DE E9 9F 9A 49 AB F7 BC 64 AE E6
3FFEE990 3A DF 1C BB FE B7 C2 73 18 A6 A5 3F A4 68 B5 69
3FFEE9A0 F9 54 68 D9 4D 7C 96 4D 66 9A 02 BF 33 46 46 45
3FFEE9B0 45 39 33 33 2F 0D 08 18 BF 95 AF 87 AC D0 C7 5D
3FFEE9C0 F2 99 B6 43 DF 19 C9 74 10 BD 8C AE 5A 7F 13 F1
3FFEE9D0 9E 00 3D 6F 7F 74 2A 2B 52 2D F4 01 2D 7D B5 1C
3FFEE9E0 4A 88 88 B5 2D BE B1 38 57 79 B2 66 11 2D A1 76
3FFEE9F0 F6 68 1F 71 37 9E C1 82 43 A6 A4 9A 57 5D CA 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 CE 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
```

Esto corresponde al contenido de la memoria flash.

¿Y el lenguaje C no podía hacer eso?

Sí, pero no tan simple e interactivo como en el lenguaje FORTH.

Veamos otro caso que destaca la extraordinaria compacidad del lenguaje FORTH...

Pero ¿por qué una pila en lugar de variables?

La pila es un mecanismo implementado en casi todos los microcontroladores y microprocesadores. Incluso el lenguaje C aprovecha una pila, pero no tienes acceso a ella.

Sólo el lenguaje FORTH brinda acceso completo a la pila de datos. Por ejemplo, para hacer una suma, apilamos dos valores, ejecutamos la suma, mostramos el resultado: **2 5 + .** muestra **7 .**

Es un poco desestabilizador, pero cuando comprendes el mecanismo de la pila de datos, aprecias enormemente su formidable eficiencia.

La pila de datos permite pasar datos entre palabras ADELANTE mucho más rápidamente que procesando variables como en el lenguaje C o cualquier otro lenguaje que use variables.

¿Estás convencido?

Personalmente, dudo que este único capítulo lo convierta irremediablemente a programar en el lenguaje FORTH. Cuando buscas dominar las placas ESP32, tienes dos opciones :

- programar en lenguaje C y utilizar las numerosas bibliotecas disponibles. Pero permanecerá encerrado en las capacidades de estas bibliotecas. Adaptar códigos al lenguaje C requiere conocimientos reales de programación en lenguaje C y dominar la arquitectura de las tarjetas ESP32. Desarrollar programas complejos siempre será un problema.
- prueba la FORTH aventura y explora un mundo nuevo y emocionante. Por supuesto, no será fácil. Necesitará comprender en profundidad la arquitectura de las tarjetas ESP32, los registros y las banderas de registro. A cambio, tendrás acceso a una programación perfectamente adaptada a tus proyectos.

¿Hay alguna solicitud profesional escrita en FORTH?

¡Oh sí! Empezando por el telescopio espacial HUBBLE, algunos de cuyos componentes fueron escritos en lenguaje FORTH.

El TGV ICE alemán (Intercit y Express) utiliza procesadores RTX2000 para controlar motores mediante semiconductores de potencia. El lenguaje de máquina del procesador RTX2000 es el lenguaje FORTH.



Este mismo procesador RTX2000 se utilizó para la sonda Philae que intentó aterrizar en un cometa.

La elección del lenguaje FORTH para aplicaciones profesionales resulta interesante si consideramos cada palabra como una caja negra. Cada palabra debe ser simple, por lo tanto tener una definición bastante corta y depender de pocos parámetros.

Durante la fase de depuración, resulta fácil probar todos los valores posibles procesados por esta palabra. Una vez convertida en perfectamente fiable, esta palabra se convierte en una caja negra, es decir, una función en la que tenemos absoluta confianza en su correcto funcionamiento. De palabra en palabra, es más fácil hacer que un programa complejo sea confiable en FORTH que en cualquier otro lenguaje de programación.

Pero si nos falta rigor, si construimos plantas de gas, también es muy fácil que una aplicación funcione mal, o incluso que falle por completo!

Finalmente, es posible, en FORTH idioma, escribir las palabras que definas en cualquier idioma humano. Sin embargo, los caracteres utilizables están limitados al conjunto de caracteres ASCII entre 33 y 127. Así es como podríamos reescribir simbólicamente las palabras alto y bajo:

```
\ Pin de puerto activo, no cambie otros.
: __/ ( pinmask portadr -- )
  mset
;
\ Deshabilite un pin de puerto, no cambie los demás.
: \__ ( pinmask portadr -- )
  mclr
;
```

A partir de este momento, para encender el LED, puedes escribir :

```
_0_ __/ \ luces LED
```

¡Sí! i La secuencia **_0_ __/** está en FORTH idioma!

Con ESP32forth, aquí están todos los caracteres a tu disposición que pueden componer una FORTH palabra:

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

Buena programación.

Usando números con ESP32Forth

Iniciamos ESP32Forth sin problema. Profundizaremos ahora en algunas manipulaciones con números para entender cómo dominar el microcontrolador en el lenguaje FORTH.

Como muchos libros, podríamos comenzar con un programa de ejemplo trivial, por ejemplo, LED parpadeantes. Como este por ejemplo:

```
\ definir LED GPIO
25 constant ledRED
26 constant ledYELLOW
27 constant ledGREEN

\ definir máscaras para LED rojo amarillo y verde
1 ledRED      defMASK: mLED_RED
1 ledYELLOW   defMASK: mLED_YELLOW
1 ledGREEN    defMASK: mLED_GREEN

\ inicialización GPIO G25 G26 y G27 en modo de salida
GPIO.init ( -- )
  1 mLED_RED      GPIO_ENABLE_REG regSet
  1 mLED_YELLOW   GPIO_ENABLE_REG regSet
  1 mLED_GREEN    GPIO_ENABLE_REG regSet
;

\ define una secuencia de ENCENDIDO y APAGADO
: GPIO.on.off.sequence { retardo de máscara de posición - }
: 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 ;
```

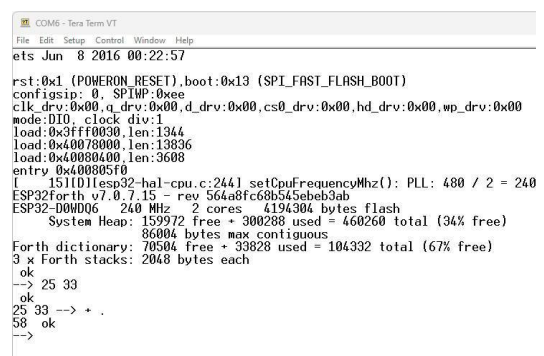
Este código, aparentemente simple, ya requiere una base de conocimientos, como la noción de dirección de memoria, registro, máscaras binarias, números hexadecimales.

Por tanto, empezaremos abordando estos conceptos básicos invitándole a realizar manipulaciones sencillas.

Números con el intérprete FORTH

Cuando se inicia ESP32Forth, la ventana del terminal TERA TERM (o cualquier otro programa de terminal de su elección) debe indicar que ESP32Forth está disponible. Presione la tecla *ENTER* en el teclado una o dos veces . ESP32Forth responde con la confirmación de que la ejecución fue exitosa . .

Vamos a probar la entrada de dos números, aquí **25** y **33**. Escriba estos números y luego *ENTER* en el teclado. ESP32Forth siempre responde con **ok**. Acaba de apilar dos números en la pila de



```
COM6 - Tera Term VT
File Edit Setup Control Window Help
ets Jun  8 2016 00:22:57

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0030,len:1344
load:0x40078000,len:13836
load:0x40080400,len:3608
entry 0x400805f0
[ 15110]!esp32-hal-cpu.c:244] setCpuFrequencyMhz(): PLL: 480 / 2 = 240
ESP32Forth v7.0.7.15 - rev 564a8fc68b545eb3ab
System Heap: 159972 free + 300288 used = 460260 total (34% free)
ESP32-D0WD06 240 MHz 2 cores 4194304 bytes flash
86004 bytes max contiguous
Forth dictionary: 70504 free + 33828 used = 104332 total (67% free)
3 x Forth stacks: 2048 bytes each
-->
ok
--> 25 33
ok
25 33 --> + .
58 ok
-->
```

idiomas ESP32Forth. Ahora ingresa + . luego presione la tecla *ENTRAR* . ESP32Forth muestra el resultado:

Esta operación fue procesada por el intérprete FORTH.

ESP32Forth, como todas las versiones del lenguaje FORTH, tiene dos estados:

- **intérprete** : el estado que acaba de probar realizando una suma simple de dos números;
- **compilador** : un estado que permite definir nuevas palabras. Este aspecto se explorará más adelante.

Ingresar números con diferentes bases numéricas

Para asimilar completamente las explicaciones, le invitamos a probar todos los ejemplos a través de la ventana del terminal TERA TERM.

Los números se pueden ingresar de forma natural. En decimal SIEMPRE será una secuencia de números, ejemplo :

```
-1234 5678 + .
```

El resultado de este ejemplo mostrará **4444**. Los CUARTOS números y palabras deben estar separados por al menos un carácter *de espacio*. El ejemplo funciona perfectamente si escribes un número o palabra por línea:

```
-1234
5678
+
.
```

Los números pueden tener un prefijo si desea ingresar valores que no sean decimales:

- signo \$ para indicar que el número es un valor hexadecimal;

Ejemplo :

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

La finalidad de estos prefijos es evitar cualquier error de interpretación en el caso de valores similares:

```
$0305
0305
```

¡No son **números iguales** si la base numérica hexadecimal no está definida explícitamente !

Cambio de base numérica

ESP32Forth tiene palabras que le permiten cambiar la base numérica:

- **hex** para seleccionar la base numérica hexadecimal;
- **binario** para seleccionar la base del número binario;
- **decimal** para seleccionar la base numérica decimal.

Cualquier número ingresado en una base numérica debe respetar la sintaxis de los números de esta base:

```
3E7F
```

causará un error si está en base decimal.

```
hex 3e7f
```

funcionará perfectamente en base hexadecimal. La nueva base numérica sigue siendo válida mientras no se seleccione otra base numérica:

```
hex
$0305
0305
```

son numeros iguales !

Una vez que un número se coloca en la pila de datos en una base numérica, su valor ya no cambia. Por ejemplo, si elimina el valor **\$ff** en la pila de datos, este valor que es **255** en decimal o **11111111** en binario no cambiará si volvemos a decimal:

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

A riesgo de insistir, **255** en decimal es **el mismo valor** que **\$ff** en hexadecimal!

En el ejemplo dado al comienzo del capítulo, definimos una constante en hexadecimal:

```
25 constant ledRED
```

Si escribimos:

```
hex ledRED .
```

Esto mostrará el contenido de esta constante en forma hexadecimal. El cambio de base **no tiene consecuencias** sobre el funcionamiento final del programa FORTH.

Binario y hexadecimal

El sistema numérico binario moderno, la base del código binario, fue inventado por Gottfried Leibniz en 1689 y aparece en su artículo Explicación de la aritmética binaria en 1703.

En su artículo, LEIBNITZ utiliza sólo los caracteres **0** y **1** para describir todos los números:

```
: 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

```

¿Es necesario entender la codificación binaria? Diré sí y no. **No** para usos cotidianos. **Sí** entender la programación de microcontroladores y dominio de operadores lógicos.

Fue Georges Boole quien describió formalmente la lógica. Su obra quedó en el olvido hasta la aparición de los primeros ordenadores. Fue Claude Shannon quien se dio cuenta de que este álgebra podía aplicarse en el diseño y análisis de circuitos eléctricos.

El álgebra booleana se ocupa exclusivamente de **0** y **1** .

Los componentes fundamentales de todas nuestras computadoras y memorias digitales utilizan codificación binaria y álgebra booleana.

La unidad de almacenamiento más pequeña es el byte. Es un espacio compuesto por 8 bits. Un bit sólo puede tener dos estados: **0** o **1** . El valor más pequeño que se puede almacenar en un byte es **00000000** , siendo el mayor **11111111** . Si cortamos un byte en dos, tendremos:

- cuatro bits de orden inferior, que pueden tomar los valores del **0000** al **1111** ;
- cuatro bits más significativos que pueden tomar uno de estos mismos valores.

Si numeramos todas las combinaciones entre 0000 y 1111, empezando por 0, llegamos a 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

```

En la parte derecha de cada línea mostramos el mismo valor que en la parte izquierda, pero en hexadecimal: ¡ **1101** y **D** son los mismos valores!

Se eligió la representación hexadecimal para representar números en informática por razones prácticas. Para la parte de orden superior o inferior de un byte, en 4 bits, las únicas combinaciones de representación hexadecimal estarán entre **0** y **F**. ¡ Aquí las letras de la A a la F **son números hexadecimales** !

```
$3E \ es más legible como 00111110
```

Por tanto, la representación hexadecimal ofrece la ventaja de representar el contenido de un byte en un formato fijo, de **00** a **FF** . En decimal, se debería haber utilizado del 0 al 255.

Tamaño de los números en la pila de datos FORTH

ESP32forth utiliza una pila de datos de 32 bits de tamaño de memoria, o 4 bytes (8 bits x 4 = 32 bits). El valor hexadecimal más pequeño que se puede apilar en la FORTH pila será **00000000** , el más grande será **FFFFFFFF** . Cualquier intento de acumular un valor mayor da como resultado el recorte de ese valor:

```

hex
abcdefabcdefabcdef . \ display: EFABCDEF

```

Apilemos el valor más grande posible en formato hexadecimal de 32 bits (4 bytes):

```

decimal
$ffffffff . \ mostrar: -1

```

Te veo sorprendido, ¡pero este resultado es **normal** ! Palabra **.** Muestra el valor que se encuentra en la parte superior de la pila de datos en su forma firmada. Para mostrar el mismo valor sin signo, debe usar la palabra **u.** :

```
$fffffffff u.    \ pantalla: 4294967295
```

Esto se debe a que los 32 bits utilizados por FORTH para representar un número entero, el bit más significativo se utiliza como signo:

- si el bit más significativo es **0** , el número es positivo;
- si el bit más significativo es **1** , el número es negativo.

Entonces, si siguió correctamente, nuestros valores decimales 1 y -1 están representados en la pila, en formato binario, de esta forma:

```
binary
00000000000000000000000000000000000001 \ empujar 1 en la pila
11111111111111111111111111111111111111 \ push -1 en la pila
```

Y aquí es donde pediremos a nuestro matemático, el señor LEIBNITZ, que sume estos dos números en binario. Si en el colegio nos gusta, empezando por la derecha, simplemente habrá que respetar esta regla: $1 + 1 = 10$ en binario. Ponemos los resultados en una tercera línea:

```
0000000000000000000000000000000000001  
1111111111111111111111111111111111111  
10
```

Etapa siguiente :

[illegible]

Al final tendremos el resultado:

[illegible]

Pero como este resultado tiene el bit 33 más significativo en 1, sabiendo que el formato de entero está estrictamente limitado a 32 bits, el resultado final es 0. Es sorprendente? Sin embargo, esto es lo que hace todo reloj digital. Ocultar las horas. Cuando llegue a 59, agregue 1, el reloj mostrará 0.

¡Las reglas de la aritmética decimal, es decir $-1 + 1 = 0$, se han respetado perfectamente en la lógica binaria!

Acceso a memoria y operaciones lógicas.

La pila de datos no es en ningún caso un espacio de almacenamiento de datos. Su tamaño también es muy limitado. Y la pila la comparten muchas palabras. El orden de los parámetros es fundamental. Un error puede provocar fallos de funcionamiento. Tomemos el caso de la palabra **dump** que muestra el contenido de un espacio de memoria:

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

En **negrita y rojo** encontramos los cuatro bytes reservados para almacenar un valor en nuestra variable puntuación. Almacenemos cualquier valor en **score** :

```
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
```

Encontramos los cuatro bytes que contienen nuestro valor decimal **1900** , **0000076C** en hexadecimal. ¿Aún estás sorprendido? Así que la causa es el efecto de la codificación binaria y sus sutilezas. En la memoria los bytes se almacenan empezando por los menos significativos. Tras la recuperación, el mecanismo de transformación es transparente:

```
score @ . \ mostrar 1900
```

Volvamos al código que hace parpadear un LED. Extracto :

```
1 mLED_RED      GPIO_ENABLE_REG reqSet
```

Este código activa una salida GPIO asociada a un LED. La palabra **GPIO_ENABLE_REG** es una constante, cuyo contenido es una máscara que apunta a este LED. Bien podríamos haber escrito esto:

```
1 25 lshift GPIO_ENABLE_REG !
```

Aquí, la palabra **lshift** realiza un desplazamiento lógico de 25 bits hacia la izquierda:

```
\ before shift: %0000000000000000000000000000000000001  
\ after shift: %000000100000000000000000000000000000000
```

Como recordatorio, los GPIO ²están numerados del 0 al 31. Para activar otro GPIO, por ejemplo GPIO17, habríamos ejecutado esto:

```
1 17 lshift GPIO_ENABLE_REG !
```

2 Entrada/Salida de propósito general = Entrada-salida de propósito general

Supongamos que queremos activar los GPIO 17 y 25 en un solo comando, ejecutaremos esto:

```
1 25 lshift
1 17 lshift or GPIO_ENABLE_REG !
```

¿Qué hemos hecho? Aquí el detalle de las operaciones:

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

La palabra **or** ha realizado una operación que combina los dos desplazamientos en una única máscara binaria.

Volvamos a nuestra variable **de score** . Queremos aislar el byte menos significativo. Disponemos de varias soluciones. Una solución utiliza enmascaramiento binario con el operador lógico **and** :

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

Para aislar el segundo byte de la derecha:

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

Aquí nos divertimos con el contenido de una variable. Para dominar un microcontrolador como el montado en la tarjeta ESP32, los mecanismos no son muy diferentes. La parte más difícil es encontrar los registros correctos. Éste será el tema de otro capítulo.

Para concluir este capítulo, todavía queda mucho que aprender sobre la lógica binaria y las diferentes codificaciones digitales posibles. Si ha probado los pocos ejemplos que se dan aquí, seguramente comprenderá que FORTH es un lenguaje interesante:

- gracias a su intérprete que permite realizar numerosas pruebas de forma interactiva sin necesidad de recompilar cargando código;
- un diccionario a cuya mayoría de palabras el intérprete puede acceder;
- un compilador que le permite agregar nuevas palabras *sobre la marcha* y luego probarlas inmediatamente.

Finalmente, lo que no estropea nada, el código FORTH, una vez compilado, es ciertamente tan eficiente como su equivalente en lenguaje C.

Un verdadero FORTH de 32 bits con ESP32Forth

ESP32Forth es un FORTH real de 32 bits. Qué significa eso ?

El lenguaje FORTH favorece la manipulación de valores enteros. Estos valores pueden ser valores literales, direcciones de memoria, contenidos de registros, etc.

Valores en la pila de datos

Cuando se inicia ESP32Forth, el intérprete FORTH está disponible. Si ingresa cualquier número, se colocará en la pila como un entero de 32 bits:

```
35
```

Si apilamos otro valor, también se apilará. El valor anterior será empujado hacia abajo una posición:

```
45
```

Para sumar estos dos valores, usamos una palabra, aquí **+** :

```
+
```

Nuestros dos valores enteros de 32 bits se suman y el resultado se coloca en la pila. Para mostrar este resultado, usaremos la palabra **.** :

```
. \ mostrar 80
```

En el lenguaje FORTH podemos concentrar todas estas operaciones en una sola línea:

```
35 45 +. \ mostrar 80
```

A diferencia del lenguaje C, no definimos un tipo **int8** , **int16** o **int32** .

Con ESP32Forth, un carácter ASCII será designado por un entero de 32 bits, pero cuyo valor estará acotado [32..256[. Ejemplo :

```
67 emit \ mostrar C
```

Valores en la memoria

ESP32Forth le permite definir constantes y variables. Su contenido siempre estará en formato de 32 bits. Pero hay situaciones en las que eso no necesariamente nos conviene. Tomemos un ejemplo sencillo: definamos un alfabeto en código Morse. Sólo necesitamos unos pocos bytes:

- uno para definir el número de signos del código morse

- uno o más bytes por cada letra del código Morse

```
create mA ( -- addr )
  2 c,
  char . c,   char - c,

create mB ( -- addr )
  4 c,
  char - c,   char . c,   char . c,   char . c,

create mC ( -- addr )
  4 c,
  char - c,   char . c,   char - c,   char . c,
```

Aquí definimos solo 3 palabras, **mA** , **mB** y **mC** . En cada palabra se almacenan varios bytes. La pregunta es: ¿cómo recuperaremos la información contenida en estas palabras?

La ejecución de una de estas palabras deposita un valor de 32 bits, valor que corresponde a la dirección de memoria donde almacenamos nuestra información en código Morse. Es la palabra **c@** la que usaremos para extraer el código Morse de cada letra:

```
mA c@ . \ muestra 2
mB c@ . \ muestra 4
```

El primer byte extraído así se utilizará para gestionar un bucle para mostrar el código Morse de una letra:

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

Hay muchos ejemplos ciertamente más elegantes. Aquí, es para mostrar una forma de manipular valores de 8 bits, nuestros bytes, mientras usamos estos bytes en una pila de 32 bits.

Procesamiento de textos dependiendo del tamaño o tipo de datos.

En todos los demás idiomas tenemos una palabra genérica, como **echo** (en PHP) que muestra cualquier tipo de datos. Ya sea un número entero, real o una cadena, siempre usamos la misma palabra. Ejemplo en lenguaje PHP:

```
$bread = "Pain cuit";
```



```
$price = 2.30;
echo $bread . " : " . $price;
// affiche    Pain cuit: 2.30
```

¡Para todos los programadores, esta forma de hacer las cosas es EL ESTÁNDAR! Entonces, ¿cómo haría FORTH este ejemplo en PHP?

```
: pain s" Pain cuit" ;
: prix s" 2.30" ;
pain type    s" : " type    prix type
\ affiche    Pain cuit: 2.30
```

Aquí, el **tipo de palabra** nos dice que acabamos de procesar una cadena de caracteres.

Cuando PHP (o cualquier otro lenguaje) tiene una función genérica y un analizador, FORTH lo compensa con un único tipo de datos, pero con métodos de procesamiento adaptados que nos informan sobre la naturaleza de los datos procesados.

Aquí hay un caso absolutamente trivial para FORTH, que muestra una cantidad de segundos en formato HH:MM:SS:

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

Me encanta este ejemplo porque, hasta la fecha, **NINGÚN OTRO LENGUAJE DE PROGRAMACIÓN** es capaz de realizar esta conversión HH:MM:SS de forma tan elegante y concisa.

Lo has entendido, el secreto de FORTH está en su vocabulario.

Conclusión

FORTH no tiene tipificación de datos. Todos los datos pasan a través de una pila de datos. ¡Cada posición en la pila es SIEMPRE un entero de 32 bits!

Eso es todo lo que hay que saber.

Los puristas de los lenguajes hiperestructurados y prolijos, como C o Java, ciertamente gritarán herejía. Y aquí me permitiré responderlas: ¿por qué necesitas escribir tus datos?

Porque es en esa simplicidad donde reside el poder de FORTH: una única pila de datos con un formato sin tipo y operaciones muy sencillas.

Y les voy a mostrar lo que muchos otros lenguajes de programación no pueden hacer:
definir nuevas palabras de definición:

```
: 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    .- -... -.-.
```

Aquí, la palabra **morse:** se ha convertido en una palabra de definición, del mismo modo que **constante** o **variable** ...

Porque FORTH es más que un lenguaje de programación. Es un metalenguaje, es decir un lenguaje para construir tu propio lenguaje de programación....

Comentarios y aclaraciones

No existe un IDE³ para gestionar y presentar código escrito en lenguaje FORTH de forma estructurada. En el peor de los casos, utiliza un editor de texto ASCII y, en el mejor de los casos, un IDE real y archivos de texto:

- **edit** o **wordpad** en Windows
- **edit** en Linux
- **PsPad** bajo Windows
- **Netbeans** bajo Windows o Linux...

Aquí hay un fragmento de código que podría escribir un principiante:

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

Este código será perfectamente compilado por ESP32 en adelante. Pero, ¿seguirá siendo comprensible en el futuro si es necesario modificarlo o reutilizarlo en otra aplicación?

Escribir código ADELANTE legible

Comencemos con el nombre de la palabra a definir, aquí **cycle.stop**. ESP32forth te permite escribir nombres de palabras muy largos. El tamaño de las palabras definidas no influye en el rendimiento de la aplicación final. Por tanto, tenemos cierta libertad para escribir estas palabras:

- como programación de objetos en JavaScript: **cycle.stop**
- el Camino del CamelloCiclo **de cycleStop**
- **cycle-stop-lights** un código muy comprensible
- el código **cs1** conciso

No hay ninguna regla. Lo principal es que puedes volver a leer fácilmente tu código FORTH. Sin embargo, los programadores informáticos en lenguaje FORTH tienen ciertos hábitos:

- constantes en caracteres mayúsculos **MAX_LIGHT_TIME_NORMAL_CYCLE**
- palabra que define otras palabras **defPin:** , es decir, palabra seguida de dos puntos;
- palabra de transformación de dirección **>date**, aquí el parámetro de dirección se incrementa en un valor determinado para apuntar a los datos apropiados;

3 Entorno de desarrollo integrado = Entorno de desarrollo integrado

- almacenamiento de memoria palabra **date@** o **date!**
- Palabra de visualización de datos **.fecha**

¿Y qué hay de nombrar palabras FORTH en un idioma que no sea el inglés? Una vez más, sólo hay una regla: ¡ **libertad total** ! Sin embargo, tenga cuidado, ESP32forth no acepta nombres escritos en alfabetos distintos del alfabeto latino. Sin embargo, puedes utilizar estos alfabetos para comentarios:

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

O

```
: ..date \海报今天的日期
...codificado... ;
```

Sangría del código fuente

Si el código tiene dos líneas, diez líneas o más no tiene ningún efecto en el rendimiento del código una vez compilado. Por lo tanto, también puedes sangrar tu código de forma estructurada:

- una línea por palabra de la estructura de control **if else then** , **begin while repeat...** Para la palabra if, podemos precederla con la prueba lógica que procesará;
- una línea mediante la ejecución de una palabra predefinida, precedida si es necesario por los parámetros de esta palabra.

Ejemplo :

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

Si el código procesado en una estructura de control es escaso, el código FORTH se puede compactar:

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

```
;
```

Este suele ser el caso de las estructuras **case of endof endcase** ;

```
: 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
;
```

Los comentarios

Como cualquier lenguaje de programación, el lenguaje FORTH permite agregar comentarios en el código fuente. Agregar comentarios no tiene ningún impacto en el rendimiento de la aplicación después de compilar el código fuente.

En FORTH idioma, tenemos dos palabras para delimitar los comentarios:

- la palabra **(** debe ir seguida de al menos un carácter de espacio. Este comentario se completa con el carácter **)** ;
- la palabra **** debe ir seguida de al menos un carácter de espacio. Esta palabra va seguida de un comentario de cualquier tamaño entre esta palabra y el final de la línea.

La palabra **(** se usa ampliamente para comentarios de pila. Ejemplos:

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

Comentarios de pila

Como acabamos de ver, están marcados con **(** y **)** . Su contenido no tiene ningún efecto en el código FORTH durante la compilación o ejecución. Entonces podemos poner cualquier cosa entre **(** y **)** . En cuanto a los comentarios de la pila, seremos muy concisos. El signo **--** simboliza la acción de una CUARTA palabra. Las indicaciones antes de **--** corresponden a los datos colocados en la pila de datos antes de la ejecución de la palabra. Las indicaciones después de **--** corresponden a los datos que quedan en la pila de datos después de la ejecución de la palabra. Ejemplos:

- **words (--)** significa que esta palabra no procesa ningún dato en la pila de datos;
- **emit (c --)** significa que esta palabra procesa datos como entrada y no deja nada en la pila de datos;
- **bl (-- 32)** significa que esta palabra no procesa ningún dato de entrada y deja el valor decimal 32 en la pila de datos;

No existe limitación en la cantidad de datos procesados antes o después de la ejecución de la palabra. Le recordamos que las indicaciones entre (y) son sólo informativas.

Significado de los parámetros de la pila en los comentarios.

Para empezar es necesaria una pequeña pero muy importante aclaración. Este es el tamaño de los datos en la pila. Con ESP32Forth, los datos de la pila ocupan 4 bytes. Entonces estos son números enteros en formato de 32 bits. Sin embargo, algunas palabras procesan datos en formato de 8 bits. Entonces, ¿qué ponemos en la pila de datos? Con ESP32Forth, ¡ **SIEMPRE serán DATOS DE 32 BITS** ! ¡Un ejemplo con la palabra **c!** :

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

Aquí el parámetro **c** indica que apilamos un valor entero en formato de 32 bits, pero cuyo valor siempre estará incluido en el intervalo [0..255].

El parámetro estándar es siempre **n** . Si son varios números enteros los numeraremos: **n1 n2 n3** , etc.

Por tanto, podríamos haber escrito el ejemplo anterior así:

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

Pero es mucho menos explícita que la versión anterior. Aquí hay algunos símbolos que verá en los códigos fuente:

- **addr** indica una dirección de memoria literal o entregada por una variable;
- **c** indica un valor de 8 bits en el intervalo [0..255]
- **d** indica un valor de precisión doble.
No se utiliza con ESP32Forth que ya está en formato de 32 bits;
- **fl** indica un valor booleano, 0 o distinto de cero;
- **n** indica un número entero. Entero con signo de 32 bits para ESP32Forth;

- **str** indica una cadena de caracteres. Equivalente a **addr len --**
- **u** indica un entero sin signo

Nada nos impide ser un poco más explícitos:

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

Definición de palabras Comentarios de palabras

Las palabras de definición usan **create** y **does>**. Para estas palabras, es recomendable escribir comentarios de pila como este :

```
\ definir un comando o flujo de datos para SSD1306
: streamCreate: ( comp: <name> | exec: -- addr len )
  create
    here      \ dejar el puntero del diccionario actual en la pila
    0 c,      \ datos de longitud inicial es 0
  does>
    dup 1+ swap c@
    \ enviar matriz de datos a SSD1306 conectado a través bus
I2C
    sendDataToSSD1306
;
```

Aquí, el comentario está dividido en dos partes por el carácter **|** :

- a la izquierda, la parte de acción cuando se ejecuta la palabra de definición, con el prefijo **comp:**
- a la derecha la parte de acción de la palabra que se definirá, con el prefijo **exec:**

A riesgo de insistir, esto no es un estándar. Estas son sólo recomendaciones.

Comentarios textuales

Se indican con la palabra **** seguida de al menos un espacio y un texto explicativo:

```
\ 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!
;
```

Estos comentarios se pueden escribir en cualquier alfabeto admitido por su editor de código fuente:

```

\ 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!
;

```

Comentario al principio del código fuente.

Con una práctica intensiva de programación, rápidamente se encontrará con cientos o incluso miles de archivos fuente. Para evitar errores en la elección de archivos, se recomienda marcar el inicio de cada archivo fuente con un comentario:

```

\ *****
\ Administrar comandos para pantalla OLED SSD1306 128x32
\ Nombre de archivo: SSD10306commands.fs
\ 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
\ *****

```

Toda esta información queda a tu discreción. Pueden resultar muy útiles cuando vuelve al contenido de un archivo meses o años después.

Para concluir, no dude en comentar y sangrar sus archivos fuente en el idioma ADELANTE.

Herramientas de diagnóstico y ajuste.

La primera herramienta se refiere a la alerta de compilación o interpretación:

```

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

```

Aquí la palabra **DDUP** no existe. Cualquier compilación posterior a este error fallará.

El descompilador

En un compilador convencional, el código fuente se transforma en código ejecutable que contiene las direcciones de referencia a una biblioteca que equipa el compilador. Para tener código ejecutable, debes vincular el código objeto. En ningún momento el

programador puede tener acceso al código ejecutable contenido en su biblioteca únicamente con los recursos del compilador.

Con ESP32Forth, el desarrollador puede descompilar sus definiciones. Para descompilar una palabra, simplemente escriba **ver** seguido de la palabra a descompilar:

```
: C>F ( 0C --- 0F) \ Conversión de Celsius a Fahrenheit
  9 5 */ 32 +
;
see c>f
\ mostrar:
: C>F
  9 5 */ 32 +
;
```

Muchas palabras del diccionario FORTH de ESP32Forth se pueden descompilar.

Descompilar tus palabras te permite detectar posibles errores de compilación.

Volcado de memoria

A veces es deseable poder ver los valores que hay en la memoria. La palabra **dump** acepta dos parámetros: la dirección inicial en la memoria y el número de bytes a mostrar:

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

Monitor de pila

El contenido de la pila de datos se puede mostrar en cualquier momento usando la palabra **.s**. Aquí está la definición de la palabra **.DEBUG** que explota **.s**:

```
variable debugStack

: debugOn ( -- )
  -1 debugStack !
;

: debugOff ( -- )
  0 debugStack !
;

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

```
then  
;
```

Para utilizar `.DEBUG`, simplemente insértelo en un lugar estratégico de la palabra a depurar:

```
\ ejemplo de uso:  
: myTEST  
  128 32 do  
    i .DEBUG  
    emit  
  loop  
;
```

Aquí, mostraremos el contenido de la pila de datos después de la ejecución de la palabra `i` en nuestro bucle **do loop**. Activamos el foco y ejecutamos **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
```

Cuando la depuración está habilitada por **debugOn** , cada visualización del contenido de la pila de datos detiene nuestro ciclo **ciclo do loop**. Ejecute **debugOff** para que la palabra **myTEST** se ejecute normalmente.

Diccionario / Pila / Variables / Constantes

Ampliar diccionario

Forth pertenece a la clase de lenguajes interpretativos tejidos. Esto significa que puede interpretar comandos escritos en la consola, así como compilar nuevas subrutinas y programas.

El compilador Forth es parte del lenguaje y se utilizan palabras especiales para crear nuevas entradas de diccionario (es decir, palabras). Los más importantes son **:** (iniciar una nueva definición) y **;** (termina la definición). Probemos esto escribiendo:

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

¿Lo que pasó? La acción de: es crear una nueva entrada de diccionario llamada ***+** y cambiar del modo de interpretación al modo de compilación. En modo de compilación, el intérprete busca palabras y, en lugar de ejecutarlas, instala punteros a su código. Si el texto es un número, en lugar de colocarlo en la pila, ESP32forth construye el número en el espacio del diccionario asignado para la nueva palabra, siguiendo un código especial que coloca el número almacenado en la pila cada vez que se ejecuta la palabra. La acción de ejecución de ***+** es por tanto ejecutar secuencialmente las palabras previamente definidas ***** y **+**.

La palabra **;** es especial. Es una palabra inmediata y siempre se ejecuta, incluso si el sistema está en modo compilación. ¿Qué hace **?** es doble. Primero, instala código que devuelve el control al siguiente nivel externo del intérprete y, segundo, regresa del modo de compilación al modo de interpretación.

Ahora prueba tu nueva palabra:

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

Este ejemplo ilustra dos actividades de trabajo principales en Forth: agregar una nueva palabra al diccionario y probarla tan pronto como se haya definido.

Gestión de diccionarios

La palabra **forget** seguida de la palabra a eliminar eliminará todas las entradas del diccionario que haya realizado desde esa palabra:

```
: test1 ;  
: test2 ;  
: test3 ;  
forget test2 \ borrar test2 y test3 del diccionario
```

Pilas y notación polaca inversa

Forth tiene una pila explícitamente visible que se utiliza para pasar números entre palabras (comandos). Usar Forth efectivamente te obliga a pensar en términos de la pila. Esto puede resultar difícil al principio, pero como todo, se vuelve mucho más fácil con la práctica.

En FORTH, la pila es análoga a una pila de cartas con números escritos en ellas. Los números siempre se agregan en la parte superior de la pila y se eliminan de la parte superior de la pila. ESP32forth integra dos pilas: la pila de parámetros y la pila de retroalimentación, cada una de las cuales consta de una cantidad de celdas que pueden contener números de 16 bits.

La FORTH línea de entrada:

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

deja la pila de parámetros como está

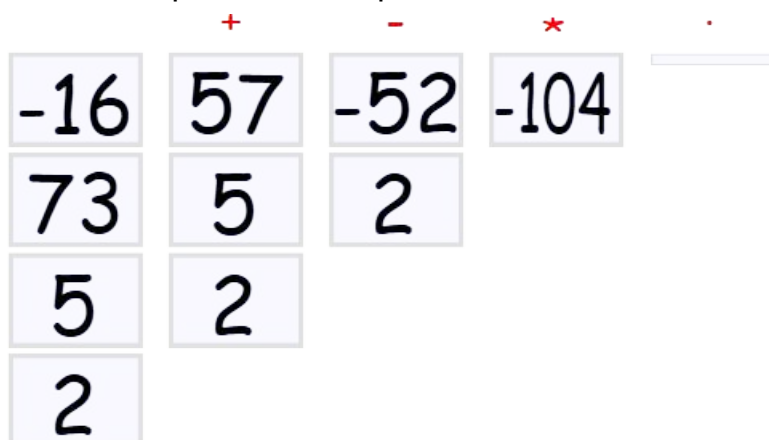
Célula	contenido	comentario
0	-dieciséis	(TOS) Arriba a la derecha
1	73	(NOS) El siguiente en la pila
2	5	
3	2	

Normalmente usaremos numeración relativa de base cero en estructuras de datos Forth, como pilas, matrices y tablas. Tenga en cuenta que cuando se ingresa una secuencia de números de esta manera, el número más a la derecha se convierte en *TOS* y el número más a la izquierda está en la parte inferior de la pila.

Supongamos que seguimos la línea de entrada original con la línea

```
+ - * .
```

Las operaciones producirían operaciones de pila sucesivas:



Después de las dos líneas, la consola muestra:

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

Tenga en cuenta que ESP32 en adelante muestra convenientemente los elementos de la pila al interpretar cada línea y que el valor de -16 se muestra como un entero sin signo de 32 bits. Además, la palabra `.` consume el valor de datos -104, dejando la pila vacía. Si ejecutamos. en la pila ahora vacía, el intérprete externo aborta con un error de puntero de pila STACK UNDERFLOW ERROR.

La notación de programación donde los operandos aparecen primero, seguidos por los operadores se llama notación polaca inversa (RPN).

Manejo de la pila de parámetros

Al ser un sistema basado en pilas, ESP32forth debe proporcionar formas de poner números en la pila, eliminarlos y reorganizar su orden. Ya hemos visto que podemos poner números en la pila simplemente escribiéndolos. También podemos integrar números en la definición de una CUARTA palabra.

La palabra **drop** elimina un número de la parte superior de la pila y coloca así el siguiente en la parte superior. La palabra **swap** intercambia los 2 primeros números. **dup** copia el número en la parte superior, empujando todos los demás números hacia abajo. **rot** rota los primeros 3 números. Estas acciones se presentan a continuación.

	drop	swap	rot	dup
-16	73	5	2	2
73	5	73	5	2
5	2	2	73	5
2				73

La pila de retorno y sus usos

Al compilar una nueva palabra, ESP32forth establece vínculos entre la palabra que llama y las palabras previamente definidas que serán invocadas por la ejecución de la nueva palabra. Este mecanismo de vinculación, en tiempo de ejecución, utiliza rstack. La dirección de la siguiente palabra que se invocará se coloca en la pila trasera para que cuando la palabra actual haya terminado de ejecutarse, el sistema sepa dónde pasar a la siguiente palabra. Dado que las palabras se pueden anidar, debe haber una pila de estas direcciones de retorno.

Además de servir como reserva de direcciones de retorno, el usuario también puede almacenar y recuperar de la pila de retorno, pero esto debe hacerse con cuidado porque la pila de retorno es esencial para la ejecución del programa. Si utiliza la batería de retorno para almacenamiento temporal, debe devolverla a su estado original; de lo contrario, es probable que bloquee el sistema ESP32forth. A pesar del peligro, hay ocasiones en las que usar backstack como almacenamiento temporal puede hacer que su código sea menos complejo.

Para almacenar en la pila, use **>r** para mover la parte superior de la pila de parámetros a la parte superior de la pila de retorno. Para recuperar un valor, **r>** mueve el valor superior de la pila nuevamente a la parte superior de la pila de parámetros. Para simplemente eliminar un valor de la parte superior de la pila, existe la palabra **rdrop**. La palabra **r@** copia la parte superior de la pila nuevamente en la pila de parámetros.

Uso de memoria

En ESP32 en adelante, los números de 32 bits se recuperan de la memoria a la pila mediante la palabra **@** (fetch) y se almacenan desde arriba en la memoria mediante la palabra **.** (ciego). **@** espera una dirección en la pila y reemplaza la dirección con su contenido. **!** espera un número y una dirección para almacenarlo. Coloca el número en la ubicación de memoria a la que hace referencia la dirección, consumiendo ambos parámetros en el proceso.

Los números sin signo que representan valores de 8 bits (bytes) se pueden colocar en caracteres del tamaño de un carácter. celdas de memoria usando **c@** y **c!**.

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

Variables

Una variable es una ubicación con nombre en la memoria que puede almacenar un número, como el resultado intermedio de un cálculo, fuera de la pila. Por ejemplo:

```
variable x
```

crea una ubicación de almacenamiento llamada **x**, que se ejecuta dejando la dirección de su ubicación de almacenamiento en la parte superior de la pila:

```
x . \ muestra la dirección
```

Luego podremos recoger o almacenar en esta dirección:

```
variable x
```

```
3 x !  
x @ .      \ muestra: 3
```

Constantes

Una constante es un número que no desea cambiar mientras se ejecuta un programa. El resultado de ejecutar la palabra asociada a una constante es el valor de los datos que quedan en la pila.

```
\ define los pines VSPI  
19 constant VSPI_MISO  
23 constant VSPI_MOSI  
18 constant VSPI_SCLK  
05 constant VSPI_CS  
  
\ establece la frecuencia del puerto SPI  
4000000 constant SPI_FREQ  
  
\ seleccionar vocabulario SPI  
only FORTH SPI also  
  
\ inicializa el puerto SPI  
: init.VSPI ( -- )  
  VSPI_CS OUTPUT pinMode  
  VSPI_SCLK VSPI_MISO VSPI_MOSI VSPI_CS SPI.begin  
  SPI_FREQ SPI.setFrequency  
;
```

Valores pseudoconstantes

Un valor definido con valor es un tipo híbrido de variable y constante. Establecemos e inicializamos un valor y se invoca como lo haríamos con una constante. También podemos cambiar un valor como podemos cambiar una variable.

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

La palabra **to** también funciona en definiciones de palabras, reemplazando el valor que le sigue con lo que esté actualmente en la parte superior de la pila. Debe tener cuidado de que **a** vaya seguido de un valor definido por **value** y no de otra cosa.

Herramientas básicas para la asignación de memoria.

Las palabras **create** y **allot** son las herramientas básicas para reservar espacio en la memoria y colocarle una etiqueta. Por ejemplo, la siguiente transcripción muestra una nueva entrada del diccionario de **graphic-array** :

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

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

Cuando se ejecuta, la palabra **graphic-array** insertará la dirección de la primera entrada.

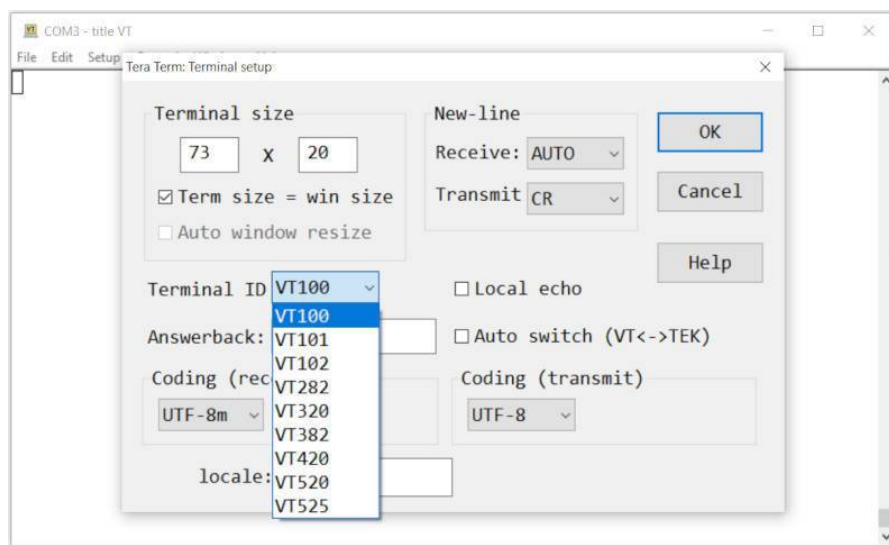
Ahora podemos acceder a la memoria asignada a la **graphic-array** usando las palabras de búsqueda y almacenamiento explicadas anteriormente. Para calcular la dirección del tercer byte asignado a **graphic-array** podemos escribir **graphic-array 2 +** , recordando que los índices comienzan en 0.

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


Colores de texto y posición de visualización en el terminal

Codificación ANSI de terminales.

Si utiliza software de terminal para comunicarse con ESP32 en adelante, es muy probable que este terminal emule un terminal tipo VT o equivalente. Aquí, TeraTerm configurado para emular un terminal VT100:



Estos terminales cuentan con dos características interesantes:

- colorear el fondo de la página y el texto que se mostrará
- posicionar el cursor de visualización

Ambas funciones están controladas por secuencias ESC (escape). Así es como se definen las palabras **bg** y **fg** en ESP32 en adelante:

```
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" ;
```

Palabra **normal** anula las secuencias de color definidas por **bg** y **fg** .

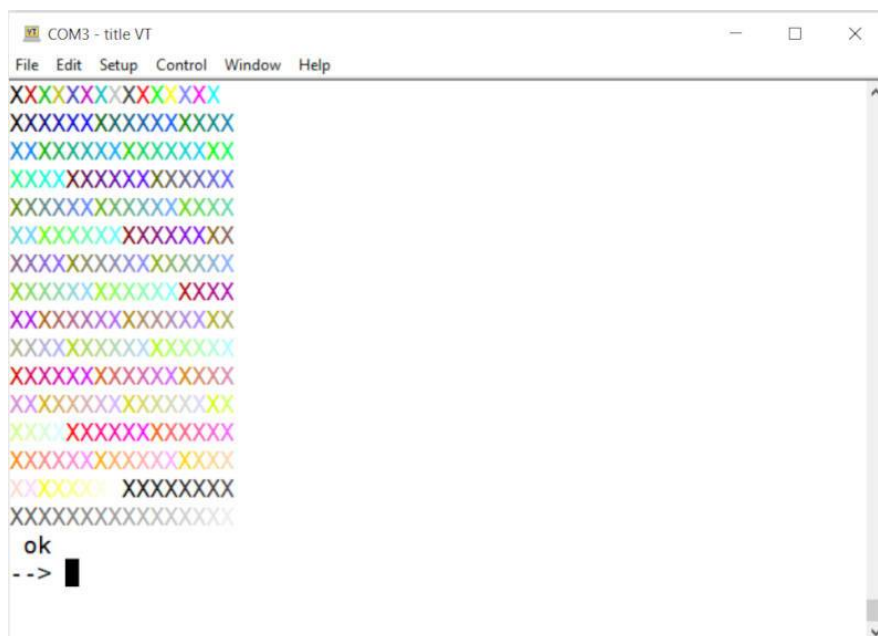
La palabra **page** borra la pantalla del terminal y coloca el cursor en la esquina superior izquierda de la pantalla.

Coloración de texto

Veamos primero cómo colorear el texto:

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

Al ejecutar **testFG** se muestra esto:



Para probar los colores de fondo, procederemos de la siguiente manera:

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

Al ejecutar **testBG** se muestra esto:



Posición de visualización

El terminal es la solución más sencilla para comunicarse con ESP32forth. Con secuencias de escape ANSI es fácil mejorar la presentación de datos.

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

: 3boxes ( -- )
  page
  2 4 20 6 cyan box
  8 6 28 8 red box
  14 8 36 10 yellow box
  0 0 at-xy
;
```

Ejecutar **3boxes** muestra esto:



Ahora está equipado para crear interfaces simples y efectivas que permitan la interacción con las definiciones FORTH compiladas por ESP32forth.

Variables locales con ESP32Forth

Introducción

El lenguaje FORTH procesa datos principalmente a través de la pila de datos. Este mecanismo muy simple ofrece un rendimiento inigualable. Por el contrario, seguir el flujo de datos puede volverse complejo rápidamente. Las variables locales ofrecen una alternativa interesante.

El comentario de la pila falsa

Si sigues los diferentes ejemplos FORTH, habrás notado los comentarios de la pila enmarcados por `(y)` . Ejemplo:

```
\ suma dos valores sin signo, deja la suma y la lleva a la pila
: um+ ( u1 u2 -- sum carry )
\ aquí la definición
;
```

Aquí, el comentario `(u1 u2 - sum carry)` no tiene absolutamente ninguna acción sobre el resto del código FORTH. Esto es puro comentario.

Al preparar una definición compleja, la solución es utilizar variables locales enmarcadas por `{ y }` . Ejemplo:

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

Definimos cuatro variables locales `a b c y d` .

Las palabras `{ y }` se parecen a las palabras `(y)` pero no tienen el mismo efecto en absoluto. Los códigos colocados entre `{ y }` son variables locales. La única restricción: no utilice nombres de variables que puedan ser palabras FORTH del diccionario FORTH. También podríamos haber escrito nuestro ejemplo así:

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

Cada variable tomará el valor de la pila de datos en el orden de su depósito en la pila de datos. aquí, 1 entra en `varA` , 2 en `varB` , etc.:

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

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

Nuestro comentario de pila falsa se puede completar así:

```
: 20VER { varA varB varC varD -- varA varB varC varD varA varB }
.....
```

Los caracteres siguientes `--` tienen ningún efecto. El único punto es hacer que nuestro comentario falso parezca un comentario de pila real.

Acción sobre variables locales.

Las variables locales actúan exactamente como pseudovariables definidas por valor.

Ejemplo:

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

Tiene el mismo efecto que este:

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

En este ejemplo, `var` se define explícitamente por valor.

Asignamos un valor a una variable local con la palabra **to** o **+to** para incrementar el contenido de una variable local. En este ejemplo, agregamos una variable local **result** inicializada a cero en el código de nuestra palabra:

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

¿No es más legible que esto?

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

Aquí hay un ejemplo final, la definición de la palabra **um+** que suma dos enteros sin signo y deja la suma y el valor de desbordamiento de esta suma en la pila de datos:

```
\ suma dos enteros sin signo, deja la suma y la lleva a la pila
: 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
;
```

Aquí hay un ejemplo más complejo, reescribiendo **DUMP** usando variables locales:

```
\ variables locales en DUMP:
\ START_ADDR    \ primera dirección para el volcado
\ END_ADDR      \ última dirección para el volcado
\ 0START_ADDR   \ primera dirección del bucle en el volcado
\ LINES         \ número de líneas para el bucle de volcado
\ myBASE        \ base numérica actual
internals
: dump ( start len -- )
  cr cr ." --addr--- "
  ." 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  -----
chars-----"
  2dup + { END_ADDR }          \ store latest address to dump
  swap { START_ADDR }         \ store START address to dump
  START_ADDR 16 / 16 * { 0START_ADDR } \ calc. addr for loop start
  16 / 1+ { LINES }
  base @ { myBASE }           \ save current base
  hex
  \ outer loop
  LINES 0 do
    0START_ADDR i 16 * +      \ calc start address for current
line
    cr <# # # # # [char] - hold # # # # #> type
    space space              \ and display address
    \ first inner loop, display bytes
    16 0 do
      \ calculate real address
      0START_ADDR j 16 * i + +
      ca@ <# # # #> type space \ display byte in format: NN
    loop
    space
    \ second inner loop, display chars
    16 0 do
```

```

        \ calculate real address
        @START_ADDR j 16 * i + +
        \ display char if code in interval 32-127
        ca@      dup 32 < over 127 > or
        if      drop [char] . emit
        else    emit
        then
    loop
loop
myBASE base !      \ restore current base
cr cr
;
forth

```

El uso de variables locales simplifica enormemente la manipulación de datos en pilas. El código es más legible. Tenga en cuenta que no es necesario declarar previamente estas variables locales, basta con designarlas al utilizarlas, por ejemplo: **base @ { myBASE }** .

ADVERTENCIA: si utiliza variables locales en una definición, no utilice más las palabras **>r** y **r>** , de lo contrario corre el riesgo de alterar la gestión de las variables locales. Basta con mirar la descompilación de esta versión de **DUMP** para comprender el motivo de esta advertencia:

```

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

```


Estructuras de datos para ESP32forth

Preámbulo

ESP32forth es una versión de 32 bits del lenguaje FORTH. Quienes han practicado FORTH desde sus inicios han programado con versiones de 16 bits. Este tamaño de datos está determinado por el tamaño de los elementos depositados en la pila de datos. Para conocer el tamaño en bytes de los elementos, debes ejecutar la palabra `celda`. Ejecutando esta palabra para ESP32 en adelante:

```
cell . \ muestra 4
```

El valor 4 significa que el tamaño de los elementos colocados en la pila de datos es de 4 bytes, o $4 \times 8 \text{ bits} = 32 \text{ bits}$.

Con una versión FORTH de 16 bits, la celda apilará el valor 2. Del mismo modo, si usa una versión de 64 bits, la celda apilará el valor 8.

Tablas en FORTH

Comencemos con estructuras bastante simples: tablas. Sólo discutiremos matrices uni o bidimensionales.

Matriz de datos unidimensional de 32 bits

Este es el tipo de mesa más simple. Para crear una tabla de este tipo utilizamos la palabra **create** seguida del nombre de la tabla a crear:

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

En esta tabla almacenamos 6 valores: 34, 37....12. Para recuperar un valor, simplemente use la palabra `@` incrementando la dirección apilada por **temperatures** con el desplazamiento deseado:

temperatures	\ dirección de pila
0 cell *	\ calcular desplazamiento 0
+	\ agregar desplazamiento a la dirección
@ .	\ muestra 34
temperatures	\ dirección de pila
1 cell *	\ calcula el desplazamiento 1
+	\ agregar desplazamiento a la dirección
@ .	\ muestra 37

Podemos factorizar el código de acceso al valor deseado definiendo una palabra que calculará esta dirección:

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

Notarás que para n valores almacenados en esta tabla, aquí 6 valores, el índice de acceso siempre debe estar en el intervalo $[0..n-1]$.

Palabras de definición de tabla

A continuación se explica cómo crear una definición de palabra de matrices de enteros unidimensionales:

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

En nuestro ejemplo almacenamos 6 valores entre 0 y 255. Es fácil crear una variante de **matriz** para gestionar nuestros datos de una forma más compacta:

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

Con esta variante, los mismos valores se almacenan en cuatro veces menos espacio de memoria.

Leer y escribir en una tabla.

Es completamente posible crear una matriz vacía de n elementos y escribir y leer valores en esta matriz:

```
arrayC myCTemps
  6 allot \ reservar 6 bytes
  0 myCTemps 6 0 fill \ llenar estos 6 bytes con valor 0
```

```

32 0 myCTemps c!      \ almacena 32 en myCTemps[0]
25 5 myCTemps c!      \ almacena 25 en myCTemps[5]
0 myCTemps c@.        \ muestra 32

```

En nuestro ejemplo, la matriz contiene 6 elementos. Con ESP32 en adelante, hay suficiente espacio de memoria para procesar matrices mucho más grandes, con 1.000 o 10.000 elementos, por ejemplo. Es fácil crear tablas multidimensionales. Ejemplo de una matriz bidimensional:

```

63 constant SCR_WIDTH
16 constant SCR_HEIGHT
create mySCREEN
  SCR_WIDTH SCR_HEIGHT * allot          \ reservar 63 * 16 bytes
  mySCREEN SCR_WIDTH SCR_HEIGHT * bl fill
  \ llenar este espacio con 'espacio'

```

Aquí, definimos una tabla bidimensional llamada **mySCREEN** que será una pantalla virtual de 16 filas y 63 columnas.

Simplemente reserva un espacio de memoria que es el producto de las dimensiones X e Y de la tabla a utilizar. Ahora veamos cómo gestionar esta matriz bidimensional:

```

: 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!      \ almacena el carácter X en el cuello 15 línea 5
15 5 SCR@ emit        \ visualizaciones

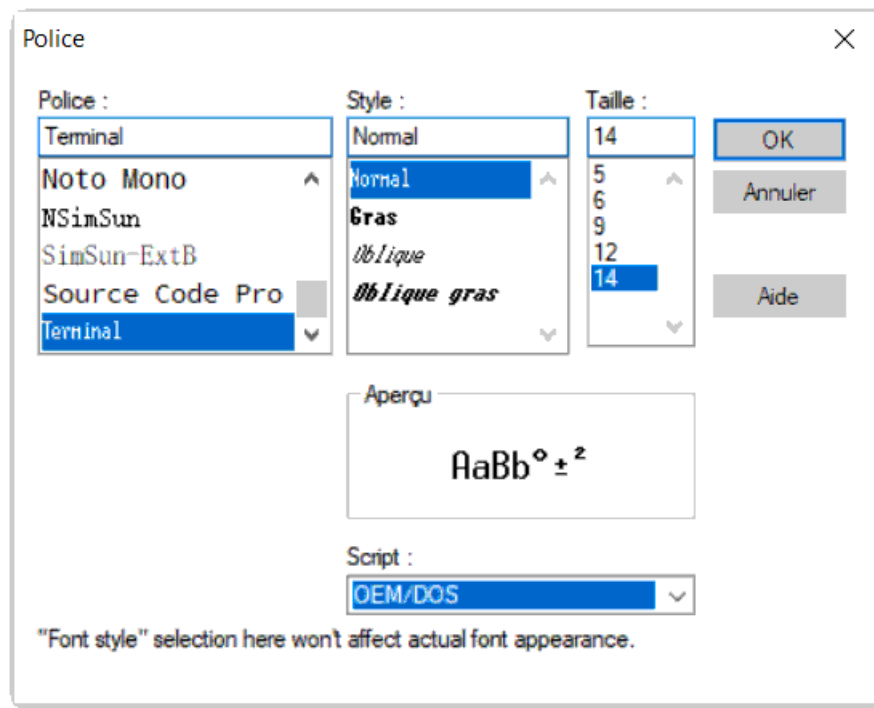
```

Ejemplo práctico de gestión de una pantalla virtual

Antes de continuar en nuestro ejemplo de gestión de una pantalla virtual, veamos cómo modificar el juego de caracteres del terminal TERA TERM y mostrarlo.

Inicie TERA TERM:

- en la barra de menú, haga clic en *Setup*
- seleccione *Font* y *Font...*
- configure la fuente a continuación:



A continuación se explica cómo mostrar la tabla de caracteres disponibles:

```
: tableChars ( -- )
  base @ >r hex
  128 32 do
    16 0 do
      j i + dup . space emit space space
    loop
  cr
  16 +loop
  256 160 do
    16 0 do
      j i + dup . space emit space space
    loop
  cr
  16 +loop
  cr
  r> base !
;
tableChars
```

Aquí está el resultado de ejecutar **tableChars**:

```
--> tableChars
```

20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F
A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF
D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF
E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF
F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF

Estos caracteres son los del conjunto ASCII de MS-DOS. Algunos de estos personajes son semigráficos. Aquí tienes una inserción muy sencilla de uno de estos personajes en nuestra pantalla virtual:

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

Ahora veamos cómo mostrar el contenido de nuestra pantalla virtual. Si consideramos cada línea de la pantalla virtual como una cadena alfanumérica, solo necesitamos definir esta palabra para mostrar una de las líneas de nuestra pantalla virtual:

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

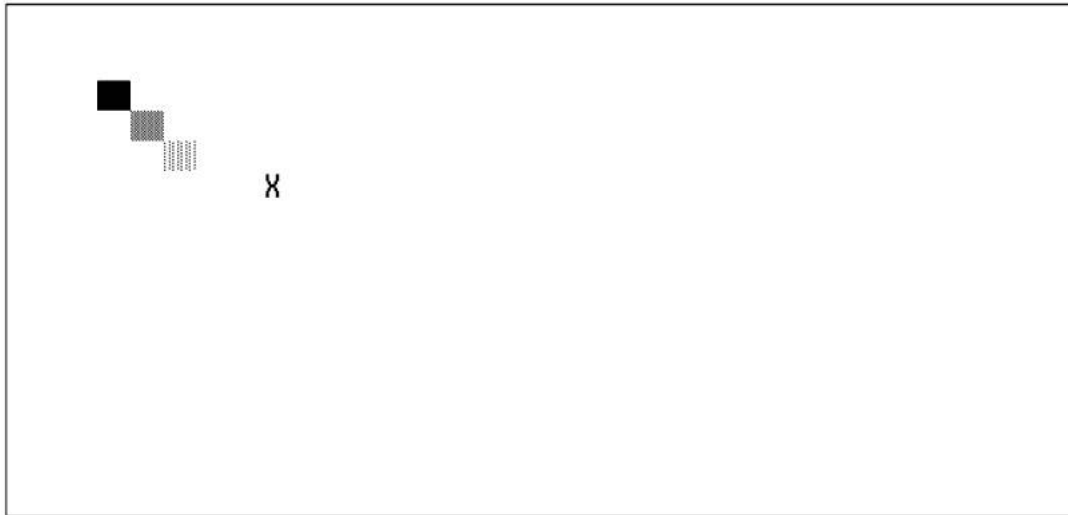
En el camino, crearemos una definición que permita mostrar el mismo carácter n veces:

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

Y ahora, definimos la palabra que nos permitirá mostrar el contenido de nuestra pantalla virtual. Para ver claramente el contenido de esta pantalla virtual, la enmarcamos con caracteres especiales:

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

Al ejecutar nuestra palabra **dispScreen** se muestra esto:



En nuestro ejemplo de pantalla virtual, mostramos que administrar una matriz bidimensional tiene una aplicación concreta. Nuestra pantalla virtual es accesible para escribir y leer. Aquí mostramos nuestra pantalla virtual en la ventana de terminal. Esta pantalla está lejos de ser eficiente. Pero puede ser mucho más rápido en una pantalla OLED real.

Gestión de estructuras complejas.

ESP32 en adelante tiene el vocabulario de estructuras. El contenido de este vocabulario permite definir estructuras de datos complejas.

Aquí hay una estructura de ejemplo trivial:

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

Aquí, definimos la estructura YMDHMS. Esta estructura gestiona los punteros **>year** **>month** **>day** **>hour** **>min** y **>sec** .

de la palabra **YMDHMS** es inicializar y agrupar los punteros en la estructura compleja. Así es como se utilizan estos consejos:

```
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

```

Hemos definido la palabra **DateTime** que es una tabla simple de 6 celdas consecutivas de 32 bits. El acceso a cada celda se realiza mediante el puntero correspondiente. Podemos redefinir el espacio asignado de nuestra estructura **YMDHMS** usando la palabra **i8** para señalar bytes:

```

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      \ muestra:
\ YEAR: 2022
\ MONTH: 3
\ DAY: 21
\ HH: 22

```

```
\ MM: 36
\ SS: 15
```

En esta estructura cYMDHMS, mantuvimos el año en formato de 32 bits y redujimos todos los demás valores a enteros de 8 bits. Vemos, en el código .cDate, que el uso de punteros permite un fácil acceso a cada elemento de nuestra compleja estructura....

Definición de sprites

Anteriormente definimos una pantalla virtual como una matriz bidimensional. Las dimensiones de esta matriz están definidas por dos constantes. Recordatorio de la definición de esta pantalla virtual:

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

La desventaja de este método de programación es que las dimensiones se definen en constantes y, por tanto, fuera de la tabla. Sería más interesante incrustar las dimensiones de la mesa en la mesa. Para ello definiremos una estructura adaptada a este caso:

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

create myVscreen \ define una pantalla de 8x32 bytes
    32 c, \ ancho de compilación
    08 c, \ altura de compilación
myVscreen >width c@
myVscreen >height c@ * allot
```

Para definir un sprite de software, simplemente compartiremos esta definición:

```
: sprite: ( width height -- )
    create
        swap c, c, \ compilar ancho y alto
    does>
;
2 1 sprite: blackChars
    $db c, $db c,
2 1 sprite: greyChars
    $b2 c, $b2 c,
blackChars >content 2 type \ muestra el contenido del sprite
blackChars
```

Aquí se explica cómo definir un sprite de 5 x 7 bytes:

```
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,

```

Para mostrar el sprite, desde una posición xy en la ventana de terminal, basta con un simple bucle:

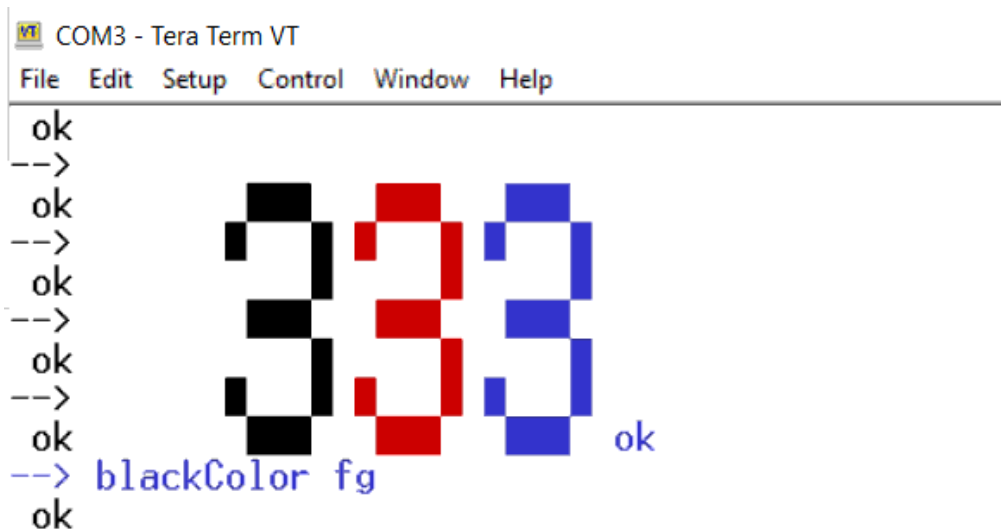
```

: .sprite { xpos ypos sprAddr -- }
  sprAddr >height c@ 0 do
    xpos ypos at-xy
    sprAddr >width c@ i *
    \ calcular el desplazamiento en los datos del sprite
    sprAddr >content +
    \ calcular la dirección real para la línea n en datos de
sprites
    sprAddr >width c@ type \ línea de visualización
    1 +to ypos            \ incrementar y posición
  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

```

Resultado de mostrar nuestro sprite:



Espero que el contenido de este capítulo te haya dado algunas ideas interesantes que te gustaría compartir...

Instalación de la biblioteca OLED para SSD1306

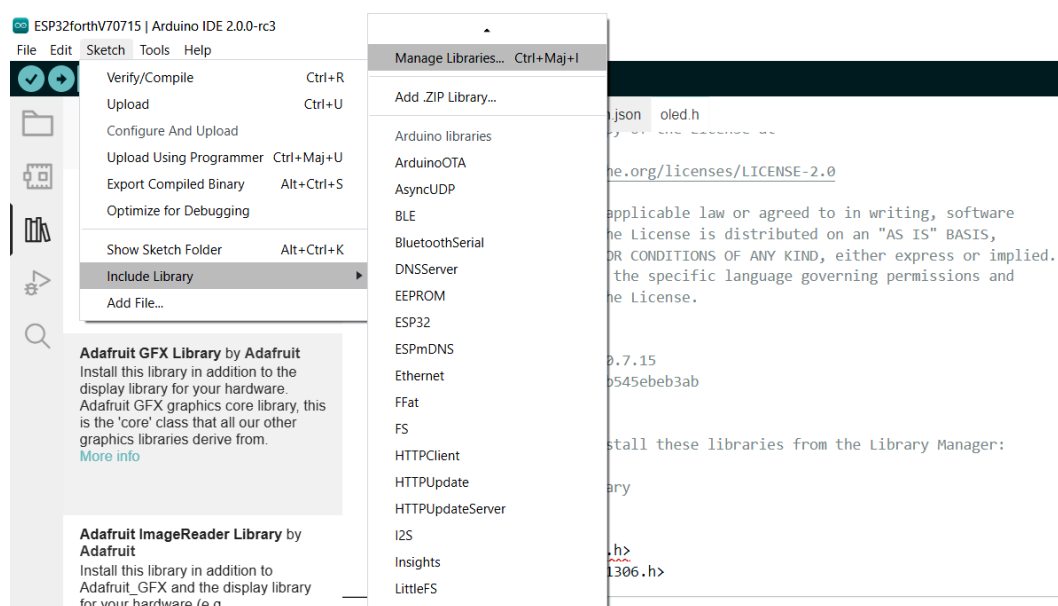
Desde ESP32 en adelante versión 7.0.7.15, las opciones están disponibles en la carpeta **optional**:

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

Para tener el vocabulario **oled**, copie el archivo **oled.h** a la carpeta que contiene el archivo **ESP32forth.ino**.

Luego inicie ARDUINO IDE y seleccione el archivo **ESP32forth.ino** más reciente.

Si la biblioteca OLED no se ha instalado, en ARDUINO IDE, haga clic en *Sketch* y seleccione *Include*, luego seleccione *Manage Libraries*.



En la barra lateral izquierda, busque la biblioteca **Adafruit SSD1306 by Adafruit**.

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

Ahora puede comenzar a compilar el boceto haciendo clic en *Sketch* y seleccionando *Upload*.

Una vez que el boceto esté cargado en la placa ESP32, inicie la terminal TeraTerm. Compruebe que el vocabulario **oled** esté presente:

```
oled vlist \ display:
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
```


Números reales con ESP32 en adelante

Si probamos la operación **1 3 /** en FORTH idioma, el resultado será 0.

No es sorprendente. Básicamente, ESP32forth solo usa enteros de 32 bits a través de la pila de datos. Los números enteros ofrecen ciertas ventajas:

- velocidad de procesamiento;
- resultado de cálculos sin riesgo de deriva en caso de iteraciones;
- Adecuado para casi todas las situaciones.

Incluso en cálculos trigonométricos podemos utilizar una tabla de números enteros. Simplemente crea una tabla con 90 valores, donde cada valor corresponde al seno de un ángulo, multiplicado por 1000.

Pero los números enteros también tienen límites:

- resultados imposibles para cálculos de división simples, como nuestro ejemplo de $1/3$;
- Requiere manipulaciones complejas para aplicar fórmulas físicas.

Desde la versión 7.0.6.5, ESP32 incluye operadores que tratan con números reales.

Los números reales también se llaman números de coma flotante.

Los reales con ESP32 en adelante

Para distinguir los números reales, deben terminar en la letra "e":

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

Es la palabra con **f.** lo que le permite mostrar un número real ubicado en la parte superior de la pila de reales.

Precisión de números reales con ESP32forth

La palabra **set-precision** le permite indicar el número de decimales que se mostrarán después del punto decimal. Veamos esto con la constante **pi** :

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

```
pi f.          \ display 3.1415
```

La precisión límite para procesar números reales con ESP32 en adelante es de seis decimales:

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

Si reducimos la precisión de visualización de los números reales por debajo de 6, los cálculos se seguirán realizando con una precisión de 6 decimales.

Constantes y variables reales

Una constante real se define con la palabra **fconstant** :

```
0.693147e fconstante ln2 \ logaritmo natural de 2
```

Una variable real se define con la palabra **fvariable** :

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

ATENCIÓN: todos los números reales pasan por la **pila de números reales** . En el caso de una variable real, sólo la dirección que apunta al valor real pasa a través de la pila de datos.

La palabra **SF!** almacena un valor real en la dirección o variable señalada por su dirección de memoria. La ejecución de una variable real coloca la dirección de memoria en la pila de datos clásica.

La palabra **SF@** apila el valor real al que apunta su dirección de memoria.

Operadores aritméticos en números reales

ESP32Forth tiene cuatro operadores aritméticos **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 también tiene estas palabras:

- **1/F** calcula el inverso de un número real;
- **fsqrt** calcula la raíz cuadrada de un número real.

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

Operadores matemáticos sobre números reales

ESP32forth tiene varios operadores matemáticos:

- **F**** eleva un r_val real a la potencia r_exp
- **FATAN2** calcula el ángulo en radianes a partir de la tangente.
- **FCOS** (r1 -- r2) Calcula el coseno de un ángulo expresado en radianes.
- **FEXP** (ln-r -- r) calcula el real correspondiente a e EXP r
- **FLN** (r -- ln-r) calcula el logaritmo natural de un número real.
- **FSIN** (r1 -- r2) calcula el seno de un ángulo expresado en radianes.
- **FSINCOS** (r1 -- rcos rsin) calcula el coseno y el seno de un ángulo expresado en radianes.

Algunos ejemplos :

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

4.605170e FEXP F.    \ mostrar 100.000018

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

Operadores lógicos en números reales

ESP32forth también le permite realizar pruebas lógicas en datos reales:

- **F0<** (r -- fl) prueba si un número real es menor que cero.
- **F0=** (r -- fl) indica verdadero si el real es cero.
- **f<** (r1 r2 -- fl) fl es verdadero si $r1 < r2$.
- **f<=** (r1 r2 -- fl) fl es verdadero si $r1 \leq r2$.
- **f<>** (r1 r2 -- fl) fl es verdadero si $r1 \neq r2$.
- **f=** (r1 r2 -- fl) fl es verdadera si $r1 = r2$.
- **f>** (r1 r2 -- fl) fl es verdadero si $r1 > r2$.
- **f>=** (r1 r2 -- fl) fl es verdadero si $r1 \geq r2$.

Entero ↔ transformaciones reales

ESP32forth tiene dos palabras para transformar números enteros en reales y viceversa:

- **F>S** (r -- n) convierte un real en un número entero. Deje la parte entera en la pila de datos si el real tiene partes decimales.
- **S>F** (n -- r: r) convierte un número entero en un número real y transfiere este número real a la pila de reales.

Ejemplo :

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

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

Mostrar números y cadenas de caracteres

Cambio de base numérica

FORTH no procesa cualquier número. Los que usó al probar los ejemplos anteriores son enteros con signo de precisión simple. El dominio de definición para enteros de 32 bits es -2147483648 a 2147483647. Ejemplo:

```
2147483647 .      \ muestra 2147483647
2147483647 1+ .   \ muestra -2147483648
-1 u.            \ muestra 4294967295
```

Estos números se pueden procesar en cualquier base numérica, siendo válidas todas las bases numéricas entre 2 y 36:

```
255 HEX . DECIMAL \ muestra FF
```

Puede elegir una base numérica aún mayor, pero los símbolos disponibles quedarán fuera del conjunto alfanumérico [0..9,A..Z] y correrán el riesgo de volverse inconsistentes.

La base numérica actual está controlada por una variable denominada **BASE** y cuyo contenido se puede modificar. Entonces, para cambiar a binario, simplemente almacene el valor **2** en **BASE**. Ejemplo:

```
2 BASE !
```

y escriba **DECIMAL** para volver a la base numérica decimal.

ESP32forth tiene dos palabras predefinidas que le permiten seleccionar diferentes bases numéricas:

- **DECIMAL** para seleccionar la base numérica decimal. Esta es la base numérica que se toma por defecto al iniciar ESP32 en adelante;
- **HEX** para seleccionar la base numérica hexadecimal.

Tras la selección de una de estas bases numéricas, los números literales se interpretarán, mostrarán o procesarán en esta base. Cualquier número ingresado previamente en una base numérica distinta de la base numérica actual se convierte automáticamente a la base numérica actual. Ejemplo :

```
DECIMAL      \ base a decimal
255          \ pila 255
HEX          \ selecciona base hexadecimal
1+          \ incrementos 255 se convierte en 256
.           \ muestra 100
```

Uno puede definir su propia base numérica definiendo la palabra apropiada o almacenando esta base en **BASE**. Ejemplo :

```
: BINARY ( ---)      \ selecciona la base numérica binaria
  2 BASE ! ;
DECIMAL 255 BINARY .  \ muestra 11111111
```

El contenido de **BASE** se puede apilar como el contenido de cualquier otra variable:

```
VARIABLE RANGE_BASE  \ Definición de variable RANGE-BASE
BASE @ RANGE_BASE !   \ contenido de almacenamiento BASE en RANGE-
BASE
HEX FF 10 + .         \ muestra 10F
RANGE_BASE @ BASE !   \ restaura BASE con contenidos de RANGE-BASE
```

En una definición : , el contenido de **BASE** puede pasar a través de la pila de retorno:

```
: OPERATION ( ---)
  BASE @ >R           \ almacena BASE en la pila posterior
  HEX FF 10 + .       \ operación del ejemplo anterior
  R> BASE ! ;         \ restaura el valor BASE inicial
```

ADVERTENCIA : las palabras **>R** y **R>** no se pueden utilizar en modo interpretado. Sólo puedes utilizar estas palabras en una definición que será compilada.

Definición de nuevos formatos de visualización.

Forth tiene primitivas que le permiten adaptar la visualización de un número a cualquier formato. Con ESP32 en adelante, estas primitivas procesan números enteros:

- **<#** comienza una secuencia de definición de formato;
- **#** inserta un dígito en una secuencia de definición de formato;
- **#S** equivale a una sucesión de **#** ;
- **HOLD** inserta un carácter en una definición de formato;
- **#>** completa una definición de formato y deja en la pila la dirección y la longitud de la cadena que contiene el número a mostrar.

Estas palabras sólo pueden usarse dentro de una definición. Ejemplo, ya sea para mostrar un número que exprese un importe denominado en euros con la coma como separador decimal:

```
: .EUROS ( n ---)
  <# # # [char] , hold #S #>
  type space ." EUR" ;
1245 .euros
```

Ejemplos de ejecución:

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

En la definición de **EUROS**, la palabra **<#** comienza la secuencia de definición del formato de visualización. Las dos palabras **#** colocan los dígitos de las unidades y las decenas en la cadena de caracteres. La palabra **HOLD** coloca el carácter **,** (coma) después de los dos dígitos de la derecha, la palabra **#S** completa el formato de visualización con los dígitos distintos de cero después de **,**. La palabra **#>** cierra la definición de formato y coloca en la pila la dirección y la longitud de la cadena que contiene los dígitos del número a mostrar. La palabra **TYPE** muestra esta cadena de caracteres.

En tiempo de ejecución, una secuencia de formato de visualización trata exclusivamente con enteros de 32 bits con o sin signo. La concatenación de los distintos elementos de la cadena se realiza de derecha a izquierda, es decir, empezando por los dígitos menos significativos.

El procesamiento de un número mediante una secuencia de formato de visualización se ejecuta en función de la base numérica actual. La base numérica se puede modificar entre dos dígitos.

Aquí hay un ejemplo más complejo que demuestra la compacidad de FORTH. Esto implica escribir un programa que convierta cualquier cantidad de segundos al formato HH:MM:SS:

```
: :00 ( ---)
  DECIMAL #          \ insertar unidad de dígito en decimal
  6 BASE !           \ selección base 6
  #                  \ insertar dígito diez
  [char] : HOLD      \ carácter de inserción:
  DECIMAL ;          \ devolver base decimal
: HMS ( n ---)       \ muestra el número de segundos en formato
HH:MM:SS
  <# :00 :00 #S #> TYPE SPACE ;
```

Ejemplos de ejecución:

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

Explicación: El sistema para mostrar segundos y minutos se llama sistema sexagesimal. Las unidades se expresan en base numérica **decimal**, las decenas se expresan en base seis. La palabra **:00** gestiona la conversión de unidades y decenas en estas dos bases para formatear los números correspondientes a segundos y minutos. Para los tiempos, los números son todos decimales.

Otro ejemplo, para definir un programa que convierte un entero decimal de precisión simple en binario y lo muestra en el formato bbbb bbbb bbbb bbbb:

```

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

```

Ejemplo de ejecución:

```

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

```

Otro ejemplo es crear una agenda telefónica donde uno o más números de teléfono estén asociados a un apellido. Definimos una palabra por apellido:

```

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

```

Este directorio telefónico, que puede compilarse a partir de un archivo fuente, es fácilmente editable y, aunque los nombres no están clasificados, la búsqueda es extremadamente rápida.

Mostrar caracteres y cadenas de caracteres

Se muestra un carácter usando la palabra **EMIT**:

```

65 EMIT          \ muestra A

```

Los caracteres visualizables están en el rango 32..255. También se mostrarán códigos entre 0 y 31, sujeto a que ciertos caracteres se ejecuten como códigos de control. Aquí hay una definición que muestra todo el juego de caracteres de la tabla ASCII:

```

variable #out
: #out+! ( n -- )
  #out +!          \ incremente #out
;
: (.) ( n -- a l )

```

```

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

```

Al ejecutar **JEU-ASCII** se muestran los códigos ASCII y los caracteres cuyo código esté entre 32 y 127. Para mostrar la tabla equivalente con los códigos ASCII en hexadecimal, escriba **HEX JEU-ASCII** :

```

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

```

Las cadenas de caracteres se muestran de varias maneras. El primero, utilizable sólo en compilación, muestra una cadena de caracteres delimitada por el carácter " (comillas):

```

: TITRE ." MENU GENERAL" ;
  TITRE    \ muestra MENÚ GENERAL

```

La cadena está separada de la palabra **."** por al menos un carácter de espacio.

Una cadena de caracteres también puede estar compilada por la palabra **s"** y delimitada por el carácter " (comillas):

```

: LIGNE1 ( --- adr len)
  S" E..Registro de datos" ;

```

La ejecución de **LIGNE1** coloca la dirección y la longitud de la cadena compilada en la definición en la pila de datos. La visualización se realiza mediante la palabra **TYPE**:

```
LIGNE1 TYPE \ muestra E..Registro de datos
```

Al final de mostrar una cadena de caracteres, se debe activar el salto de línea si se desea:

```
CR TITRE CR CR LIGNE1 TYPE CR
\ display:
\ MENÚ GENERAL
\
\ E..Registro de datos
```

Se pueden agregar uno o más espacios al inicio o al final de la visualización de una cadena alfanumérica:

```
SPACE \ muestra un carácter de espacio
10 SPACES \ muestra 10 caracteres de espacio
```

Variables de cadena

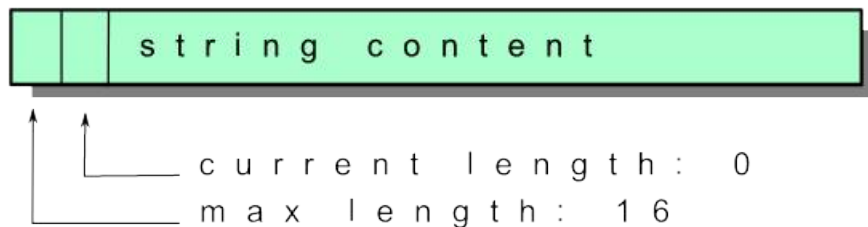
Las variables de texto alfanuméricas no existen de forma nativa en ESP32 en adelante. Aquí está el primer intento de definir la palabra **string** :

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

Una variable de cadena de caracteres se define así:

```
16 string strState
```

Así se organiza el espacio de memoria reservado para esta variable de texto:



Código de palabra de gestión de variables de texto

Aquí está el código fuente completo para gestionar variables de texto:

```

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

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

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

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

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

\ Example:
\ : s1
\     s" this is constant string" ;

```



```

\ 200 string test
\ s1 test $!

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

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

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

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

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

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

```

Crear una cadena de caracteres alfanuméricos es muy sencillo:

```
64 string myNewString
```

Aquí creamos una variable alfanumérica **myNewString** que puede contener hasta 64 caracteres.

Para mostrar el contenido de una variable alfanumérica, simplemente use **el tipo** .
Ejemplo :

```
s"Este es mi primer ejemplo..." myNewString $!
myNewString type \ display: Este es mi primer ejemplo.
```

Si intentamos guardar una cadena de caracteres más larga que el tamaño máximo de nuestra variable alfanumérica, la cadena se truncará:

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

Agregar carácter a una variable alfanumérica

Algunos dispositivos, como el transmisor LoRa, requieren procesar líneas de comando que contienen caracteres no alfanuméricos. La palabra **c+\$!** permite la inserción de este código:

```
32 string AT_BAND
s" AT+BAND=868500000" AT_BAND $! \ set frequency at 865.5 Mhz
$0a AT_BAND c+$!
$0d AT_BAND c+$! \agregar código CR LF al final del comando
```

El volcado de memoria del contenido de nuestra variable alfanumérica **AT_BAND** confirma la presencia de los dos caracteres de control al final de la cadena:

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

Aquí tienes una forma inteligente de crear una variable alfanumérica que te permitirá transmitir un retorno de carro, un **CR+LF** compatible con el final de los comandos del transmisor LoRa:

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

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

Vocabularios con ESP32forth

En FORTH, la noción de procedimiento y función no existe. Las FORTH instrucciones se llaman PALABRAS. Como un lenguaje tradicional, FORTH organiza las palabras que lo componen en VOCABULARIOS, un conjunto de palabras con un rasgo común.

Programar en FORTH consiste en enriquecer un vocabulario existente, o definir uno nuevo, relacionado con la aplicación que se está desarrollando.

Lista de vocabularios

Un vocabulario es una lista ordenada de palabras, buscadas desde las creadas más recientemente hasta las creadas menos recientemente. El orden de búsqueda es una pila de vocabularios. Al ejecutar un nombre de vocabulario, se reemplaza la parte superior de la pila del orden de búsqueda con ese vocabulario.

Para ver la lista de diferentes vocabularios disponibles en ESP32 en adelante, usaremos la palabra **voclist** :

```
--> internals voclist      \ displays
registers
ansi
editor
streams
tasks
rtos
sockets
Serial
ledc
SPIFFS
SD_MMC
SD
WiFi
Wire
ESP
structures
internalized
internals
FORTH
```

Esta lista no es limitada. Pueden aparecer vocabularios adicionales si compilamos determinadas extensiones.

El vocabulario principal se llama FORTH . Todos los demás vocabularios están adjuntos al vocabulario **FORTH** .

Vocabularios esenciales

Aquí está la lista de los principales vocabularios disponibles en ESP32 en adelante:

- **ansi** en un terminal ANSI;
- **editor** proporciona acceso a comandos para editar archivos de bloques;
- **oled** de pantallas OLED de 128 x 32 o 128 x 64 píxeles. El contenido de este vocabulario sólo está disponible después de compilar la extensión **oled.h** ;
- **structures** gestión de estructuras complejas;

Lista de contenidos de vocabulario

Para ver el contenido de un vocabulario utilizamos la palabra **vlist** habiendo seleccionado previamente el vocabulario adecuado:

```
sockets vlist
```

vocabulario **de sockets** y muestra su contenido:

```
--> sockets vlist \ affiche:  
ip. ip# ->h_addr ->addr! ->addr@ ->port! ->port@ sockaddr l, s, bs, SO_REUSEADDR  
SOL_SOCKET sizeof(sockaddr_in) AF_INET SOCK_RAW SOCK_DGRAM SOCK_STREAM  
socket setsockopt bind listen connect sockaccept select poll send sendto  
sendmsg recv recvfrom recvmsg gethostbyname errno sockets-builtins
```

La selección de un vocabulario da acceso a las palabras definidas en este vocabulario.

no se puede acceder a la palabra **voclist** sin invocar primero el vocabulario **internals**.

La misma palabra se puede definir en dos vocabularios diferentes y tener dos acciones diferentes: la palabra **l** se define tanto en el vocabulario **asm** como en **editor** .

Esto es aún más obvio con la palabra **server**, definida en los vocabularios **httpd** , **telnetd** y **web-interface**.

Usando palabras de vocabulario

Para compilar una palabra definida en un vocabulario distinto del FORTH, existen dos soluciones. La primera solución es simplemente llamar a este vocabulario antes de definir la palabra que utilizará palabras de este vocabulario.

Aquí, definimos una palabra **serial2-type** que usa la palabra **Serial2.write** definida en el vocabulario **serial** :

```
serial \ Selection vocabulaire Serial
: serial2-type ( a n -- )
  Serial2.write drop
;
```

La segunda solución te permite integrar una sola palabra de un vocabulario específico:

```
: serial2-type ( a n -- )
  [ serial ] Serial2.write [ FORTH ]
  \ compile mot depuis vocabulaire serial
drop
;
```

La selección de un vocabulario se puede realizar de forma implícita a partir de otra palabra del vocabulario **FORTH**.

Encadenamiento de vocabularios

El orden en el que se busca una palabra en un vocabulario puede ser muy importante. En el caso de palabras con el mismo nombre eliminamos cualquier ambigüedad controlando el orden de búsqueda en los diferentes vocabularios que nos interesan.

Antes de crear una cadena de vocabularios, restringimos el orden de búsqueda palabra **only**:

```
asm xtensa
order \ affiche:      xtensa >> asm >> FORTH
only
order \ affiche:      FORTH
```

Luego duplicamos el encadenamiento de vocabularios con la palabra **also** :

```
only
order \ affiche:      FORTH
asm also
order \ affiche:      asm >> FORTH
xtensa
order \ affiche:      xtensa >> asm >> FORTH
```

Aquí hay una secuencia de encadenamiento compacta:

```
only asm also xtensa
```

El último vocabulario así encadenado será el primero explorado cuando ejecutemos o compilemos una nueva palabra.

```
only
order      \ affiche:      FORTH
also ledc also serial also SPIFFS
order      \ affiche:      SPIFFS >> FORTH
           \               Serial >> FORTH
           \               ledc >> FORTH
           \               FORTH
```

El orden de búsqueda, aquí, comenzará con el vocabulario **SPIFFS** , luego **Serial** , luego **ledc** y finalmente el vocabulario **FORTH** :

- si no se encuentra la palabra buscada, hay un error de compilación;
- si la palabra se encuentra en un vocabulario, es esta palabra la que se compilará, incluso si está definida en el siguiente vocabulario;

Palabras de acción retrasada

Las palabras de acción diferida se definen mediante la palabra de definición **defer**. Para comprender los mecanismos y el interés en explotar este tipo de palabras, veamos con más detalle el funcionamiento del intérprete interno del lenguaje FORTH.

Cualquier definición compilada por **:** (dos puntos) contiene una secuencia de direcciones codificadas correspondientes a los campos de código de las palabras previamente compiladas. En el corazón del sistema FORTH, la palabra **EXECUTE** acepta como parámetros estas direcciones de campo de código, direcciones que abreviamos con **cfa** para Dirección de campo de código. Cada palabra FORTH tiene un **cfa** y esta dirección es utilizada por el intérprete interno de FORTH:

```
' <mot>
\ coloca el cfa de <palabra> en la pila de datos
```

Ejemplo:

```
' WORDS
\apila las WORDS cfa.
```

A partir de este **cfa**, conocido como único valor literal, la ejecución de la palabra se puede realizar con **EXECUTE** :

```
' WORDS EXECUTE
\ ejecuta PALABRAS
```

Por supuesto, hubiera sido más fácil escribir **PALABRAS directamente**. Desde el momento en que un **cfa** está disponible como único valor literal, puede manipularse y, en particular, almacenarse en una variable:

```
variable vector
' WORDS vector !
vector @ .
\ muestra cfa de WORDS almacenadas en la variable vectorial
```

Puede ejecutar **WORDS** indirectamente desde el contenido del **vector** :

```
vector @ EXECUTE
```

Esto inicia la ejecución de la palabra cuyo **cfa** se almacenó en la variable **vector** y luego se vuelve a colocar en la pila antes de que **EXECUTE** la use.

Este es un mecanismo similar que es explotado por la parte de ejecución de la palabra de definición de **defer**. Para simplificar, **defer** crea un encabezado en el diccionario, como una **variable** o **constante**, pero en lugar de simplemente colocar una dirección o valor en la pila, inicia la ejecución de la palabra cuyo cfa **se** almacenó en el área paramétrica de la palabra definida. por **defer**.

Definición y uso de palabras con defer

La inicialización de una palabra definida por defer se realiza **mediante** :

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

La ejecución de **vector** provoca que se ejecute la palabra cuyo **cfa** fue asignado previamente:

```
vector      \ ejecuta  words
```

Una palabra creada por **defer** se utiliza para ejecutar otra palabra sin invocar explícitamente esa palabra. El principal interés de este tipo de palabras reside sobre todo en la posibilidad de modificar la palabra a ejecutar:

```
' page is vector
```

El **vector** ahora ejecuta **page** y ya no **words**.

Básicamente utilizamos las palabras definidas por **defer** en dos situaciones:

- definición de referencia directa;
- Definición de una palabra dependiendo del contexto operativo.

En el primer caso, la definición de una referencia anterior permite superar las limitaciones de la sacrosanta precedencia de las definiciones.

En el segundo caso, la definición de una palabra en función del contexto operativo permite resolver la mayoría de los problemas de interfaz con un entorno de software en evolución, mantener la portabilidad de las aplicaciones, adaptar el comportamiento de un programa a situaciones controladas por varios parámetros sin perjudicar el rendimiento del software.

Establecer una referencia directa

A diferencia de otros compiladores, FORTH no permite compilar una palabra en una definición antes de definirla. Este es el principio de precedencia de las definiciones:

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

Esto genera un error al compilar **word1** , porque **word2** aún no está definida. A continuación se explica cómo sortear esta restricción con **defer**:

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

Esta vez **word2** se compiló sin errores. No es necesario asignar un cfa a la palabra de ejecución vectorizada **word2** . Solo después de la definición de **(word2)** se actualiza el área de parámetros de **word2** . Después de la asignación de la palabra de ejecución

vectorizada **word2** , **word1** podrá ejecutar el contenido de su definición sin errores. La explotación de las palabras creadas por **defer** en esta situación debe seguir siendo excepcional.

Dependencia del contexto operativo

ESP32forth utiliza de forma nativa una conexión a través del puerto serie 1 como flujo de entrada y salida.

En el código fuente de ESP32forth, encontramos estas líneas:

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

Para pasar por el puerto serie, ESP32forth inicializa la palabra **type** así:

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

flujo **de type** se redirigirá de la siguiente manera:

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

type de redirección si utilizamos un flujo TELNET:

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

Y si quisiéramos redirigir la visualización de texto a una pantalla OLED, simplemente tendríamos que actuar sobre el **type** de letra de la misma manera. En el capítulo *Configuración del transmisor LoRa REYAX RYLR890* , explotamos esta propiedad **de type** de la siguiente manera:

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

```
;
```

Al hacer esto, resulta muy fácil transmitir un flujo de texto al puerto serie 2:

```
: optionChoice
  ." choice option:" ;
optionChoice      \ display      choice options:  on terminal
typeToLoRa
optionChoice      \ display      choice options:  thru serial2
typeToTerm        \ restaure normal display
```

En este caso específico, definimos muchas palabras que le permiten controlar un transmisor LoRa usando palabras comunes como **emit**, **type**,.. Etc. Si no activamos la transmisión al puerto serie 2, por lo tanto al transmisor LoRa, las palabras que comunican con este transmisor se pueden desarrollar fácilmente:

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

Si ejecutamos **ATaddress** , el flujo de texto se mostrará en el terminal. Si siguió correctamente, sabrá qué palabra ejecutar para redirigir el flujo desde **ATaddress** al puerto serie 2.

En resumen, gracias a las palabras de ejecución diferida, podemos actuar sobre la acción de FORTH palabras ya definidas.

Un caso práctico

Tienes una aplicación para crear, con visualizaciones en dos idiomas. He aquí una forma inteligente de explotar una palabra definida por diferir para generar texto en francés o inglés. Para empezar simplemente crearemos una tabla de días en inglés:

```
:noname s" Saturday" ;
:noname s" Friday" ;
:noname s" Thursday" ;
:noname s" Wednesday" ;
:noname s" Tuesday" ;
:noname s" Monday" ;
:noname s" Sunday" ;

create ENdayNames ( --- addr)
  , , , , , , ,
```

Luego creamos una tabla similar para los días en francés:

```
:noname s" Samedi" ;
```

```

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

create FRdayNames ( --- addr)
    , , , , , , ,

```

Finalmente creamos nuestra palabra de acción diferida **dayNames** y cómo inicializarla:

```

defer dayNames

: in-ENGLISH
    ['] ENdayNames is dayNames ;

: in-FRENCH
    ['] FRdayNames is dayNames ;

```

Aquí están ahora las palabras para gestionar estas dos tablas:

```

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

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

```

¡Esto es lo que hace ejecutar **getDay** :

```

en INGLÉS 3 getDay escriba cr \ display: miércoles
en FRANCÉS 3 getDay escriba cr \ display: mercredi

```

Aquí definimos la palabra **.dayList** que muestra el inicio de los nombres de los días de la semana:

```

: .dayList { tamaño -- }
    tamaño al díaLongitud
    7 0 hacer
    Obtengo espacio tipo día
    bucle
;

in-ENGLISH 3 getDay type cr \ display : Wednesday

```

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

En la segunda línea solo mostramos la primera letra de cada día de la semana.

En este ejemplo, aprovechamos **defer** para simplificar la programación. En el desarrollo web, usaríamos plantillas *para* administrar sitios multilingües. En FORTH, simplemente movemos un vector en una palabra de acción retardada. Aquí sólo manejamos dos idiomas. Este mecanismo se puede extender fácilmente a otros idiomas, porque hemos separado la gestión de mensajes de texto de la parte puramente de la aplicación.

Palabras de creación de palabras

FORTH es más que un lenguaje de programación. Es un metalenguaje. Un metalenguaje es un lenguaje utilizado para describir, especificar o manipular otros lenguajes.

Con ESP32 en adelante, se puede definir la sintaxis y la semántica de las palabras de programación más allá del marco formal de las definiciones básicas.

Ya hemos visto las palabras definidas por **constante** , **variable** , **valor** . Estas palabras se utilizan para gestionar datos digitales.

En el capítulo siguiente Estructuras de datos para ESP32, también usamos la palabra **crear** . Esta palabra crea un encabezado que permite el acceso a un área de datos almacenados en la memoria. Ejemplo :

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

Aquí, cada valor se almacena en el área de parámetros de la palabra **temperatures** con la palabra **,**.

Con ESP32forth, veremos cómo personalizar la ejecución de las palabras definidas por **create** .

Usando does>

palabras clave "**CREATE**" y "**DOES>**", que a menudo se usan juntas para crear palabras personalizadas (palabras de vocabulario) con comportamientos específicos.

Así es como funciona generalmente en Forth:

- **CREATE** : esta palabra clave se utiliza para crear un nuevo espacio de datos en el diccionario ESP32Forth. Se necesita un argumento, que es el nombre que le das a tu nueva palabra;
- **DOES>** : esta palabra clave se utiliza para definir el comportamiento de la palabra que acaba de crear con **CREATE**. Le sigue un bloque de código que especifica qué debe hacer la palabra cuando se encuentra durante la ejecución del programa.

Juntos se parece a esto:

```
forth
CREATE mi-nueva-palabra
  \ código a ejecutar cuando encuentre mi-nueva-palabra
DOES>
;
```

se encuentra la palabra **mi-nueva-palabra** en el programa FORTH, el código especificado en la parte **does>...;** será ejecutado.

```
\ definir un registro, similar a una constante
: defREG:
  create ( addr1 -- <name> )

  ,
  does> ( -- regAddr )
    @

;
```

Aquí definimos la palabra de definición **defREG:** que tiene exactamente la misma acción que **constante** . Pero ¿por qué crear una palabra que recrea la acción de una palabra que ya existe?

```
$3FF44004 constant GPIO_OUT_REG
```

O

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

son similares. Sin embargo, al crear nuestros registros con **defREG:** tenemos las siguientes ventajas:

- un código fuente ESP32forth más legible. Detectamos fácilmente todas las constantes que nombran un registro ESP32;
- nos dejamos la posibilidad de modificar la parte **does>** de **defREG:** sin tener que reescribir luego las líneas de código que no usarían **defREG:**

Aquí hay un caso clásico, procesando una tabla de datos:

```
\ palabra de definición para matriz unidimensional
: array ( comp: -- <name> | exec: index <name> -- addr )
  create
  does>
    swap cell * +
;
array temperatures
  21 ,    32 ,    45 ,    44 ,    28 ,    12 ,
0 temperatures @ . \ display 21
5 temperatures @ . \ display 12
```

La ejecución de **temperatures** debe ir precedida de la posición del valor a extraer en esta tabla. Aquí solo obtenemos la dirección que contiene el valor a extraer.

Ejemplo de gestión del color

En este primer ejemplo, definimos la palabra **color:** que recuperará el color para seleccionarlo y almacenarlo en una variable:

```
0 value currentCOLOR
```

```
\ definir palabra como COLOR constante
```

```

: color: ( n -- <name> )
  create
    ,
  does>
    @ to currentCOLOR
;

$00 color: setBLACK
$ff color: setWHITE

```

Ejecutar la palabra **setBLACK** o **setWHITE** simplifica enormemente el código ESP32. Sin este mecanismo, se habría tenido que repetir periódicamente una de estas líneas:

```
$00 currentCOLOR !
```

O

```

$00 constant BLACK
BLACK currentCOLOR !

```

Ejemplo, escribir en pinyin

Pinyin se usa comúnmente en todo el mundo para enseñar la pronunciación del chino mandarín y también se usa en varios contextos oficiales en China, como letreros de calles, diccionarios y libros de texto de aprendizaje. Facilita el aprendizaje de chino para las personas cuya lengua materna utiliza el alfabeto latino.

Para escribir chino en un teclado QWERTY, los chinos generalmente utilizan un sistema llamado "entrada pinyin". Pinyin es un sistema de romanización del chino mandarín, que utiliza el alfabeto latino para representar los sonidos del mandarín.

En un teclado QWERTY, los usuarios escriben sonidos mandarín usando la romanización pinyin. Por ejemplo, si alguien quiere escribir el carácter "你" ("nǐ" significa "tu" o "toi" en inglés), puede escribir "ni".

En este código muy simplificado, puedes programar palabras pinyin para escribirlas en mandarín. El siguiente código sólo funciona con el terminal PuTTY:

```

\ Trabajar sólo con terminal PuTTY
internals
: chinese:
  create ( c1 c2 c3 -- )
    c, c, c,
  does>
    3 serial-type
;
forth

```

Para encontrar el código UTF8 de un carácter chino, copie el carácter chino, de Google Translate, por ejemplo. Ejemplo :

```
Buenos días --> 早安 (Zao an)
```

Copie 早 y vaya a la terminal PuTTY y escriba:

```
key key key \ seguida de la tecla <enter>
```

pega el carácter 早. ESP32forth debería mostrar los siguientes códigos:

```
230 151 169
```

Para cada carácter chino, usaremos estos tres códigos de la siguiente manera:

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

Usar :

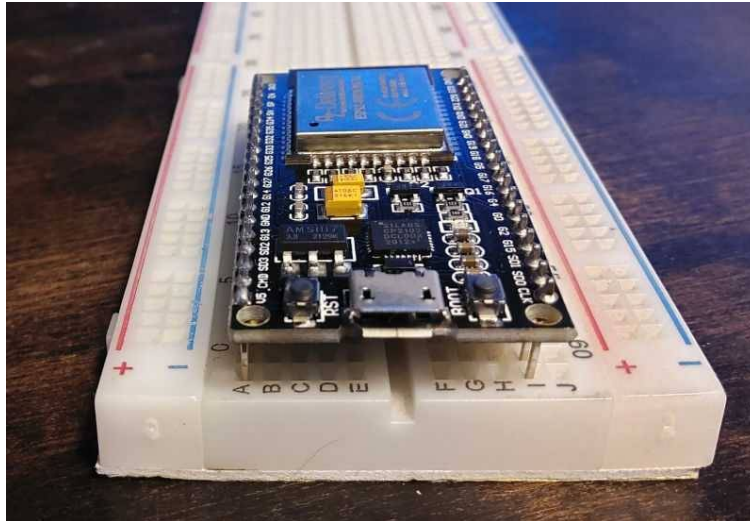
```
Zao An \ display 早安
```

Admite que programar como este es algo distinto a lo que puedes hacer en lenguaje C.
¿No?

Adaptar placas de pruebas a la placa ESP32

Placas de prueba para ESP32

Acabas de recibir tus tarjetas ESP32. Y primera mala sorpresa, esta tarjeta encaja muy mal en el tablero de prueba:

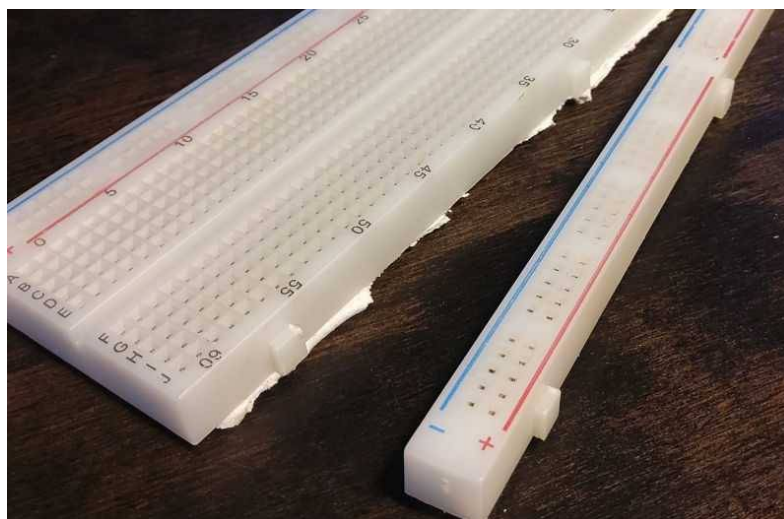


No existe una placa de pruebas específicamente adaptada a las placas ESP32.

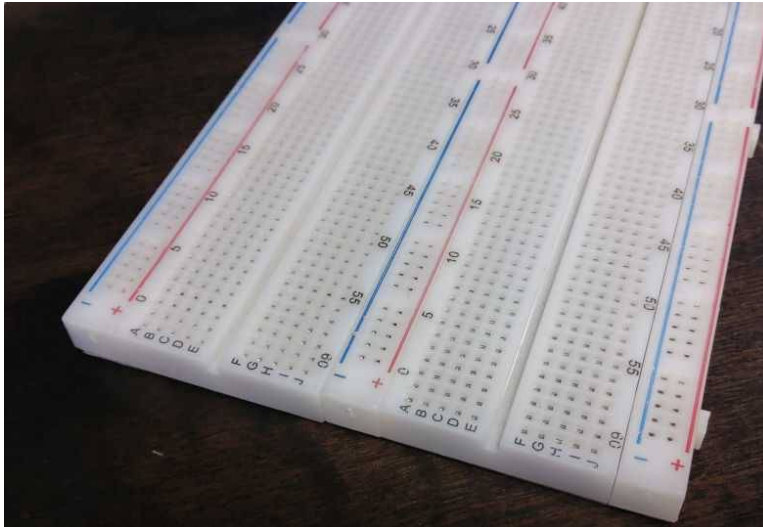
Construya una placa de pruebas adecuada para la placa ESP32

Vamos a construir nuestra propia placa de prueba. Para ello es necesario disponer de dos placas de prueba idénticas.

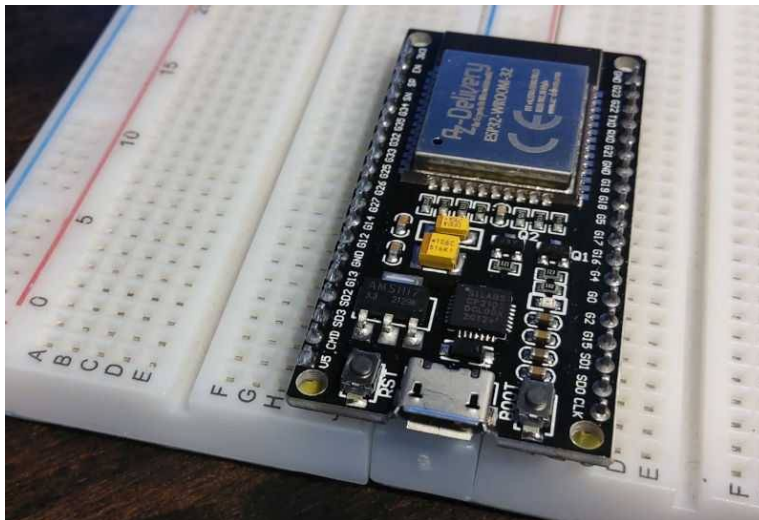
En una de las placas quitaremos un cable eléctrico. Para hacer esto, use un cortador y corte desde abajo. Debería poder separar esta línea eléctrica de esta manera:



Luego podemos volver a ensamblar todo el mapa con este mapa. Tienes vigas a los lados de las placas de prueba para conectarlas entre sí:



¡Y ahí lo tienes! Ya podemos colocar nuestra tarjeta ESP32:



Los puertos de E/S se pueden ampliar sin dificultad.

Alimentando la placa ESP32

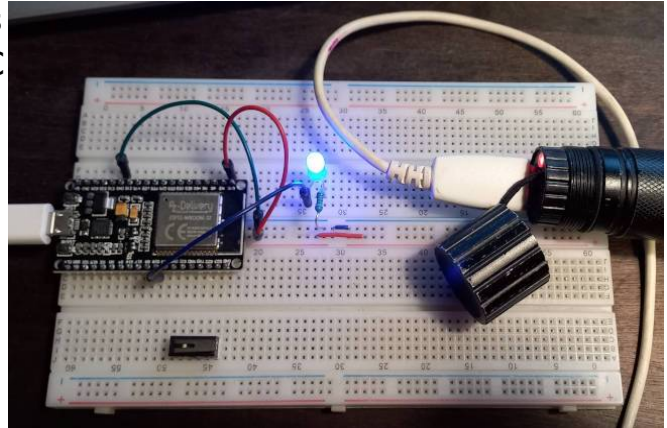
Elección de la fuente de energía

Aquí veremos cómo alimentar una tarjeta ESP32. El objetivo es brindar soluciones para ejecutar programas FORTH compilados por ESP32forth.

Alimentado por conector mini-USB

Ésta es la solución más sencilla. Sustituimos la fuente de alimentación procedente del PC por una fuente diferente:

- una fuente de alimentación de red como las que se utilizan para cargar un teléfono móvil;
- una batería de respaldo para un teléfono móvil (power bank).



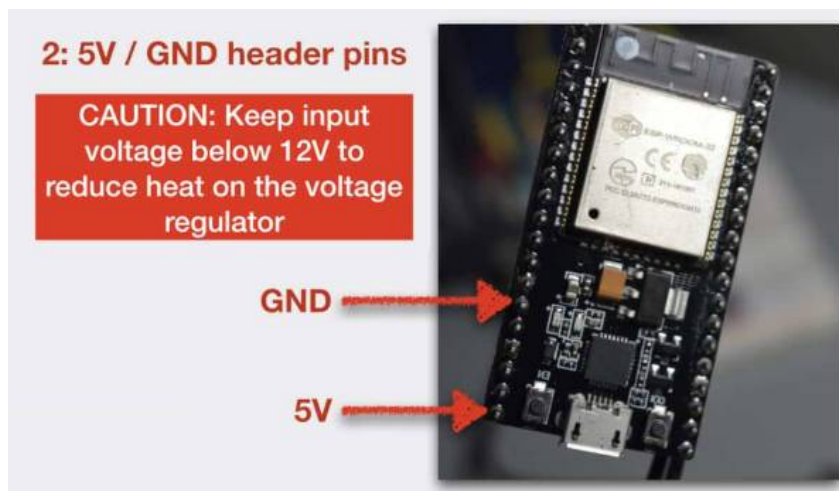
Aquí alimentamos nuestra placa ESP32 con una batería de respaldo para dispositivos móviles.

Alimentación mediante pin de 5V

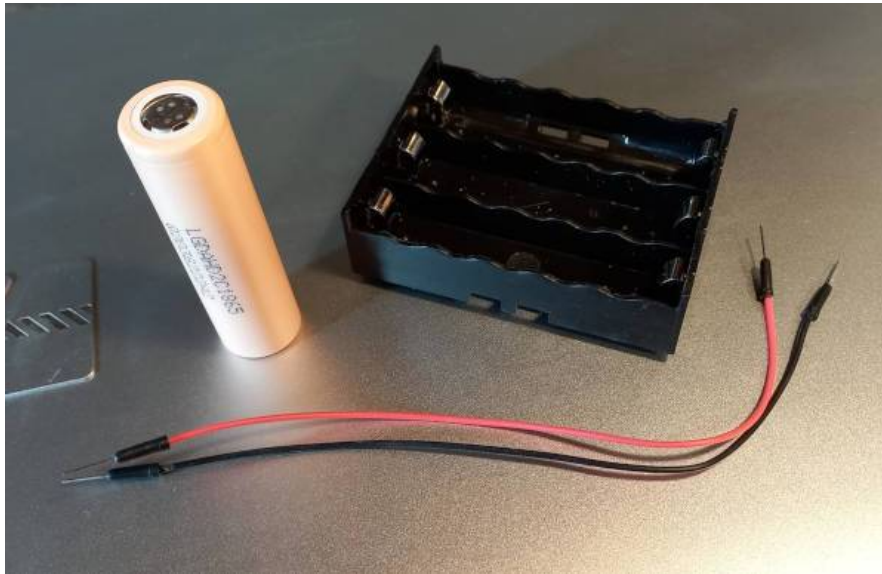
La segunda opción es conectar una fuente de alimentación externa no regulada al pin de 5V y a tierra. Cualquier voltaje entre 5 y 12 voltios debería funcionar.

Pero es mejor mantener el voltaje de entrada alrededor de 6 o 7 voltios para evitar perder demasiada energía en forma de calor en el regulador de voltaje.

Estos son los terminales que permiten una fuente de alimentación externa de 5-12 V:



Para utilizar la fuente de alimentación de 5V, necesita este equipo:



- dos baterías de litio de 3,7 V
- un soporte de batería
- dos hijos dupont

Soldamos un extremo de cada cable dupont a los terminales del soporte de la batería. Aquí nuestro soporte acepta tres baterías. Solo operaremos una unidad de batería. Las baterías están conectadas en serie.

Una vez soldados los cables dupont instalamos la batería y comprobamos que se respeta la polaridad de salida:



Ahora podemos alimentar nuestra tarjeta ESP32 a través del pin de 5V.

ATENCIÓN : la tensión de la batería debe estar entre 5 y 12 Voltios.

Inicio automático de un programa.

¿Cómo podemos estar seguros de que la tarjeta ESP32 funciona bien una vez alimentada por nuestras baterías?

La solución más sencilla es instalar un programa y configurarlo para que se inicie automáticamente cuando se enciende la placa ESP32. Compile este programa:

```
18 constant myLED

0 value LED_STATE

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

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

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

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

startup: led.blink
bye
```

Instale un LED en el pin G18.

Apague la alimentación y vuelva a conectar la placa ESP32. Si todo ha ido bien, el LED debería parpadear al cabo de unos segundos. Esta es una señal de que el programa se está ejecutando cuando se inicia la placa ESP32.

Desenchufe el puerto USB y conecte la batería. La placa ESP32 debería iniciarse y el LED parpadeará.

Todo el secreto reside en la secuencia: **startup: led.blink** . Esta secuencia congela el código FORTH compilado por ESP32forth y designa la palabra **led.blink** como la palabra que se ejecutará al iniciar ESP32forth una vez que se enciende la placa ESP32.

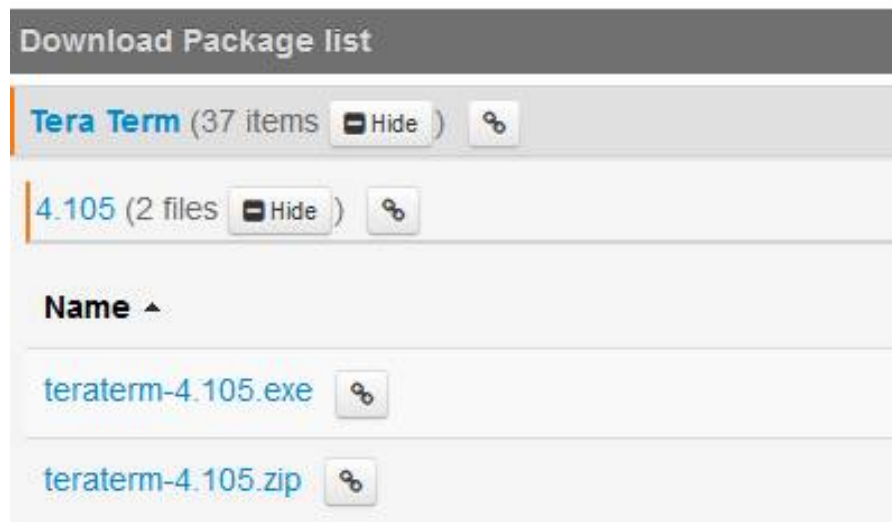
Instale y use la terminal Tera Term en Windows

Instalar Tera Term

La página en inglés de Tera Term está aquí:

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

Vaya a la página de descarga, obtenga el archivo exe o zip:

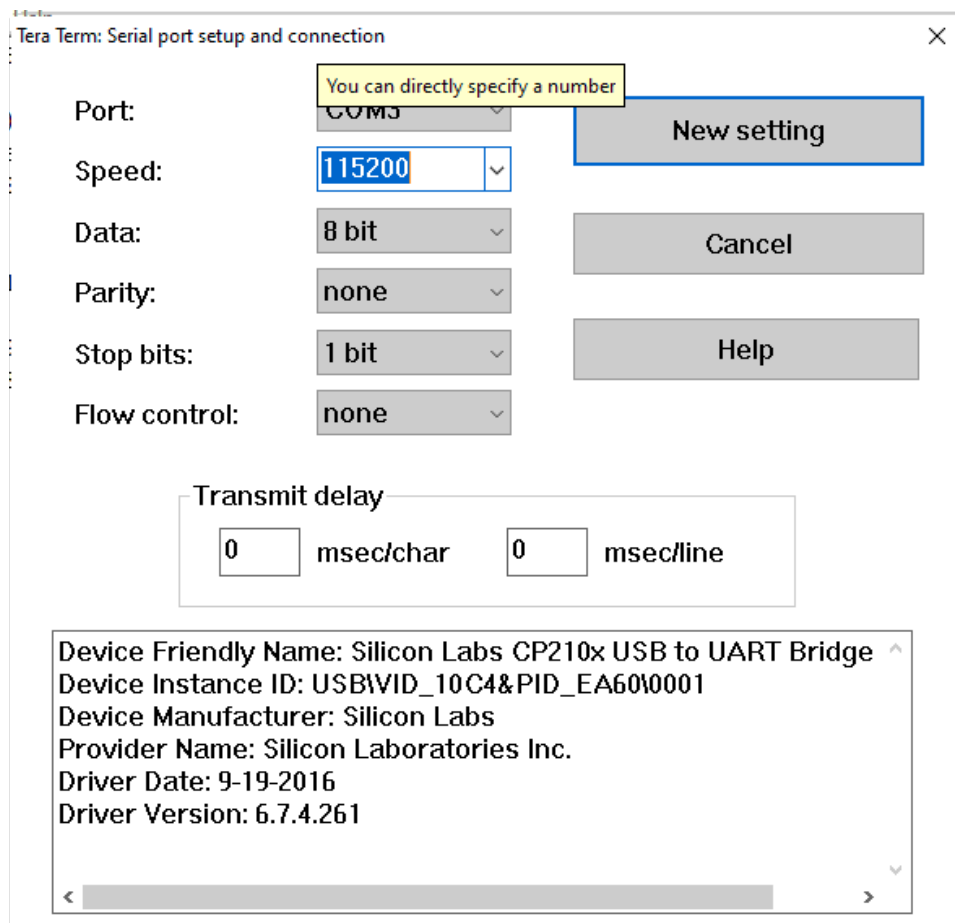


Instale Tera Term. La instalación es rápida y fácil.

Configuración de Tera Term

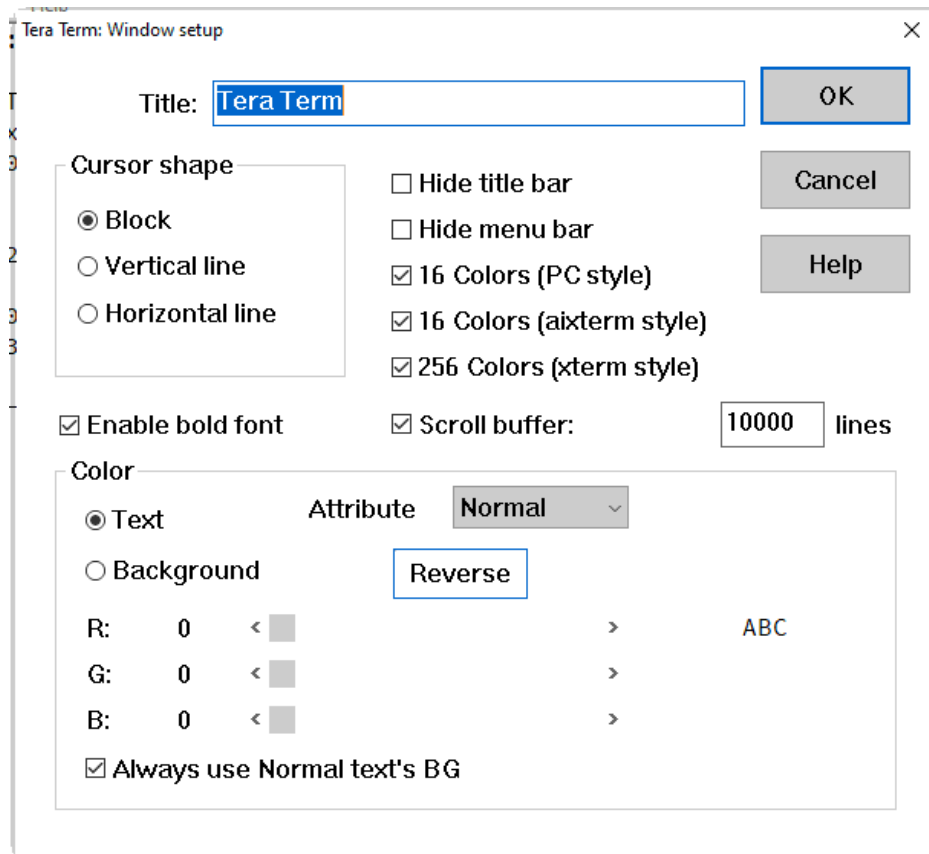
Para comunicarse con la tarjeta ESP32, debes ajustar ciertos parámetros:

- haga clic en Configuración -> puerto serie



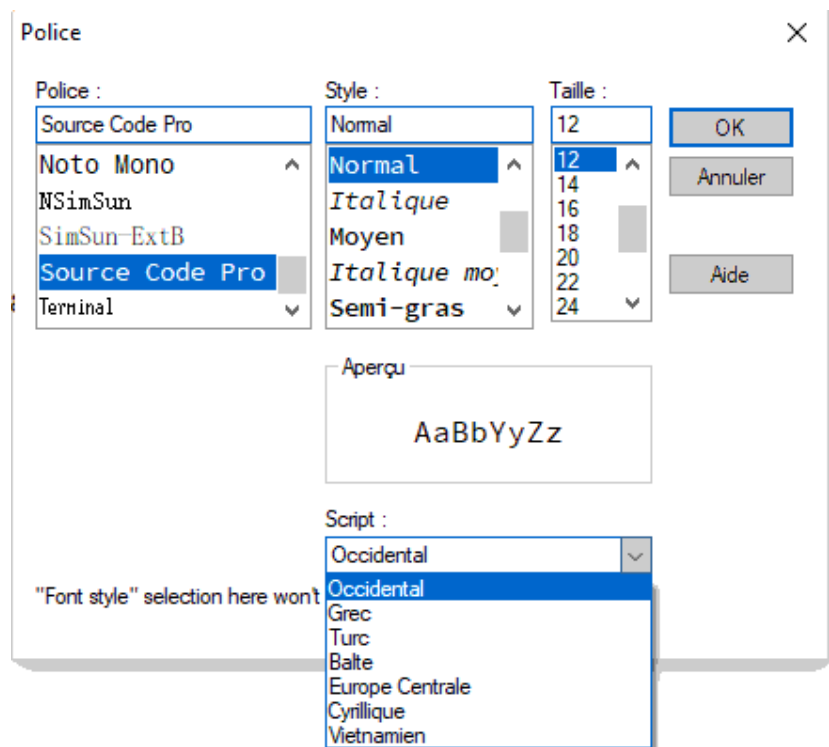
Para una visualización cómoda:

- haga clic en Configuración -> ventana



Para caracteres legibles:

- haga clic en Configuración -> fuente



Para encontrar todas estas configuraciones la próxima vez que inicie el terminal Tera Term, guarde la configuración:

- haga clic en *Configuración -> Guardar configuración*
- acepte el nombre **TERATERM.INI** .

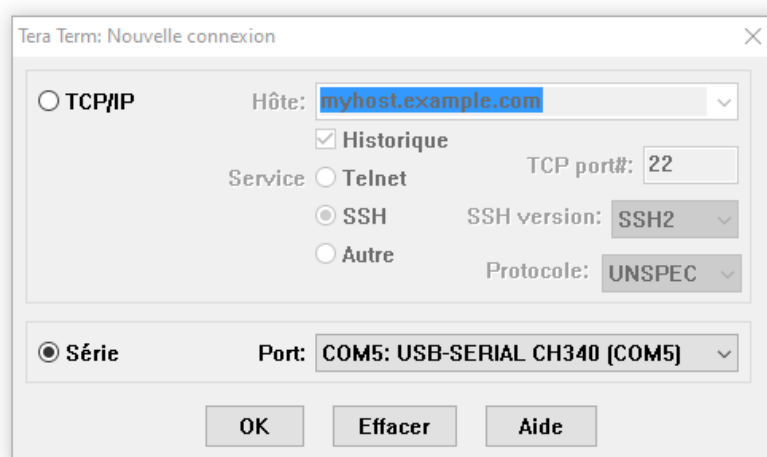
Usando el término Tera

Una vez configurado, cierre Tera Term.

Conecte su placa ESP32 a un puerto USB disponible en su PC.

Reinicie Tera Term, luego haga clic en *archivo -> nueva conexión*

Seleccione el puerto serie :



Si todo ha ido bien deberías ver esto:



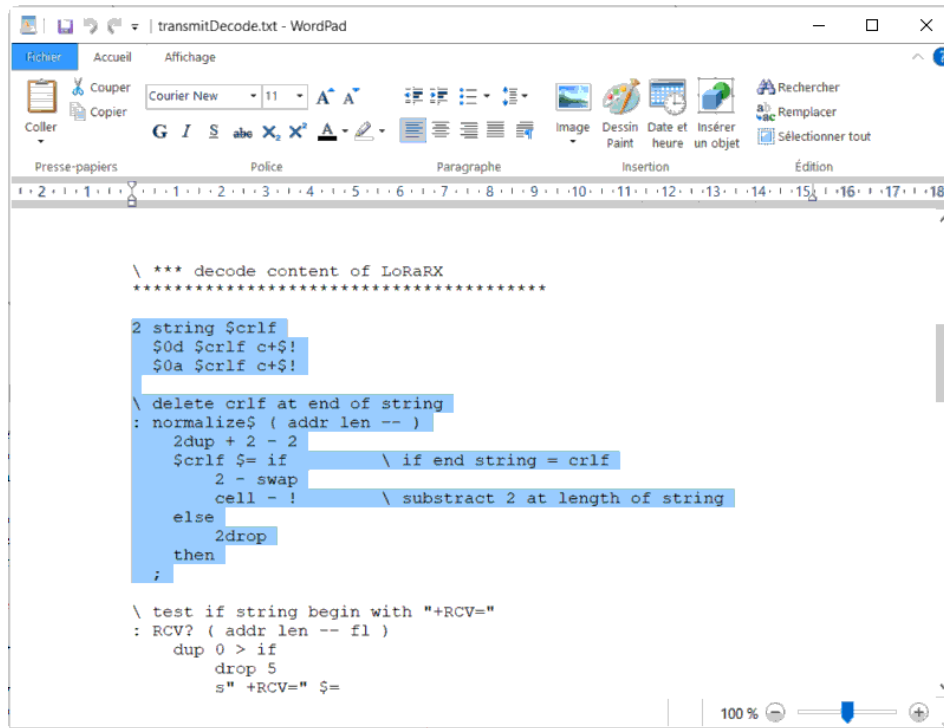
```
COM3 - Tera Term VT
File Edit Setup Control Window Help
ESP32forth v7.0.6.10 - rev 17c8b34289028a5c731d
ok
-->
```

Compilar código fuente en lenguaje Forth

En primer lugar, irecordemos que el idioma ADELANTE está en la placa ESP32! FORTH no está en tu PC. Por lo tanto, no puede compilar el código fuente de un programa en el lenguaje FORTH en la PC.

Para compilar un programa en lenguaje FORTH, primero debe abrir un archivo fuente en la PC con el editor de su elección.

Luego, copiamos el código fuente para compilar. Aquí, abre el código fuente con Wordpad:



El código fuente en lenguaje FORTH se puede componer y editar con cualquier editor de texto: notepad, PSpad, Wordpad..

Personalmente uso el IDE de Netbeans. Este IDE le permite editar y administrar códigos fuente en muchos lenguajes de programación.

Seleccione el código fuente o la porción de código que le interese. Luego haga clic en copiar. El código seleccionado está en el búfer de edición de la PC.

Haga clic en la ventana del terminal Tera Term. Hacer pasta:

Simplemente valide haciendo clic en Aceptar y el código será interpretado y/o compilado.

Para ejecutar el código compilado, simplemente escriba la palabra ADELANTE para iniciar, desde la terminal de Tera Term.

Acceder a ESP32Forth por TELNET

Antes de gestionar una conexión, debe establecer un enlace de red. La placa ESP32 tiene una interfaz WiFi. Para establecer una conexión WiFi, debes:

- tener un módem/enrutador que administre las conexiones WiFi
- tener el SSID del puerto WiFi disponible y su clave de acceso

La conexión a la red WiFi está garantizada por la palabra **login** :

```
\ conexión a LAN WiFi local
: myWiFiConnect ( -- )
  z" Mariloo"
  z" 1925144D91DE5373C3XXXXXXXX"
  login
;
```

Al ejecutar **myWiFiConnect** se muestra:

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

Cambiar el nombre DNS de la placa ESP32

Para conectarse a una placa ESP32, existen dos métodos:

- conociendo su dirección IP en la red interna. En el caso anterior, la dirección IP es 192.168.1.8. Esta dirección puede cambiar si no está bloqueada por el enrutador WiFi;
- por el nombre DNS declarado al conectarse a la red WiFi. Por defecto, ESP32forth asigna el nombre **a** la tarjeta que se conecta a la red WiFi.

cuarto nombre de host en lugar de la dirección IP:

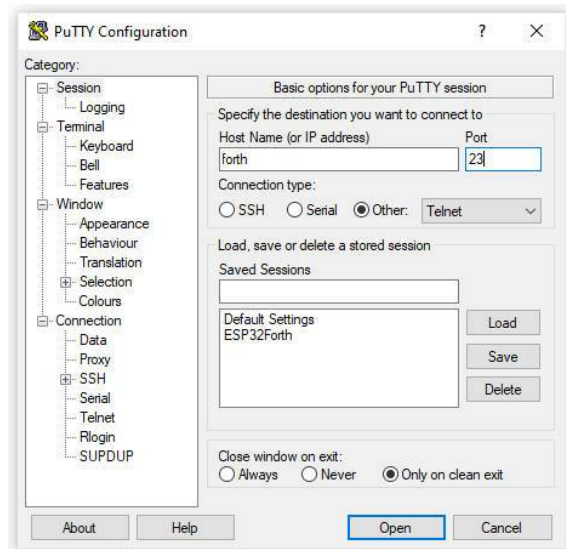


Figure 7: utilizar el nom DNS avec PuTTY

Si desea comunicarse con varias tarjetas ESP32 en la misma red, cada tarjeta debe declarar un nombre de host distinto. Código de ejemplo para dos tarjetas ESP32:

```
\ establece COM3 para la primera tarjeta ESP32
z" Mariloo"
z" 1925144D91DE5373C3C2D7XXXX"
login
z" forthCOM3" MDNS.begin
cr telnetd 552 server
```

Código para la segunda tarjeta ESP32 :

```
\ estableceCOM6 para la segunda tarjeta ESP32
z" Mariloo"
z" 1925144D91DE5373C3C2D7959F"
login
z" forthCOM6" MDNS.begin
cr telnetd 552 server
```

nombres de host de forthCOM3 y forthCOM6 en la red interna.

Conexión a placas ESP32 por su nombre de host

Inicie PuTTY. Ingresamos el nombre del host y el puerto abierto para acceder a COM3 :

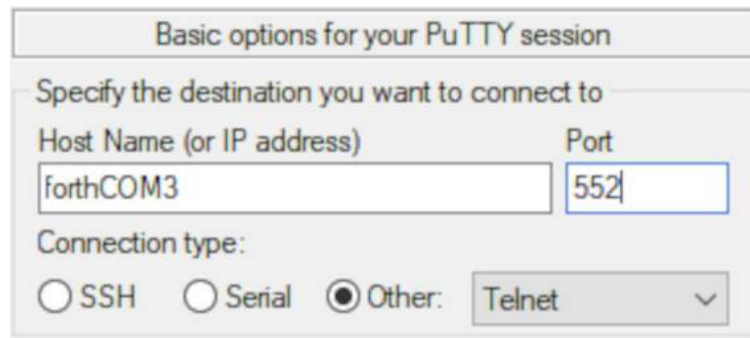


Figure 8: acceso de PuTTY a forthCOM3

Luego iniciamos una nueva sesión de PuTTY y simplemente cambiamos el nombre del host para esta sesión, de aquí en **forthCOM6**. Aquí hay dos sesiones de PuTTY que le permiten comunicarse con estas dos tarjetas ESP32:

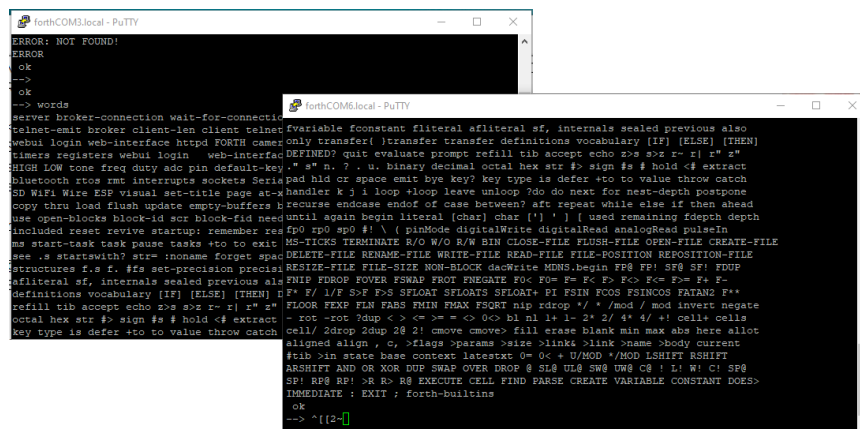


Figure 9: PuTTY accede a dos placas ESP32 separadas

Para iniciar automáticamente el cliente TELNET en la tarjeta ESP32, integraremos nuestro código de conexión en **autoexec.fs**. Aquí está el código para escribir desde la terminal. Primer tipo:

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

Luego ingrese estas pocas líneas:

```
\ establece forthCOM3 para la primera tarjeta ESP32
z" Mariloo"
z" 1925144D91DE5373C3C2DXXXXX"
login
z" forthCOM3" MDNS.begin
cr telnetd 552 server
forth
```

Luego haga CTRL-X e Y. El código se guarda y se cargará la próxima vez que inicie ESP32. El cliente TELNET se reiniciará automáticamente cuando se inicie ESP32forth. Ya no es necesario utilizar el terminal para comunicarse con la tarjeta ESP32 declarada con el nombre de host en **COM3**:

- desenchufe la placa ESP32;
- Vuelva a conectar la placa ESP32, ipero no abra el terminal!
- espera unos segundos...
- Inicie PuTTY y active una conexión TELNET con **ForthCOM3** en el puerto 552.

El acceso TELNET vía PuTTY permite las mismas operaciones que a través del terminal. Única restricción: si transmite el código FORTH copiando/pegando, limite el tamaño del código transmitido.

NOTA: Se puede acceder a las tarjetas ESP32 configuradas de esta forma desde Internet si la configuración del router WiFi lo permite.

Gestión de proyectos RECORDFILE y FORTH

Este capítulo está dedicado a un único elemento clave: **RECORDFILE** . Esta palabra permite guardar rápidamente archivos en el sistema de archivos SPIFFS.

Le aconsejo que lo lea atentamente antes de intentar manipular archivos fuente con un **visual** o **editor** .

A continuación se explica paso a paso cómo guardar la definición de **RECORDFILE** y luego utilizarla de forma eficaz.

Guarde RECORDFILE en el archivo autoexec.fs

Cuando se inicia ESP32forth, el sistema prueba la presencia del archivo **autoexec.fs** . Si este archivo está presente, se interpretará su contenido.

Aquí está el código fuente de **RECORDFILE** . Esta definición fue desarrollada por Bob EDWARDS. Copie este código y péguelo en la ventana de la terminal para compilarlo. Esta maniobra sólo habrá que realizarla una vez:

```
\ 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                                \ repeat until <EOF> found
R> CLOSE-FILE throw                 \ Close the file
;

```

Una vez compilada esta palabra, veremos cómo proceder para que esta palabra esté disponible permanentemente desde **autoexec.fs** .

En su PC, en su área de desarrollo dedicada a ESP32Forth, cree un archivo **autoexec.fs** .

Copie el código **RECORDFILE** como se indica arriba en este archivo **autoexec.fs** .

Agregue estas dos líneas de código:

```

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                                \ repeat until <EOF> found
  R> CLOSE-FILE throw                 \ Close the file
;
<EOF>

```

Copie este código fuente nuevamente, incluidas las líneas de código en rojo. Pegue este código en la ventana de la terminal nuevamente. Transmita este código a la placa ESP32.

A diferencia de la primera manipulación que consiste en compilar el código, en esta ocasión este código se guarda en el archivo **/spiffs/autoexec.fs** .

archivo **autoexec.fs** esté guardado, ejecute **ls** :

```
ls /spiffs/
```

archivo **autoexec.fs** debería aparecer en la lista de archivos. Para verificar el contenido de **autoexec.fs** , escriba:

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

Esto debería mostrar el contenido de **autoexec.fs** .

Utilice contenidos modificados del archivo autoexec.fs

Reinicie ESP32 en adelante. Si todo salió bien, **RECORDFILE** ahora estará disponible cuando se inicie ESP32forth. Ejecutar **words**. Deberías encontrar **RECORDFILE** en las primeras palabras del FORTH diccionario:

```
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...
```

No llene **autoexec.fs** con otras definiciones. Veremos cómo crear un proyecto.

Desglosando un proyecto con ESP32forth

Se crea un proyecto de desarrollo FORTH para ESP32forth en su PC:

- editar el código fuente con el editor de texto de su elección o un IDE (Netbeans por ejemplo);
- tener un terminal vinculado por USB a la tarjeta ESP32;
- tenga ESP32 adelante habilitado en la placa ESP32.

En el PC, trabaje de forma estructurada. Las siguientes explicaciones son sólo recomendaciones.

Comience por definir el directorio de trabajo general para todos los desarrollos de ESP32forth. Por ejemplo, una carpeta llamada **ESP32forth developments** .

Luego, en esta carpeta, cree dos carpetas adicionales:

- **_my Projects** que está destinado a dar cabida a todos sus proyectos;
- **_sandbox** destinado a recibir todos los pequeños programas que se van a probar y que no tienen un uso específico;
- **Tools** destinadas a dar cabida a todos los archivos fuente de interés general. Estos son archivos probados y no requieren adaptación;
- **Documentación** que está destinada a documentos de cualquier tipo.

Proyecto de ejemplo

Usaré el código fuente TEMPVS FVGIT como proyecto de ejemplo. Los códigos fuente completos están disponibles aquí:

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

El primer archivo a crear se llama **main.fs**. Este archivo debe estar escrito en una carpeta TEMPVS FVGIT:

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

Una vez más, estas son sólo recomendaciones. El interés principal es reunir todos los componentes de un solo proyecto. Contenido del archivo **main.fs** :

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

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

s" /spiffs/clepsydra.fs"    included

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

En rojo encontramos nuestra palabra **RECORDFILE** . Para guardar el código de **main.fs** en el sistema de archivos SPIFFS en la placa ESP32, simplemente copie este código fuente y páselo a ESP32 junto con el programa terminal.

En azul, en el código anterior, el contenido de **main.fs** realiza una llamada al archivo **strings.fs** . El código fuente de este archivo proviene de la carpeta **Tools** . Es una copia de **strings.fs** que luego se modifica así:

```
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>

```

Copiar y pasar este código fuente crea el archivo **strings.fs** en el sistema de archivos SPIFFS en la placa ESP32.

En esta etapa, empezamos a tener varios archivos en la tarjeta ESP32. Para compilar todos los archivos transferidos simplemente ejecutaremos:

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

No hay límite para los archivos que se pueden guardar en la tarjeta ESP32, aparte del límite de espacio físico. El espacio disponible en el sistema de archivos SPIFFS supera 1 MB de espacio de grabación.

Si necesita modificar el contenido de un componente de software de propósito general, hágalo siempre en una copia del archivo fuente de ese componente. Recuerde versionar y fechar estas modificaciones.

Para cada uno de los archivos de este proyecto, integramos **RECORDFILE** y su terminador **<EOF>** .

En cada proyecto encontramos los archivos **main.fs** y **config.fs** . Pero su contenido se adapta a cada proyecto. Para un proyecto específico, todos los archivos de extensión **fs** se cargan en la placa ESP32 en el sistema de archivos SPIFFS. Recopilar su contenido es increíblemente rápido. Pero sobre todo, el contenido de estos archivos **se conserva** entre dos reinicios de la tarjeta ESP32. Al menor bloqueo de FORTH, es fácil reiniciar la tarjeta y

La noción de caja negra

En CUARTO idioma, una palabra tiene una definición. Cuando se pasan parámetros a través de la pila, al final, solo el diseñador de la definición debe garantizar el correcto funcionamiento de la definición. Y para garantizar el correcto funcionamiento de una definición, se recomienda encarecidamente no formular definiciones demasiado largas.

```

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

```

```
\ multiplicar fn por pi
: fpi* ( f
      pi f*
      :
```

```

: fpi* (
      pi f*
;

```

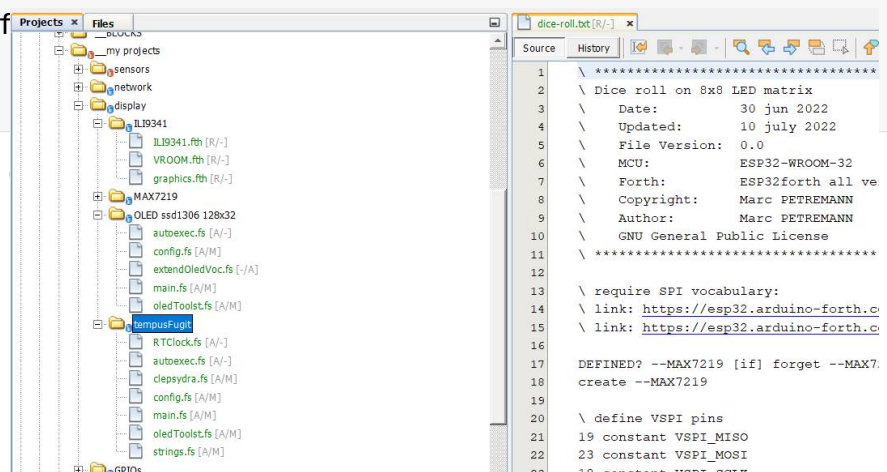


Figure 10: Estructuración de proyectos con el IDE de Netbeans

Este archivo está destinado únicamente a la realización de pruebas unitarias de baterías. Consulte la definición de la palabra **assert**(que realiza estas pruebas, definición visible aquí:

<https://github.com/MPETREMAN11/ESP32forth/blob/main/tools/assert.fs>

Y aquí hay un ejemplo de pruebas guardadas en nuestro archivo **tests.fs** :

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

assert(genera una alerta si la palabra probada no se comporta como se esperaba.

Para una definición tan simple como la de **fpi*** , puede ser necesaria una batería de pruebas si no hemos hecho que la palabra **f*** sea confiable. Integramos estas pruebas en el archivo **main.fs** :

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

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

s" /spiffs/clepsydra.fs"    included

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

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

archivo **tests.fs** también debe transferirse a la tarjeta ESP32.

De esta forma, el ciclo completo de compilación incluye también una batería de pruebas. Las pruebas no garantizan que el código sea confiable. Sólo permiten detectar posibles efectos secundarios si tenemos que modificar partes del código de la aplicación.

En resumen, recomiendo fragmentar su código integrando sistemáticamente estos archivos:

- **main.fs** que es el archivo principal. Normalmente, sea cual sea el nombre del proyecto, lo compilarás con una simple ejecución de **include /spiffs/main.fs**.

- **config.fs** que contiene los parámetros de configuración global, contraseñas de acceso WiFi por ejemplo;
- **tests.fs** que contiene una batería de pruebas. Si no está realizando ninguna prueba, no es necesario crear este archivo.

Todos los demás archivos tendrán la extensión **fs** , excepto los archivos no procesados por ESP32forth.

Si sigue estas pocas pautas, le resultará más fácil administrar aplicaciones complejas. Ahorrar tiempo cada vez que se compilan y guardan partes confiables de código en archivos en el sistema de archivos SPIFFS es el argumento principal para adoptar

RECORDFILE .

Instalación de la biblioteca OLED para SSD1306

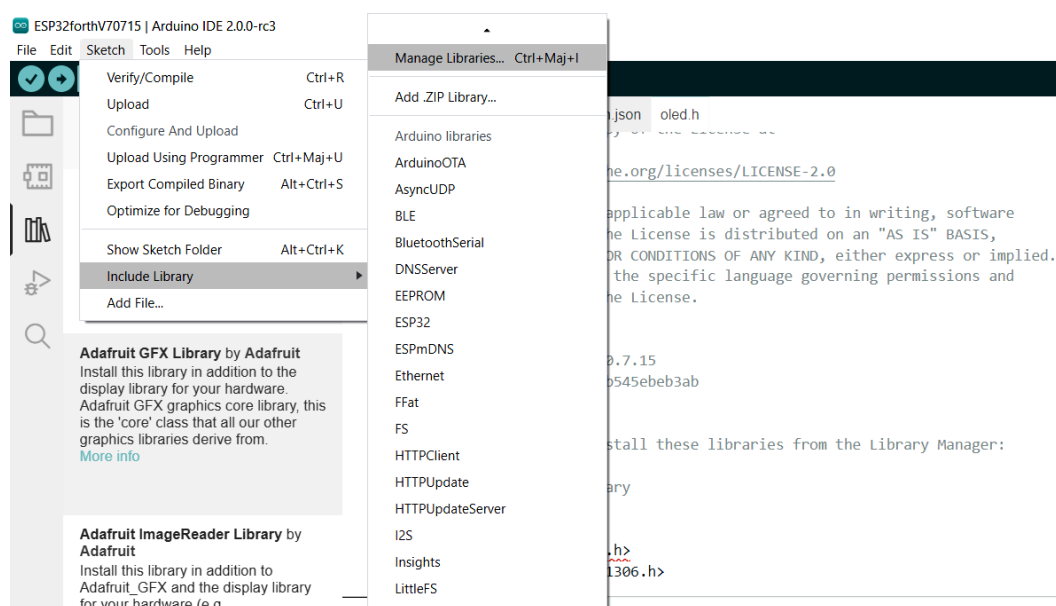
Desde ESP32 en adelante versión 7.0.7.15, las opciones están disponibles en la carpeta **optional**:

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

Para tener el vocabulario **oled**, copie el archivo **oled.h** a la carpeta que contiene el archivo **ESP32forth.ino**.

Luego inicie ARDUINO IDE y seleccione el archivo **ESP32forth.ino** más reciente.

Si la biblioteca OLED no se ha instalado, en ARDUINO IDE, haga clic en *Sketch* y seleccione *Include*, luego seleccione *Manage Libraries*.



En la barra lateral izquierda, busque la biblioteca **Adafruit SSD1306 by Adafruit**.

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

Ahora puede comenzar a compilar el boceto haciendo clic en *Sketch* y seleccionando *Upload*.

Una vez que el boceto esté cargado en la placa ESP32, inicie la terminal TeraTerm.
Compruebe que el vocabulario **oled** esté presente:

```
oled vlist \ display:
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
```

Instalación del cliente HTTP

Editando el archivo ESP32forth.ino

ESP32Forth se proporciona como un archivo fuente, escrito en lenguaje C. Este archivo debe compilarse usando ARDUINO IDE o cualquier otro compilador de C compatible con el entorno de desarrollo ARDUINO.

Aquí están las partes del código que se deben modificar. Primera parte a modificar:

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

Segunda parte a modificar:

```
// .....
#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(blueetooth) V(sockets) V(oled) \
  V(rmt) V(interrupts) V(spi_flash) V(camera) V(timers)
```

Tercera parte a modificar:

```
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
```

Cuarta parte a modificar:

```
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
```

Una vez modificado el archivo **ESP32forth.ino** , lo compilas y lo subes a la placa ESP32. Si todo salió correctamente, deberías tener un nuevo vocabulario **http** :

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

Prueba de cliente HTTP

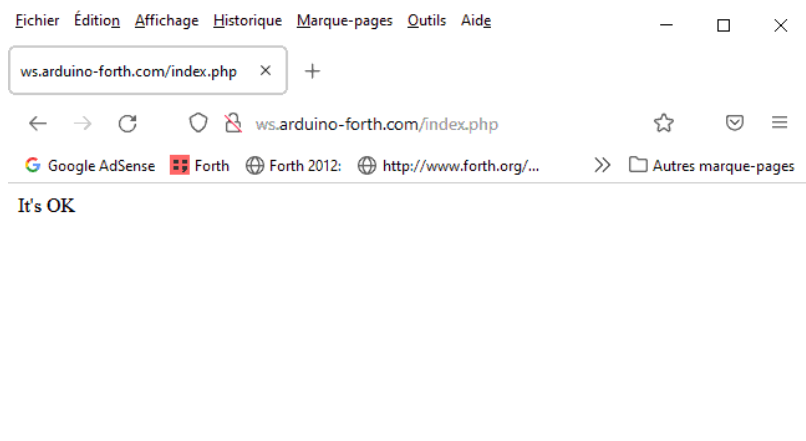
Para probar nuestro cliente HTTP, podemos hacerlo consultando cualquier servidor web. Pero para lo que veremos más adelante, es necesario tener un servidor web personal. En este servidor, creamos un subdominio:

- nuestro servidor es arduino-forth.com
- subdominio **ws**
- accedemos a este subdominio con la URL <http://ws.arduino-forth.com>

Al crearse este subdominio, no contiene ningún script para ejecutar. Creamos la página **index.php** y ponemos este código allí:

```
It's OK
```

Para comprobar que nuestro subdominio es funcional basta con consultarlo desde nuestro navegador web favorito:



Si todo va según lo planeado, deberíamos mostrar el texto **It's OK** en nuestro navegador web favorito. Veamos ahora cómo realizar esta misma consulta al servidor desde ESP32Forth...

Aquí está el código FORTH escrito rápidamente para realizar la prueba del cliente HTTP:

```
WiFi

\ connection to local WiFi LAN
: myWiFiConnect
  z" mySSID"
  z" myWiFiCode"
  login
;

Forth

create httpBuffer 700 allot
  httpBuffer 700 erase

HTTP

: run
  cr
  z" http://ws.arduino-forth.com/" HTTP.begin
  if
    HTTP.doGet dup ." Get results: " . cr 0 >
    if
      httpBuffer 700 HTTP.getPayload
      httpBuffer z>s dup . cr type
    then
  then
  HTTP.end
;
```

Activamos la conexión Wifi ejecutando **myWiFiConnect** luego **run** :

```
--> myWiFiConnect
192.168.1.23
MDNS started
```

```
ok
--> run

Get results: 200
8
It's OK
ok
```

Nuestro cliente HTTP consultó perfectamente el servidor web y mostró el mismo texto que el recuperado de nuestro navegador web.

Esta pequeña prueba exitosa abre el camino a enormes posibilidades.

El generador de números aleatorios

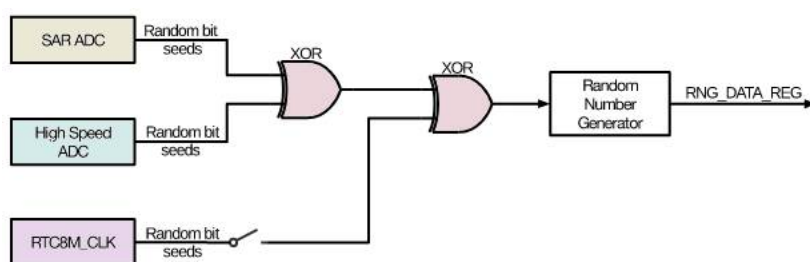
Característica

El generador de números aleatorios genera números aleatorios verdaderos, lo que significa un número aleatorio generado a partir de un proceso físico, en lugar de mediante un algoritmo. Ningún número generado dentro del rango especificado tiene más o menos probabilidad de aparecer que cualquier otro número.

Cada valor de 32 bits que el sistema lee del registro RNG_DATA_REG del generador de números aleatorios es un número aleatorio verdadero. Estos números aleatorios verdaderos se generan en función del ruido térmico en el sistema y del desfase del reloj asíncrono.

El ruido térmico proviene del ADC de alta velocidad, del SAR ADC o de ambos. Siempre que se active el ADC de alta velocidad o el ADC SAR, los flujos de bits se generarán y se introducirán en el generador de números aleatorios a través de una puerta lógica XOR como semillas aleatorias.

Cuando el reloj RTC8M_CLK está habilitado para el núcleo digital, el generador de números aleatorios también tomará muestras de RTC8M_CLK (8 MHz) como una semilla binaria aleatoria. RTC8M_CLK es una fuente de reloj asíncrona y aumenta la entropía del RNG al introducir la metaestabilidad del circuito. Sin embargo, para garantizar la máxima entropía, también se recomienda habilitar siempre una fuente ADC.



Cuando hay ruido del SAR ADC, el generador de números aleatorios se alimenta con una entropía de 2 bits en un ciclo de reloj de RTC8M_CLK (8 MHz), que se genera a partir de un oscilador RC interno (consulte el capítulo Restablecer y reloj para obtener más detalles). Por tanto, es recomendable leer el registro **RNG_DATA_REG** a una velocidad máxima de 500 kHz para obtener la máxima entropía.

Cuando hay ruido del ADC de alta velocidad, el generador de números aleatorios recibe entropía de 2 bits en un ciclo de reloj APB, que normalmente es de 80 MHz. Por tanto, es recomendable leer el registro **RNG_DATA_REG** a una velocidad máxima de 5 MHz para obtener la máxima entropía.

Se probó una muestra de datos de 2 GB, que se lee del generador de números aleatorios a una frecuencia de 5 MHz con solo el ADC de alta velocidad habilitado, utilizando el conjunto de pruebas Dieharder Random Number (versión 3.31.1). La muestra pasó todas las pruebas.

Procedimiento de programación

Cuando utilice el generador de números aleatorios, asegúrese de que se permita al menos SAR ADC, High Speed ADC o RTC8M_CLK. De lo contrario, se devolverán números pseudoaleatorios.

- SAR ADC se puede activar utilizando el controlador DIG ADC.
- El ADC de alta velocidad se habilita automáticamente cuando los módulos Wi-Fi o Bluetooth están habilitados.
- RTC8M_CLK se habilita configurando el bit RTC_CNTL_DIG_CLK8M_EN en el registro RTC_CNTL_CLK_CONF_REG.

Cuando utilice el generador de números aleatorios, lea el registro **RNG_DATA_REG** varias veces hasta que se generen suficientes números aleatorios.

Name	Description	Address	Access
RNG_DATA_REG	Random number data	\$3FF75144	RO

```
\ Datos de números aleatorios
$3FF75144 constant RNG_DATA_REG

\ obtener 32 bits aleatorio b=número
: rnd ( -- x )
  RNG_DATA_REG L@
  ;

\ obtener número aleatorio en el intervalo [0..n-1]
: random ( n -- 0..n-1 )
  rnd swap mod
  ;
```

Función RND en ensamblador XTENSA

Desde la versión 7.0.7.4, ESP32 en adelante tiene un ensamblador XTENSA. Es posible reescribir nuestra **rnd** palabra en el ensamblador XTENSA:

```
forth definitions
asm xtensa
$3FF75144 constant RNG_DATA_REG

code myRND ( -- [addr] )
  a1 32          ENTRY,
```



```
a8 RNG_DATA_REG L32R,      \ a8 = RNG_DATA_REG
a9 a8 0          L32I.N,    \ a9 = [a8]
a9              arPUSH,     \ push a9 on stack
                  RETW.N,
end-code
```

Contenido detallado de los vocabularios ESP32forth

ESP32forth proporciona numerosos vocabularios:

- **FORTH** es el vocabulario principal;
- Ciertos vocabularios se utilizan para la mecánica interna de ESP32Forth, como **internals** , **asm...**
- Muchos vocabularios permiten la gestión de puertos o accesorios específicos, como **bluetooth** , **oled** , **spi** , **wifi** , **wire...**

Aquí encontrarás la lista de todas las palabras definidas en estos diferentes vocabularios. Algunas palabras se presentan con un enlace de color:

[align](#) es una FORTH palabra ordinaria;

CONSTANT es palabra de definición;

begin marca una estructura de control;

key es una palabra de ejecución diferida;

LED es una palabra definida por **constant** , **variable** o **value** ;

registers marca un vocabulario.

Las palabras del vocabulario **FORTH** se muestran en orden alfabético. Para otros vocabularios, las palabras se presentan en su orden de visualización.

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>	rl
r~	rdrop	read-dir	READ-FILE	recurse	refill	registers
remaining	remember	RENAME-FILE	repeat	REPOSITION-FILE		required
reset	resize	RESIZE-FILE	restore	revive	RISC-V?	rm
rot	RP!	RP@	rp0	RSHIFT	rtos	s"
S>F	s>z	save	save-buffers		scr	SD
SD_MMC	sealed	see	Serial	set-precision		set-title
sf,	SF!	SF@	SFLOAT	SFLOAT+	SFLOATS	sign
SL@	sockets	SP!	SP@	sp0	space	spaces
SPIFFS	start-task	startswith?	startup:	state	str	str=
streams	structures	SW@	SWAP	task	tasks	telnetd
terminate	then	throw	thru	tib	to	tone
touch	transfer	transfer	type	u.	U/MOD	UL@
UNLOOP	until	update	use	used	UW@	value
VARIABLE	visual	vlist	vocabulary	W!	W/O	web-
interface						
webui	while	WiFi	Wire	words	WRITE-FILE	XOR
Xtensa?	z"	z>s				

asm

```

xtensa disasm 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?  
fill132 '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_RTDRAM 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](#) [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.beginTransmission Wire.endTransmission Wire.requestFrom Wire.write  
Wire.available Wire.read Wire.peek Wire.flush Wire-builtins
```

xtensa

```
WUR, WSR, WITLB, WER, WDTLB, WAITI, SSXU, SSX, SSR, SSL, SSIU, SSI, SSAI,  
SSA8L, SSA8B, SRLI, SRL, SRC, SRAI, SRA, SLLI, SLL, SICW, SICT, SEXT, SDCT,  
RUR, RSR, RSIL, RFI, ROTW, RITLB1, RITLB0, RER, 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 MULA.AD.HH, MULA.AD.LH, MULA.AD.HL, MUL.AD.LL, MUL.AD MULA.AA.HH, MULA.AA.LH, MULA.AA.HL, MULA.AA.LL, MULA.AA [MOV_T](#), MOVSP, MOV_T.S, MOV_F.S, MOVGEZ.S, MOVLTZ.S, MOVNEZ.S, MOVEQZ.S, ULE.S, OLE.S, ULT.S, OLT.S, UEQ.S, OEQ.S, UN.S, CMPSOP NEG.S, WFR, RFR, [ABS.S](#), MOV.S, ALU2.S UTRUNC.S, UFLOAT.S, FLOAT.S, CEIL.S, FLOOR.S, TRUNC.S, [ROUND.S](#), MSUB.S, MADD.S, MUL.S, SUB.S, [ADD.S](#), ALU.S [MOV_F](#), MOVGEZ, MOVLTZ, MOVNEZ, MOVEQZ, [MAX_U](#), [MIN_U](#), [MAX](#), [MIN](#), CONDOP [MOV](#), LSXU, LSX, L32E, LICW, LICT, LDCT, [JX](#), IITLB, [IDTLB](#), LSIU, LSI, LDINC, LDDEC, [L32R](#), EXTUI, S32E, S32RI, S32CI, [ADD_{MI}](#), [ADD_I](#), L32AI, L16SI, S32I, S16I, S8I, L32I, L16UI, L8UI, LDSTORE [MOV_I](#), IIU, IHU, IPFL, DIWBI, DIWB, DIU, DHU, DPFL, CACHING2 III, IHI, IPF, DII, DHI, DHWBI, DHWB, DPFWO, DPFR, DPFW, 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 cofcs cofcs. >cofs offset18 offset12 offset8 ofs18 ofs12 ofs8 ofs18. ofs12. ofs8. >ofs sr imm16 imm8 imm4 im numeric register reg. nop [a15](#) [a14](#) [a13](#) [a12](#) [a11](#) [a10](#) [a9](#) [a8](#) [a7](#) [a6](#) [a5](#) [a4](#) [a3](#) [a2](#) [a1](#) [a0](#)

Annexe A – Sommaire des registres

.....

GPIO registers

Name	Description	Address	Access
GPIO_OUT_REG	GPIO 0-31 output register	\$3FF44004	R/W
GPIO_OUT_W1TS_REG	GPIO 0-31 output register_W1TS	\$3FF44008	WO
GPIO_OUT_W1TC_REG	GPIO 0-31 output register_W1TC	\$3FF4400C	WO
GPIO_OUT1_REG GPIO	GPIO 32-39 output register	\$3FF44010	R/W
GPIO_OUT1_W1TS_REG	GPIO 32-39 output bit set register	\$3FF44014	WO
GPIO_OUT1_W1TC_REG	GPIO 32-39 output bit clear register	\$3FF44018	WO
GPIO_ENABLE_REG	GPIO 0-31 output enable register	\$3FF44020	R/W
GPIO_ENABLE_W1TS_REG	GPIO 0-31 output enable register_W1TS	\$3FF44024	WO
GPIO_ENABLE_W1TC_REG	GPIO 0-31 output enable register_W1TC	\$3FF44028	WO
GPIO_ENABLE1_REG	GPIO 32-39 output enable register	\$3FF4402C	R/W
GPIO_ENABLE1_W1TS_REG	GPIO 32-39 output enable bit set register	\$3FF44030	WO
GPIO_ENABLE1_W1TC_REG	GPIO 32-39 output enable bit clear register	\$3FF44034	WO
GPIO_STRAP_REG	Bootstrap pin value register	\$3FF44038	RO

Name	Description	Address	Access
GPIO_IN_REG	GPIO 0-31 input register	\$3FF4403C	RO
GPIO_IN1_REG	GPIO 32-39 input register	\$3FF44040	RO
GPIO_STATUS_REG	GPIO 0-31 interrupt status register	\$3FF44044	R/W
GPIO_STATUS_W1TS_REG	GPIO 0-31 interrupt status register_W1TS	\$3FF44048	WO
GPIO_STATUS_W1TC_REG	GPIO 0-31 interrupt status register_W1TC	\$3FF4404C	WO
GPIO_STATUS1_REG	GPIO 32-39 interrupt status register1	\$3FF44050	R/W
GPIO_STATUS1_W1TS_REG	GPIO 32-39 interrupt status bit set register	\$3FF44054	WO
GPIO_STATUS1_W1TC_REG	GPIO 32-39 interrupt status bit clear register	\$3FF44058	WO
GPIO_ACPU_INT_REG	GPIO 0-31 APP_CPU interrupt status	\$3FF44060	RO
GPIO_ACPU_NMI_INT_REG	GPIO 0-31 APP_CPU non-maskable interrupt status	\$3FF44064	RO
GPIO_PCPU_INT_REG	GPIO 0-31 PRO_CPU interrupt status	\$3FF44068	RO
GPIO_PCPU_NMI_INT_REG	GPIO 0-31 PRO_CPU non-maskable interrupt status	\$3FF4406C	RO
GPIO_ACPU_INT1_REG	GPIO 32-39 APP_CPU interrupt status	\$3FF44074	RO
GPIO_ACPU_NMI_INT1_REG	GPIO 32-39 APP_CPU non-maskable interrupt status	\$3FF44078	RO
GPIO_PCPU_INT1_REG	GPIO 32-39 PRO_CPU interrupt status	\$3FF4407C	RO
GPIO_PCPU_NMI_INT1_REG	GPIO 32-39 PRO_CPU non-maskable interrupt status	\$3FF44080	RO
GPIO_PIN0_REG	Configuration for GPIO pin 0	\$3FF44088	R/W
GPIO_PIN1_REG	Configuration for GPIO pin 1	\$3FF4408C	R/W
GPIO_PIN2_REG	Configuration for GPIO pin 2	\$3FF44090	R/W
GPIO_PIN38_REG	Configuration for GPIO pin 38	\$3FF44120	R/W
GPIO_PIN39_REG	Configuration for GPIO pin 39	\$3FF44124	R/W
GPIO_FUNC0_IN_SEL_CFG_REG	Peripheral function 0 input selection register	\$3FF44130	R/W
GPIO_FUNC1_IN_SEL_CFG_REG	Peripheral function 1 input selection register	\$3FF44134	R/W
GPIO_FUNC254_IN_SEL_CFG_REG	Peripheral function 254 input selection register	\$3FF44528	R/W
GPIO_FUNC255_IN_SEL_CFG_REG	Peripheral function 255 input selection register	\$3FF4452C	R/W
GPIO_FUNC0_OUT_SEL_CFG_REG	Peripheral output selection for GPIO 0	\$3FF44530	R/W
GPIO_FUNC1_OUT_SEL_CFG_REG	Peripheral output selection for GPIO 1	\$3FF44534	R/W
GPIO_FUNC38_OUT_SEL_CFG_REG	Peripheral output selection for GPIO 38	\$3FF445C8	R/W
GPIO_FUNC39_OUT_SEL_CFG_REG	Peripheral output selection for GPIO 39	\$3FF445CC	R/W
IO_MUX_PIN_CTRL	Clock output configuration register	\$3FF49000	R/W
IO_MUX_GPIO36_REG	Configuration register for pad GPIO36	\$3FF49004	R/W
IO_MUX_GPIO37_REG	Configuration register for pad GPIO37	\$3FF49008	R/W
IO_MUX_GPIO38_REG	Configuration register for pad GPIO38	\$3FF4900C	R/W
IO_MUX_GPIO39_REG	Configuration register for pad GPIO39	\$3FF49010	R/W
IO_MUX_GPIO34_REG	Configuration register for pad GPIO34	\$3FF49014	R/W
IO_MUX_GPIO35_REG	Configuration register for pad GPIO35	\$3FF49018	R/W
IO_MUX_GPIO32_REG	Configuration register for pad GPIO32	\$3FF4901C	R/W
IO_MUX_GPIO33_REG	Configuration register for pad GPIO33	\$3FF49020	R/W
IO_MUX_GPIO25_REG	Configuration register for pad GPIO25	\$3FF49024	R/W
IO_MUX_GPIO26_REG	Configuration register for pad GPIO26	\$3FF49028	R/W
IO_MUX_GPIO27_REG	Configuration register for pad GPIO27	\$3FF4902C	R/W
IO_MUX_MTMS_REG	Configuration register for pad MTMS	\$3FF49030	R/W
IO_MUX_MTDI_REG	Configuration register for pad MTDI	\$3FF49034	R/W
IO_MUX_MTCK_REG	Configuration register for pad MTCK	\$3FF49038	R/W
IO_MUX_MTDO_REG	Configuration register for pad MTDO	\$3FF4903C	R/W
IO_MUX_GPIO2_REG	Configuration register for pad GPIO2	\$3FF49040	R/W
IO_MUX_GPIO0_REG	Configuration register for pad GPIO0	\$3FF49044	R/W
IO_MUX_GPIO4_REG	Configuration register for pad GPIO4	\$3FF49048	R/W
IO_MUX_GPIO16_REG	Configuration register for pad GPIO16	\$3FF4904C	R/W
IO_MUX_GPIO17_REG	Configuration register for pad GPIO17	\$3FF49050	R/W
IO_MUX_SD_DATA2_REG	Configuration register for pad SD_DATA2	\$3FF49054	R/W

Name	Description	Address	Access
IO_MUX_SD_DATA3_REG	Configuration register for pad SD_DATA3	\$3FF49058	R/W
IO_MUX_SD_CMD_REG	Configuration register for pad SD_CMD	\$3FF4905C	R/W
IO_MUX_SD_CLK_REG	Configuration register for pad SD_CLK	\$3FF49060	R/W
IO_MUX_SD_DATA0_REG	Configuration register for pad SD_DATA0	\$3FF49064	R/W
IO_MUX_SD_DATA1_REG	Configuration register for pad SD_DATA1	\$3FF49068	R/W
IO_MUX_GPIO5_REG	Configuration register for pad GPIO5	\$3FF4906C	R/W
IO_MUX_GPIO18_REG	Configuration register for pad GPIO18	\$3FF49070	R/W
IO_MUX_GPIO19_REG	Configuration register for pad GPIO19	\$3FF49074	R/W
IO_MUX_GPIO20_REG	Configuration register for pad GPIO20	\$3FF49078	R/W
IO_MUX_GPIO21_REG	Configuration register for pad GPIO21	\$3FF4907C	R/W
IO_MUX_GPIO22_REG	Configuration register for pad GPIO22	\$3FF49080	R/W
IO_MUX_U0RXD_REG	Configuration register for pad U0RXD	\$3FF49084	R/W
IO_MUX_U0TXD_REG	Configuration register for pad U0TXD	\$3FF49088	R/W
IO_MUX_GPIO23_REG	Configuration register for pad GPIO23	\$3FF4908C	R/W
IO_MUX_GPIO24_REG	Configuration register for pad GPIO24	\$3FF49090	R/W
GPIO configuration / data registers			
RTCIO_RTC_GPIO_OUT_REG	RTC GPIO output register	0x3FF48400	R/W
RTCIO_RTC_GPIO_OUT_W1TS_REG	RTC GPIO output bit set register	0x3FF48404	WO
RTCIO_RTC_GPIO_OUT_W1TC_REG	RTC GPIO output bit clear register	0x3FF48408	WO
RTCIO_RTC_GPIO_ENABLE_REG	RTC GPIO output enable register	0x3FF4840C	R/W
RTCIO_RTC_GPIO_ENABLE_W1TS_REG	RTC GPIO output enable bit set register	0x3FF48410	WO
RTCIO_RTC_GPIO_ENABLE_W1TC_REG	RTC GPIO output enable bit clear register	0x3FF48414	WO
RTCIO_RTC_GPIO_STATUS_REG	RTC GPIO interrupt status register	0x3FF48418	WO
RTCIO_RTC_GPIO_STATUS_W1TS_REG	RTC GPIO interrupt status bit set register	0x3FF4841C	WO
RTCIO_RTC_GPIO_STATUS_W1TC_REG	RTC GPIO interrupt status bit clear register	0x3FF48420	WO
RTCIO_RTC_GPIO_IN_REG	RTC GPIO input register	0x3FF48424	RO
RTCIO_RTC_GPIO_PIN0_REG	RTC configuration for pin 0	0x3FF48428	R/W
RTCIO_RTC_GPIO_PIN1_REG	RTC configuration for pin 1	0x3FF4842C	R/W
RTCIO_RTC_GPIO_PIN2_REG	RTC configuration for pin 2	0x3FF48430	R/W
RTCIO_RTC_GPIO_PIN3_REG	RTC configuration for pin 3	0x3FF48434	R/W
RTCIO_RTC_GPIO_PIN4_REG	RTC configuration for pin 4	0x3FF48438	R/W
RTCIO_RTC_GPIO_PIN5_REG	RTC configuration for pin 5	0x3FF4843C	R/W
RTCIO_RTC_GPIO_PIN6_REG	RTC configuration for pin 6	0x3FF48440	R/W
RTCIO_RTC_GPIO_PIN7_REG	RTC configuration for pin 7	0x3FF48444	R/W
RTCIO_RTC_GPIO_PIN8_REG	RTC configuration for pin 8	0x3FF48448	R/W
RTCIO_RTC_GPIO_PIN9_REG	RTC configuration for pin 9	0x3FF4844C	R/W
RTCIO_RTC_GPIO_PIN10_REG	RTC configuration for pin 10	0x3FF48450	R/W
RTCIO_RTC_GPIO_PIN11_REG	RTC configuration for pin 11	0x3FF48454	R/W
RTCIO_RTC_GPIO_PIN12_REG	RTC configuration for pin 12	0x3FF48458	R/W
RTCIO_RTC_GPIO_PIN13_REG	RTC configuration for pin 13	0x3FF4845C	R/W
RTCIO_RTC_GPIO_PIN14_REG	RTC configuration for pin 14	0x3FF48460	R/W
RTCIO_RTC_GPIO_PIN15_REG	RTC configuration for pin 15	0x3FF48464	R/W
RTCIO_RTC_GPIO_PIN16_REG	RTC configuration for pin 16	0x3FF48468	R/W
RTCIO_RTC_GPIO_PIN17_REG	RTC configuration for pin 17	0x3FF4846C	R/W
RTCIO_DIG_PAD_HOLD_REG	RTC GPIO hold register	0x3FF48474	R/W
GPIO RTC function configuration registers			
RTCIO_HALL_SENS_REG	Hall sensor configuration	0x3FF48478	R/W
RTCIO_SENSOR_PADS_REG	Sensor pads configuration register	0x3FF4847C	R/W
RTCIO_ADC_PAD_REG	ADC configuration register	0x3FF48480	R/W
RTCIO_PAD_DAC1_REG	DAC1 configuration register	0x3FF48484	R/W
RTCIO_PAD_DAC2_REG	DAC2 configuration register	0x3FF48488	R/W

Name	Description	Address	Access
RTCIO_XTAL_32K_PAD_REG	32KHz crystal pads configuration register	0x3FF4848C	R/W
RTCIO_TOUCH_CFG_REG	Touch sensor configuration register	0x3FF48490	R/W
RTCIO_TOUCH_PAD0_REG	Touch pad configuration register	0x3FF48494	R/W
'''	'''		
RTCIO_TOUCH_PAD9_REG	Touch pad configuration register	0x3FF484B8	R/W
RTCIO_EXT_WAKEUP0_REG	External wake up configuration register	0x3FF484BC	R/W
RTCIO_XTL_EXT_CTR_REG	Crystal power down enable GPIO source	0x3FF484C0	R/W
RTCIO_SAR_I2C_IO_REG	RTC I2C pad selection	0x3FF484C4	R/W

Recursos

En inglés

- **ESP32forth** Página mantenida por Brad NELSON, el creador de ESP32forth. Allí encontrarás todas las versiones (ESP32, Windows, Web, Linux...) <https://esp32forth.appspot.com/ESP32forth.html>

•

En francés

- **ESP32 Forth** sitio en dos idiomas (francés, inglés) con muchos ejemplos <https://esp32.arduino-forth.com/>

GitHub

- **Ueforth** Recursos mantenidos por Brad NELSON. Contiene todos los archivos fuente en lenguaje Forth y C para ESP32forth <https://github.com/flagxor/ueforth>
- **ESP32forth** códigos fuente y documentación para ESP32 en adelante. Recursos mantenidos por Marc PETREMANN <https://github.com/MPETREMANN11/ESP32forth>
- **ESP32forthStation** recursos mantenidos por Ulrich HOFFMAN. Computadora Forth independiente con computadora de placa única LillyGo TTGO VGA32 y ESP32forth <https://github.com/uho/ESP32forthStation>
- **ESP32Forth** recursos mantenidos por F. J. RUSSO <https://github.com/FJRusso53/ESP32Forth>
- **esp32forth-addons** recursos mantenidos por Peter FORTH <https://github.com/PeterForth/esp32forth-addons>
- **Esp32forth-org** Repositorio de código para miembros de los grupos Forth2020 y ES32forth <https://github.com/Esp32forth-org>

•

índice léxico

and.....	38	internals.....	140	SPIFFS.....	142
ansi.....	92	interrupts.....	141	streams.....	142
asm.....	139	is.....	96	struct.....	70
BASE.....	82	ledc.....	141	structures.....	70, 92, 143
bg.....	57	login.....	117	72
binary.....	33	normal.....	57	tasks.....	143
bluetooth.....	140	números aleatorios.....	135	telnetd.....	118, 143
Colores de texto.....	57	oled.....	75, 92, 129, 141	Tera Term.....	111
create.....	101	page.....	57	to.....	62
decimal.....	33	pi.....	78	u.....	36
DECIMAL.....	82	random.....	136	visual.....	143
defer.....	96	RECORDFILE.....	121	voclist.....	91
DOES>.....	101	registers.....	141	web-interface.....	143
dump.....	49	riscv.....	141	WiFi.....	143
editor.....	140	rnd.....	136	Wire.....	143
EMIT.....	85	RNG_DATA_REG.....	136	xtensa.....	143
ESP.....	140	rtos.....	142	:noname.....	98
EXECUTE.....	95	S".....	86	.".....	86
f.....	78	SD.....	142	.s.....	49
fconstant.....	79	SD_MMC.....	142	{.....	61
fg.....	57	see.....	49	}.....	61
FORTH.....	138	Serial.....	142	#.....	83
fvariable.....	79	server.....	118	#>.....	83
hex.....	33	set-precision.....	78	#S.....	83
HEX.....	82	shift.....	37	+to.....	62
HOLD.....	83	sockets.....	142	<#.....	83
httpd.....	140	SPACE.....	87		
insides.....	140	spi.....	142		