El gran libro
por ESP32forth

versión 1.17 - 25. enero 2024

Autor

- Marc PETREMANN petremann@arduino-forth.com

Colaboradores

- Vaclav POSSELT
- Bob EDWARDS

Indice

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

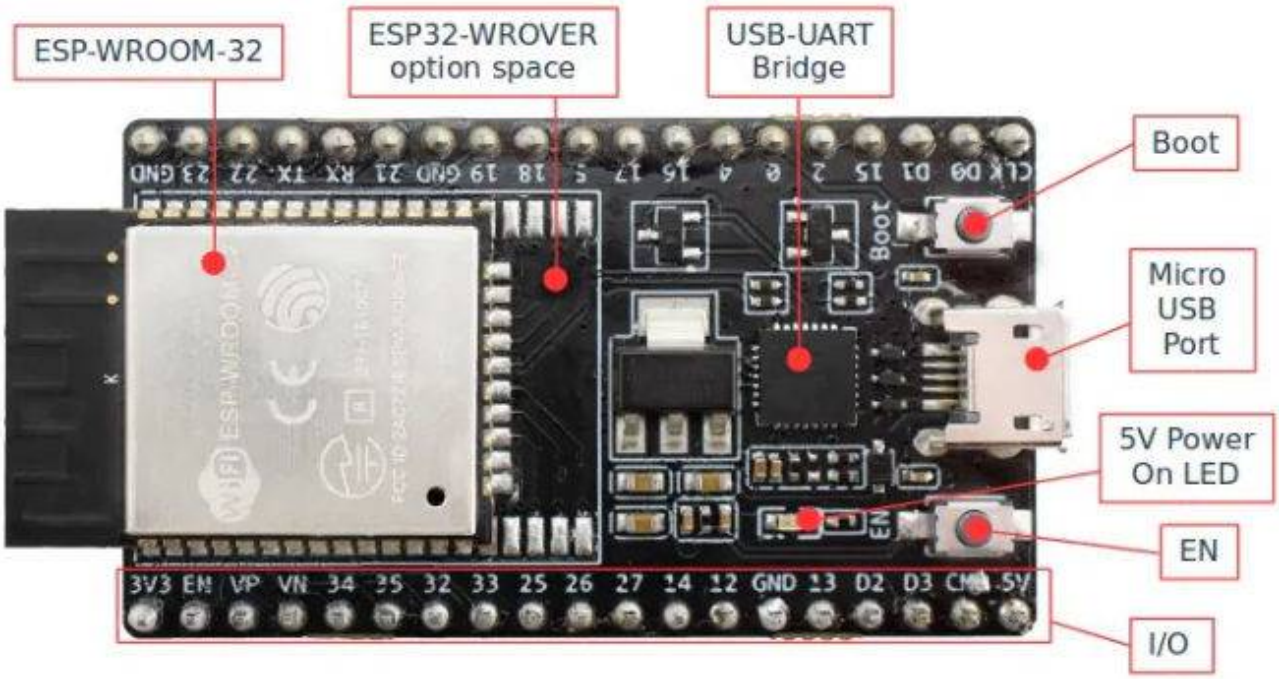
Descubrimiento de la tarjeta ESP32

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.

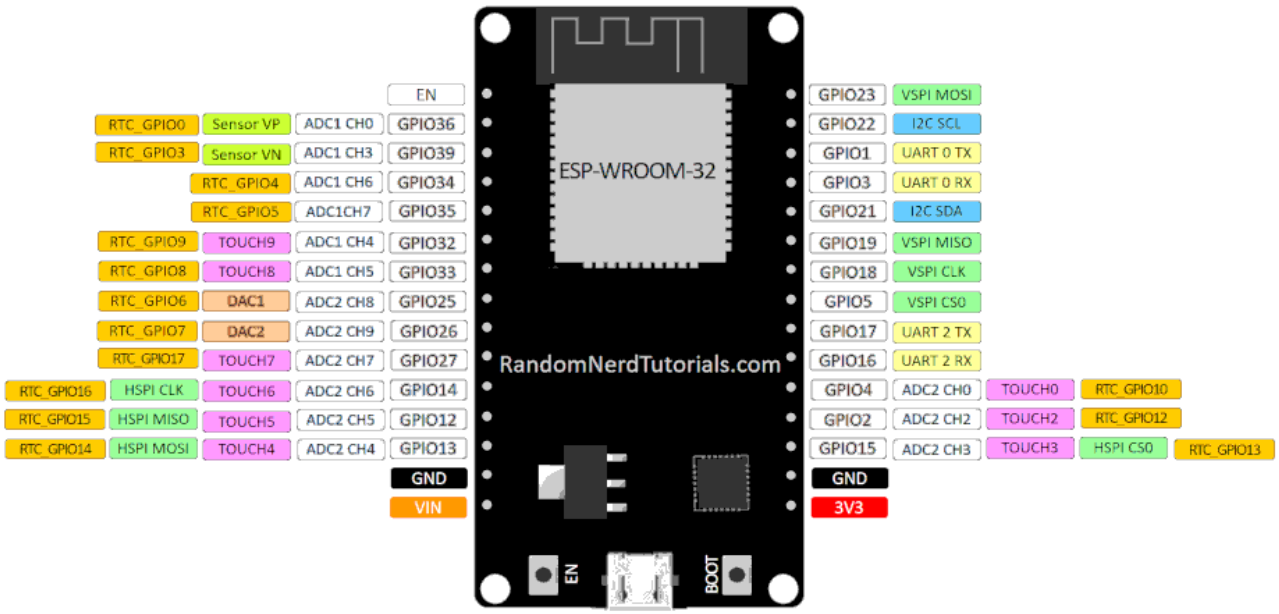
La mayoría de los conectores tienen múltiples usos.

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

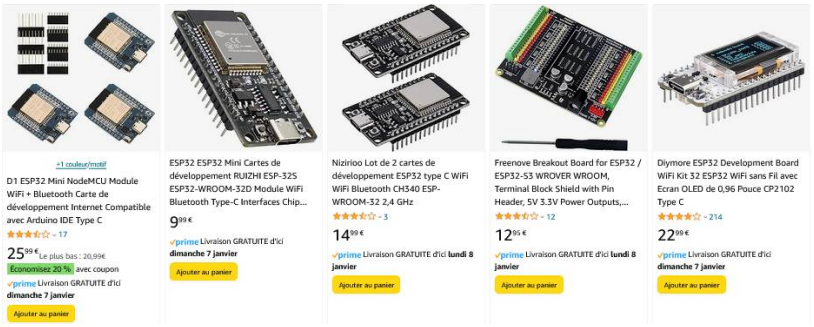


Figure 1: un grand choix de cartes ESP32 sur AMAZON

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.

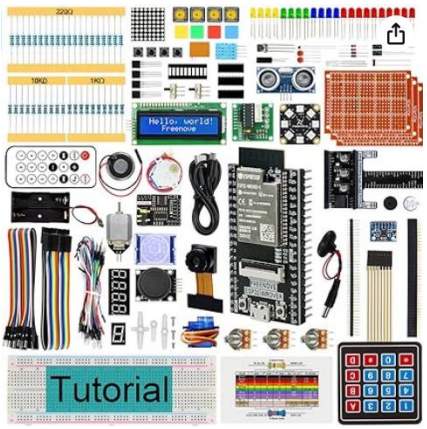


Figure 2: kit ESP32

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.

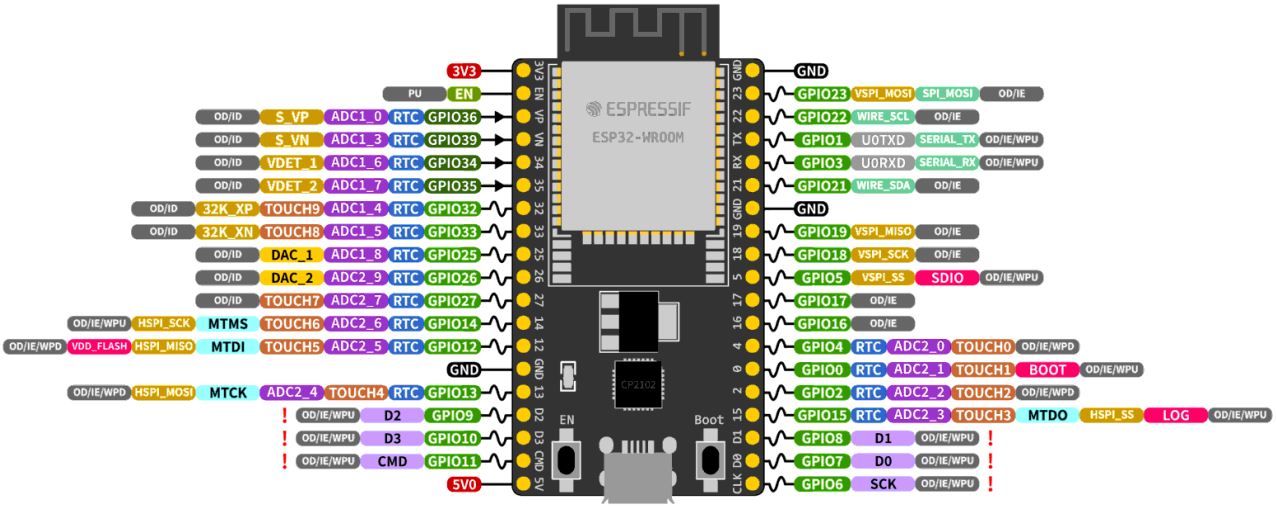
Figure 3: carte ESP32 Wroom 32

- Procesador ESP-WROOM-32
- WLAN 802.11 b/g/n
- Bluetooth 4.2/BLE
- Procesador Tensilica L108 de 32 bits a 160 MHz
- 512 KB de SRAM y 16 MB de memoria flash
- 32 pines de E/S digitales *(3,3 V)*
- 6 pines analógicos-digitales
- 3x UART, 2x SPI, 2x I2C
- 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:

- **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: 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.

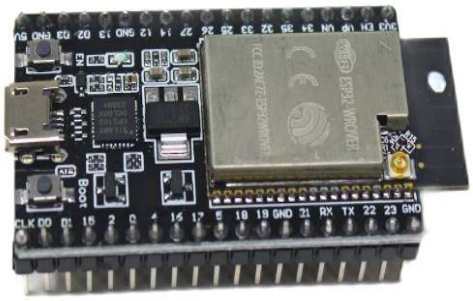


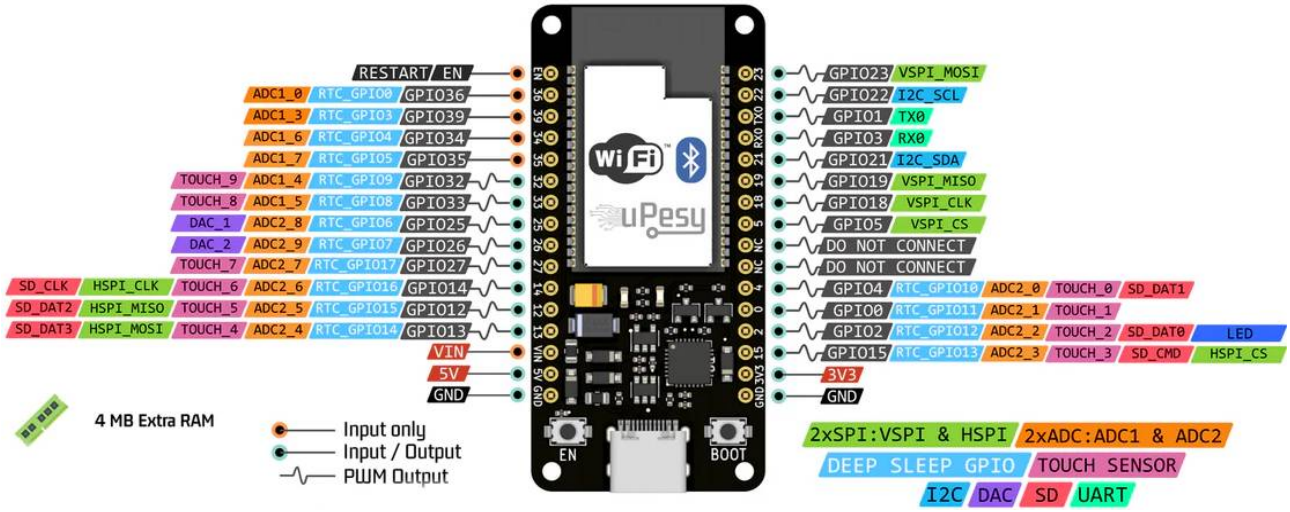
Figure 4: carte ESP32 Wrover

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



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.

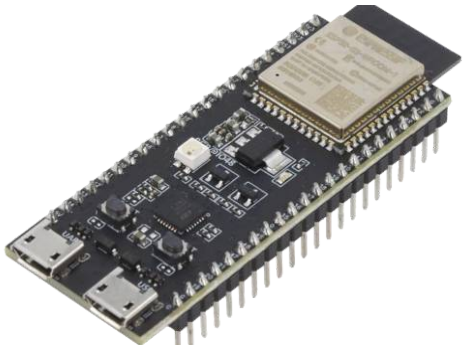


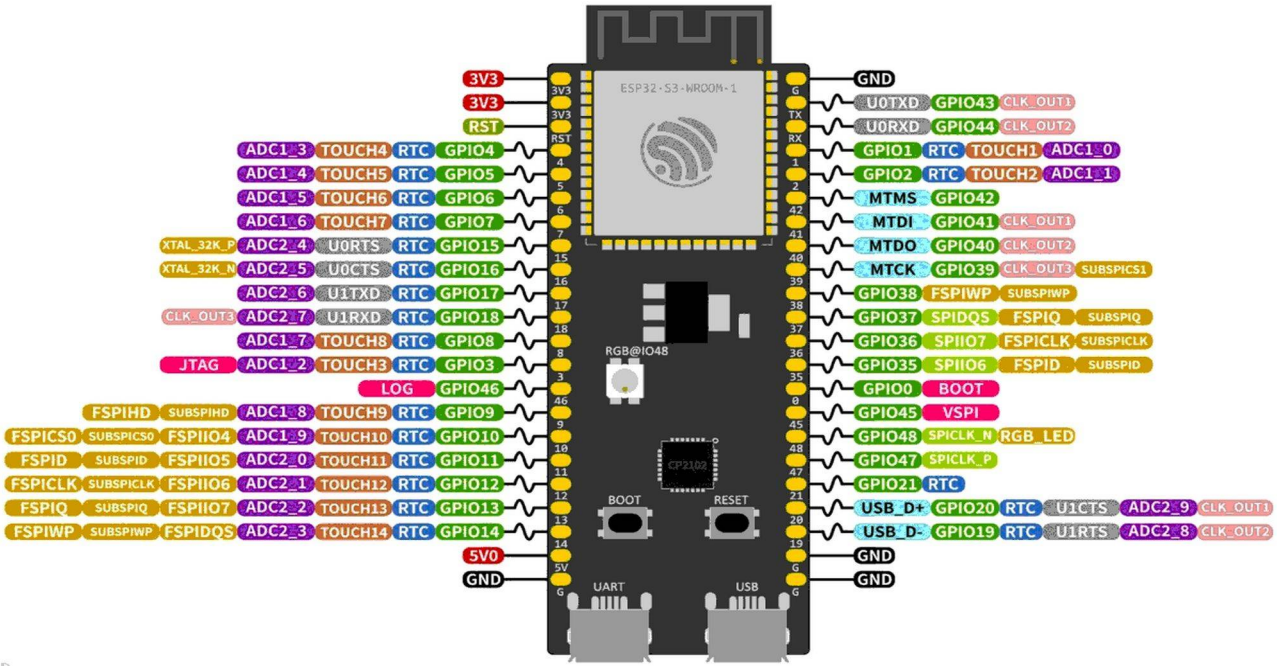
Figure 5: carte ESP32 S3

- 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

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


ESP32-S3-WROOM-1
3V3
3V3
RST
GPIO4 RTC TOUCH4 ADC1_3
GPIO5 RTC TOUCH5 ADC1_4
GPIO6 RTC TOUCH6 ADC1_5
GPIO7 RTC TOUCH7 ADC1_6
GPIO15 RTC U0RTS ADC2_4 XTAL_32K_P
GPIO16 RTC U0CTS ADC2_5 XTAL_32K_N
GPIO17 RTC U1TXD ADC2_6
GPIO18 RTC U1RXD ADC2_7 CLK_OUT3
GPIO8 RTC TOUCH8 ADC1_7
GPIO3 RTC TOUCH3 ADC1_2 JTAG
GPIO46 LOG
GPIO9 RTC TOUCH9 ADC1_8 FSPIHD SUBSPIHD
GPIO10 RTC TOUCH10 ADC1_9 FSPII04 SUBSPICS0 FSPICS0
GPIO11 RTC TOUCH11 ADC2_0 FSPII05 SUBSPID FSPID
GPIO12 RTC TOUCH12 ADC2_1 FSPII06 SUBSPICLK FSPICLK
GPIO13 RTC TOUCH13 ADC2_2 FSPII07 SUBSPIQ FSPIQ
GPIO14 RTC TOUCH14 ADC2_3 FSPIDQS SUBSPIWP FSPIWP
5V0
GND
GND
GPIO43 CLK_OUT1 U0TXD
GPIO44 CLK_OUT2 U0RXD
GPIO1 RTC TOUCH1 ADC1_0
GPIO2 RTC TOUCH2 ADC1_1
GPIO42 MTMS
GPIO41 CLK_OUT1 MTDI
GPIO40 CLK_OUT2 MTDO
GPIO39 CLK_OUT3 SUBSPICS1 MTCK
GPIO38 FSPIWP SUBSPIWP
GPIO37 SPIDQS FSPIQ SUBSPIQ
GPIO36 SPII07 FSPICLK SUBSPICLK
GPIO35 SPII06 FSPID SUBSPID
GPIO0 BOOT
GPIO45 VSPI
GPIO48 SPICLK_N RGB_LED
GPIO47 SPICLK_P
GPIO21 RTC
GPIO20 RTC U1CTS ADC2_9 CLK_OUT1 USB_D+
GPIO19 RTC U1RTS ADC2_8 CLK_OUT2 USB_D-
GND
GND
RGB@IO48
BOOT
RESET
UART
USB

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...
```

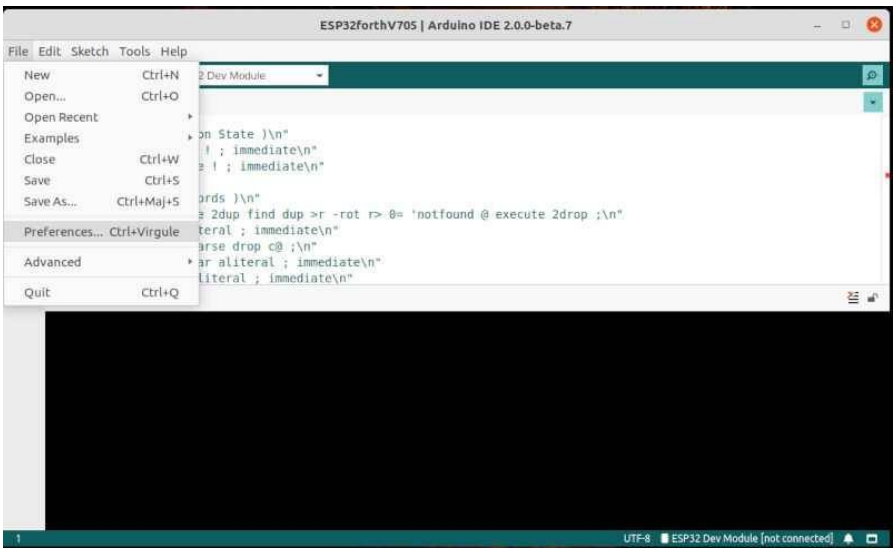
Compilando e instalando ESP32forth

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

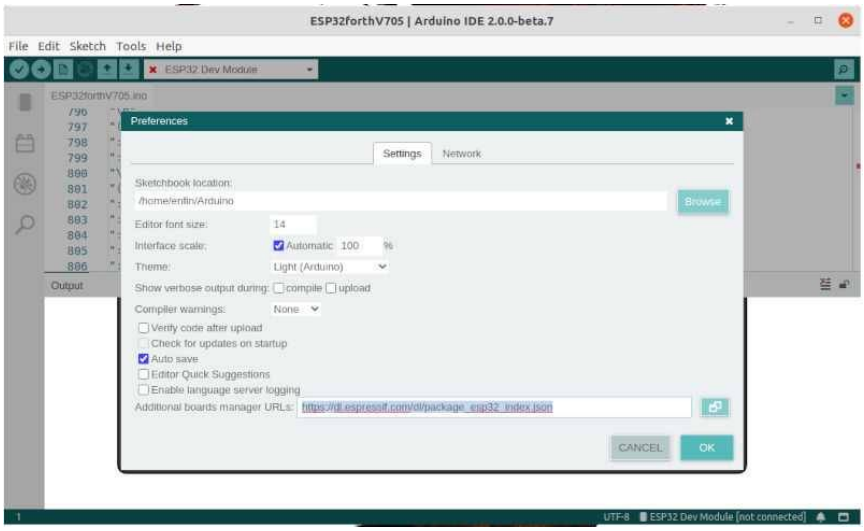
Una vez instalado ARDUINO IDE, ejecútelo. ARDUINO IDE está abierto, aquí la versión 2.0[1]. Haga clic en *file* y seleccione *Preferences* :

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.

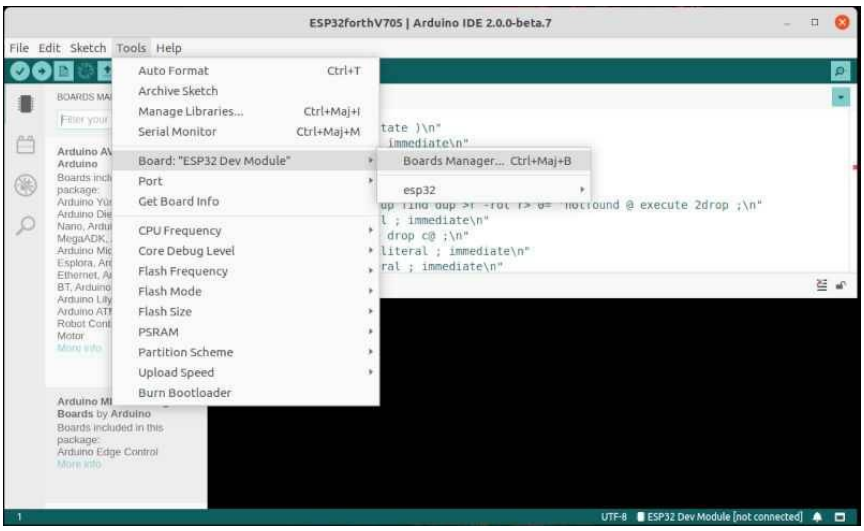


En la ventana que aparece, vaya al cuadro de entrada marcado *Additional boards manager URLs*: e ingrese esta línea:

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



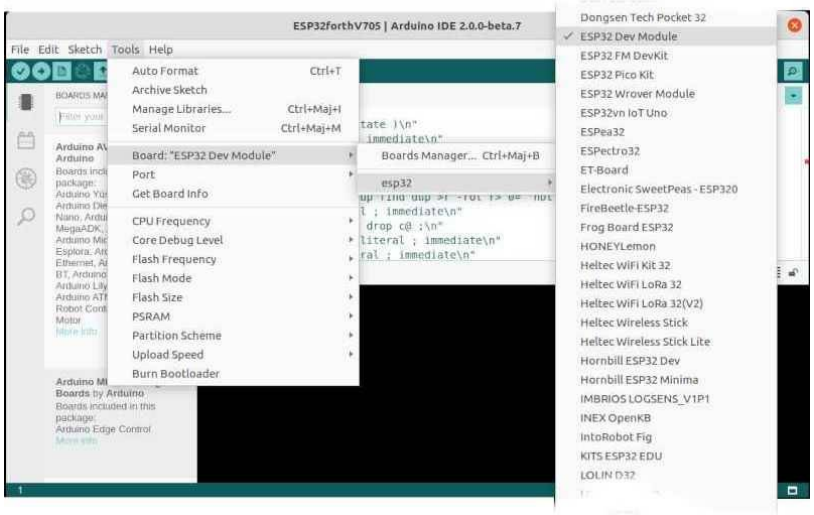
A continuación, haga clic en *Tools* y seleccione *Board* :.



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* :

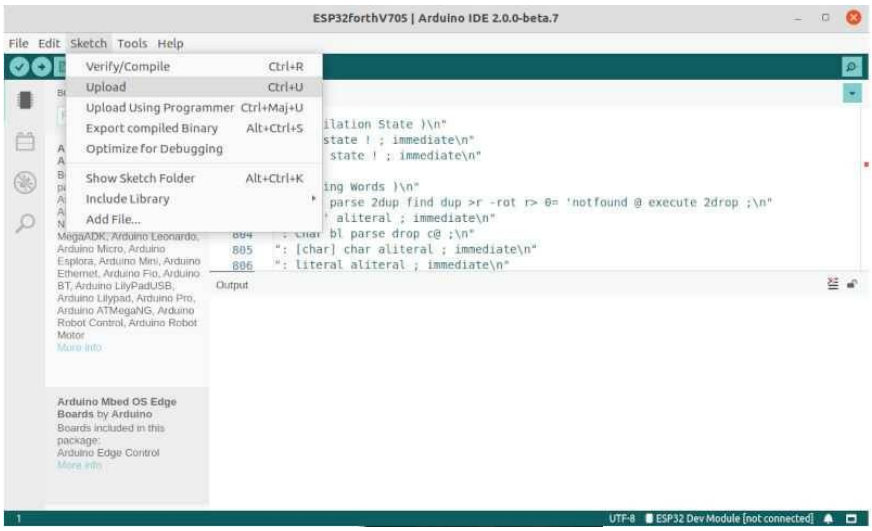
```
-- TOOLS----+-- BOARD    ----+-- ESP32 ------+-- ESP32 Dev Module
            +-- Port: -------+-- COMx
            |
            +-- CPU Frequency ---------+-- 240 Mhz
            +-- Core Debug Level ------+-- None
```

```
            +-- 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

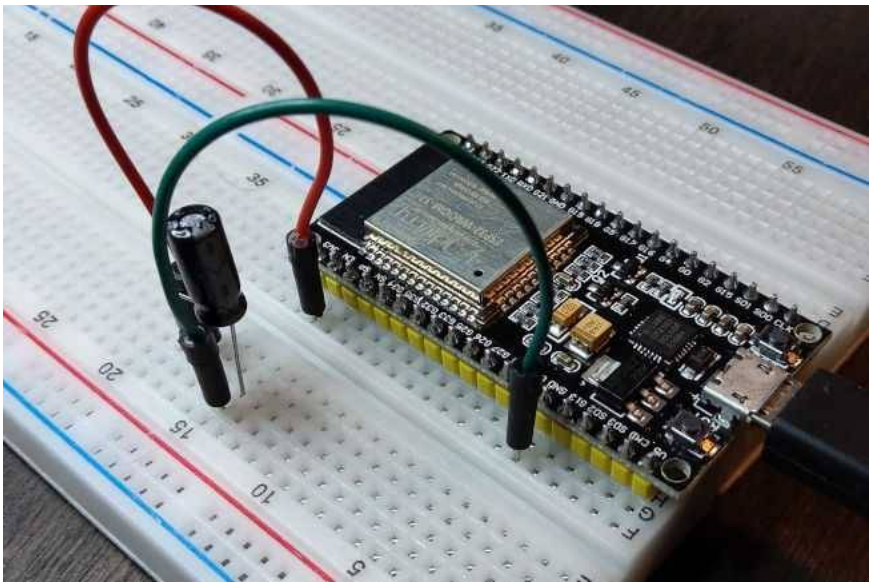
Solucionar el error de conexión de carga

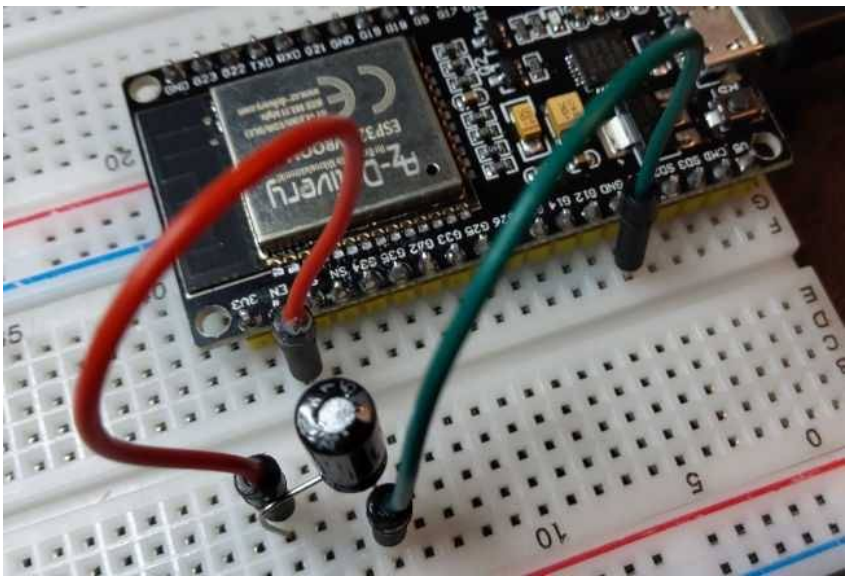
Aprenda cómo solucionar el error fatal que ocurrió: "Failed to connect to ESP32: Timed out waiting for packet header" al intentar cargar un nuevo código a su tarjeta ESP32 de una vez por todas.

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:

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:





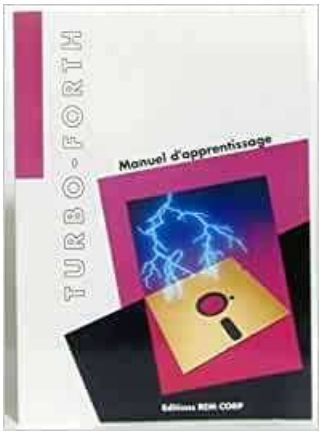
Esta manipulación sólo es necesaria si estás en la fase de carga de ESP32 adelante desde ARDUINO IDE. Una vez que ESP32forth está instalado en la placa ESP32, el uso de este condensador ya no es necesario.

¿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 Loisitech - 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 autómata 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.

Analicé 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 :

```
_O_ __/ \ luces LED
```

¡Sí! ¡ La secuencia `_O_ __/` 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 ;
```

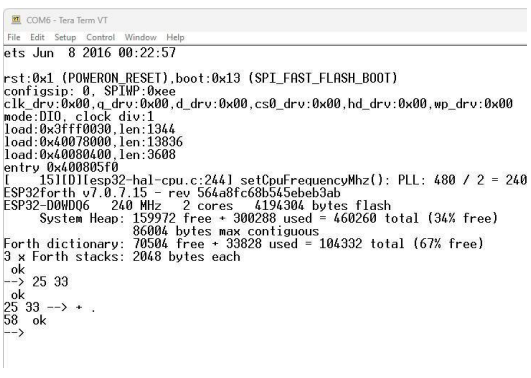
Este código, aparentemente simple, ya requiere una base de conocimientos, como la noción de dirección de memoria, registro, máscaras binarias, números hexadecimales.

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



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: i `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.` :

```
$ffffffff 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
00000000000000000000000000000001  \ empujar 1 en la pila
11111111111111111111111111111111  \ 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:

```
00000000000000000000000000000001
11111111111111111111111111111111
                              10
```

Etapa siguiente :

```
00000000000000000000000000000001
11111111111111111111111111111111
                              10
                             100
```

Al final tendremos el resultado:

```
 00000000000000000000000000000001
 11111111111111111111111111111111
100000000000000000000000000000000
```

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                                          00 00 00 00
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** :

```
ddecimal
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 regSet
```

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: %00000000000000000000000000000001
\  after shift: %00000010000000000000000000000000
```

Como recordatorio, los GPIO [2]están 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  \ %00000010000000000000000000000000
\ 1 17 lshift  \ %00000010000000001000000000000000
\  or          \ %00000010000000001000000000000000
```

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 IDE[3] 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 `csl` 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     \海報今天的日期
….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, i **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
        sendDatasToSSD1306
  ;
```

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 ( øC --- øF) \ 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

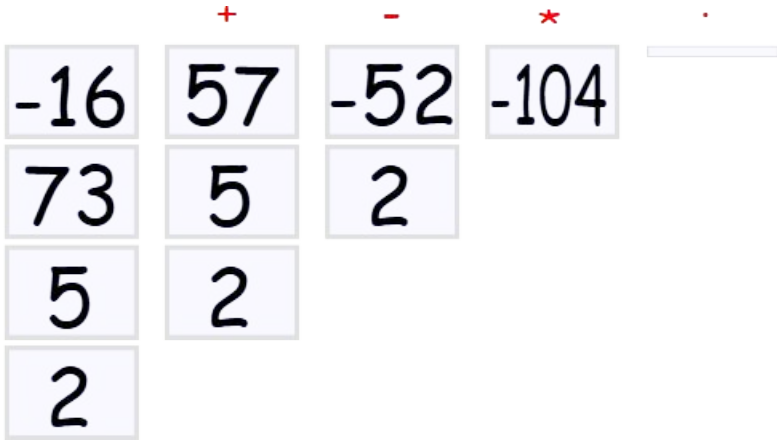
Célula	contenido	comentario
0	-dieciséis	(TOS) Arriba a la derecha
1	73	(NOS) El siguiente en la pila
2	5	
3	2	

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`** limina 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.

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.

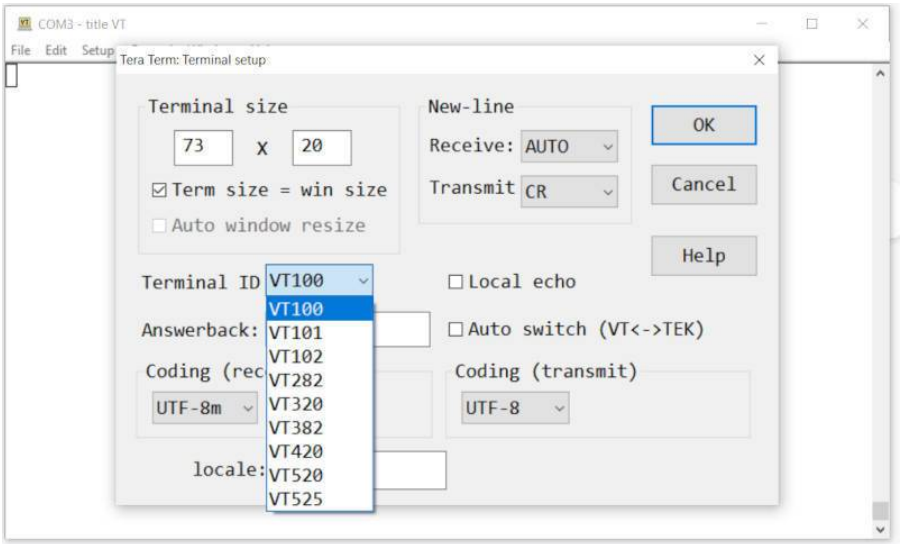
Ahora podemos acceder a la memoria asignada a la **graphic-array** usando las palabras de búsqueda y almacenamiento explicadas anteriormente. Para calcular la dirección del tercer byte asignado a **graphic-array** podemos escribir **graphic-array 2 +** , recordando que los índices comienzan en 0.

```
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:

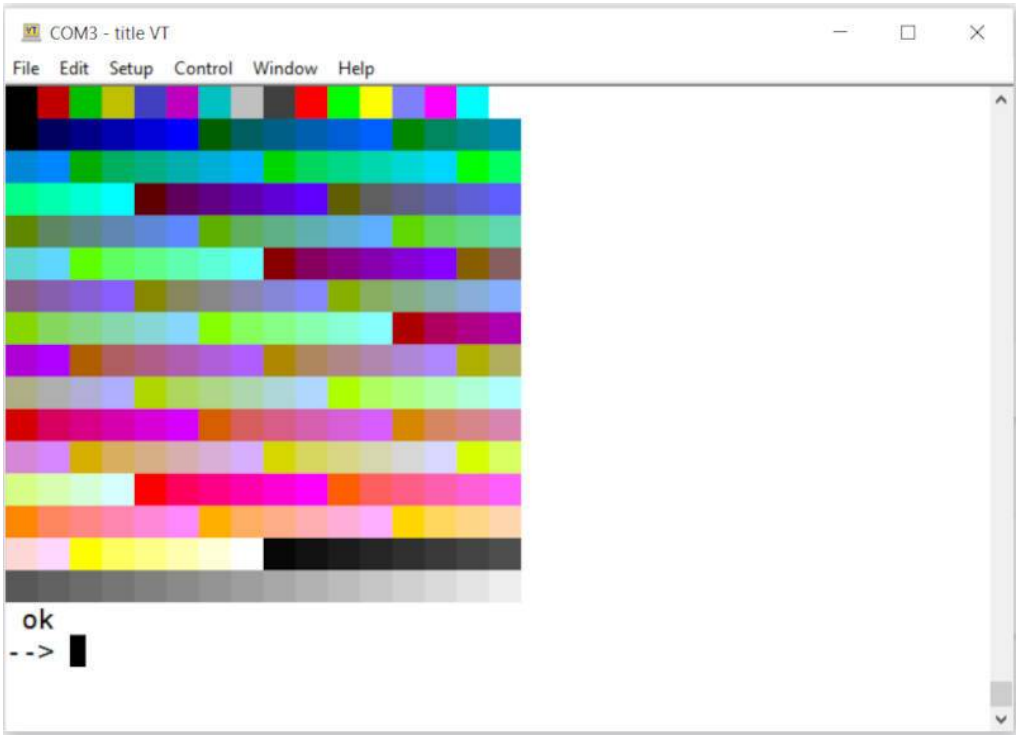
```
: testFG ( -- )
    page
    16 0 do
        16 0 do
            j 16 * i + fg
            ." X"
        loop
        cr
    loop
    normal
  ;
```

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:

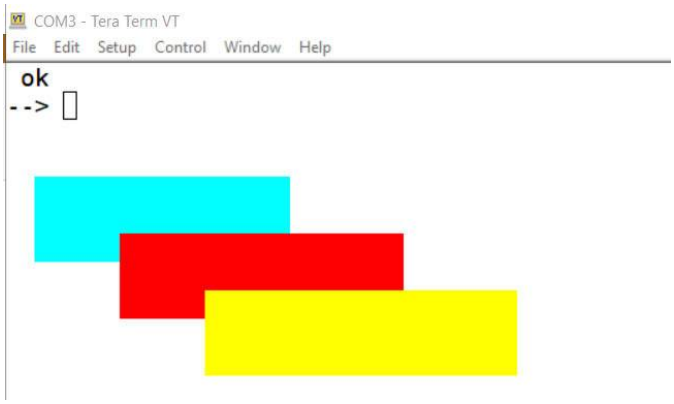


Posición de visualización

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:

```
: 2OVER { 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
            0START_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)
    0BRANCH rdrop rdrop space 16 0 (do) -28 local@ j 16 * R@ + + CA@ DUP 32 < OVER 127 > OR
    0BRANCH DROP 46 emit BRANCH emit 1 (+loop) 0BRANCH rdrop rdrop 1 (+loop)
    0BRANCH rdrop rdrop -4 local@ base ! cr cr rdrop rdrop rdrop rdrop rdrop ;
```

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 4x8 bits = 32 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:

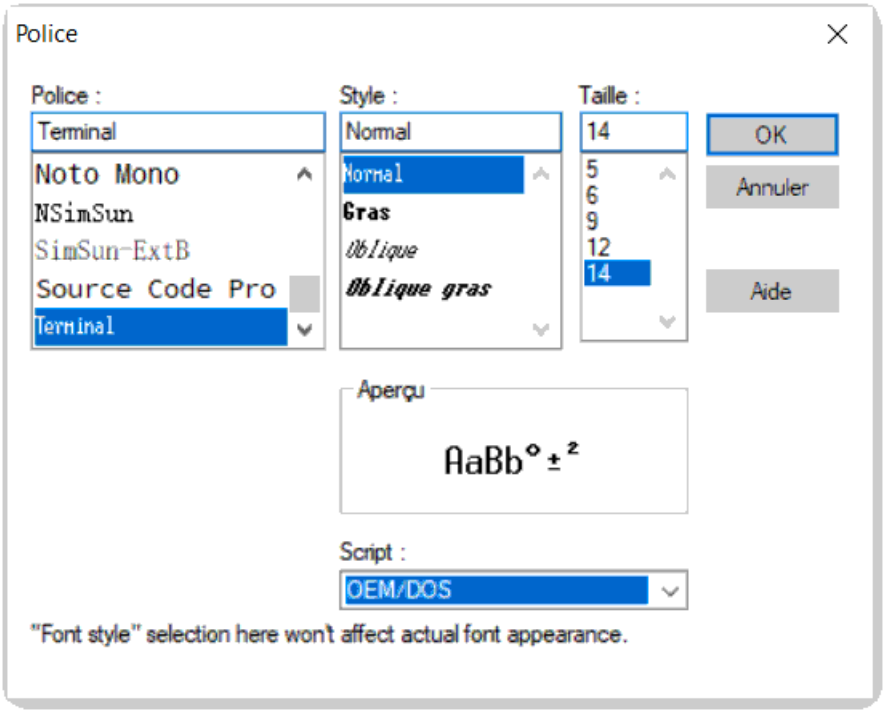
```
: xySCRaddr { x y -- addr }
    SCR_WIDTH y *
    x + mySCREEN +
  ;
: SCR@ ( x y -- c )
    xySCRaddr c@
  ;
: SCR! ( c x y -- )
    xySCRaddr c!
  ;
char X 15 5 SCR!    \ almacena el carácter X en el cuello 15 línea 5
15 5 SCR@ emit      \ visualizaciones
```

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:

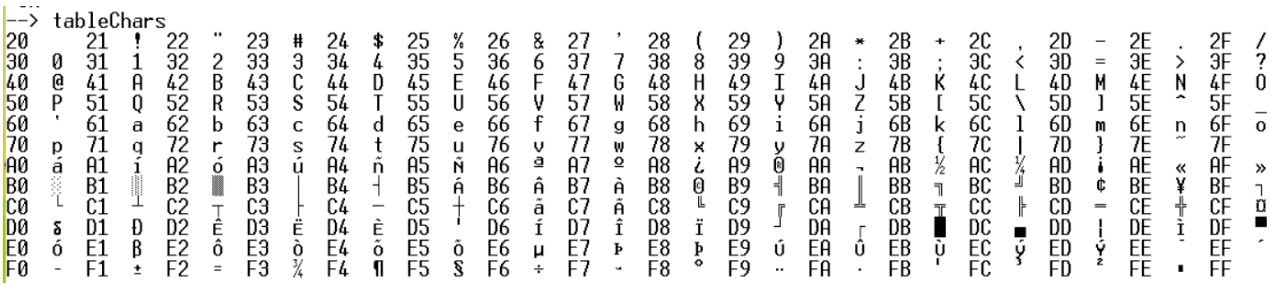
- en la barra de menú, haga clic en *Setup*
- seleccione *Font* y *Font*...
- configure la fuente a continuación:



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`:



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:

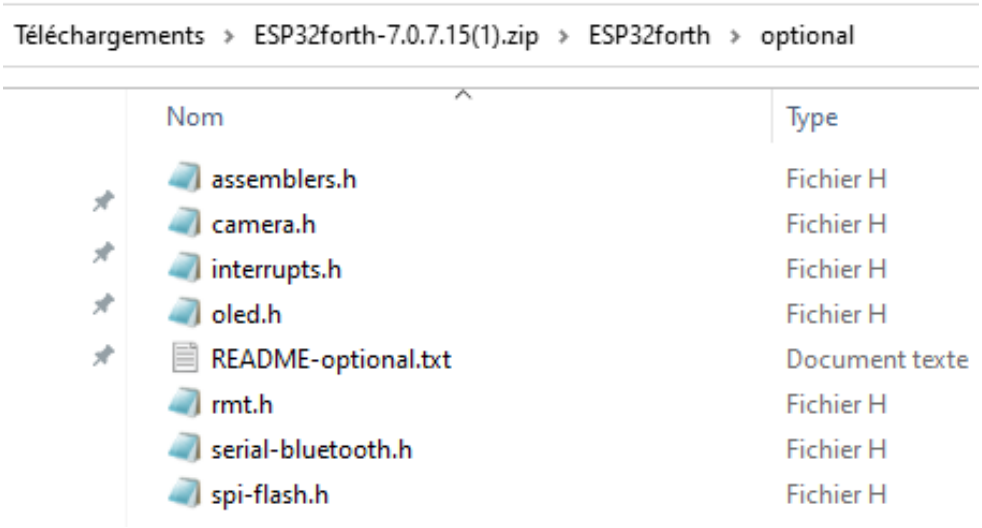
```
: .sprite { xpos ypos sprAddr -- }
    sprAddr >height c@ 0 do
        xpos ypos at-xy
        sprAddr >width c@ i *
        \ calcular el desplazamiento en los datos del sprite
        sprAddr >content +
        \ calcular la dirección real para la línea n en datos de
sprites
        sprAddr >width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

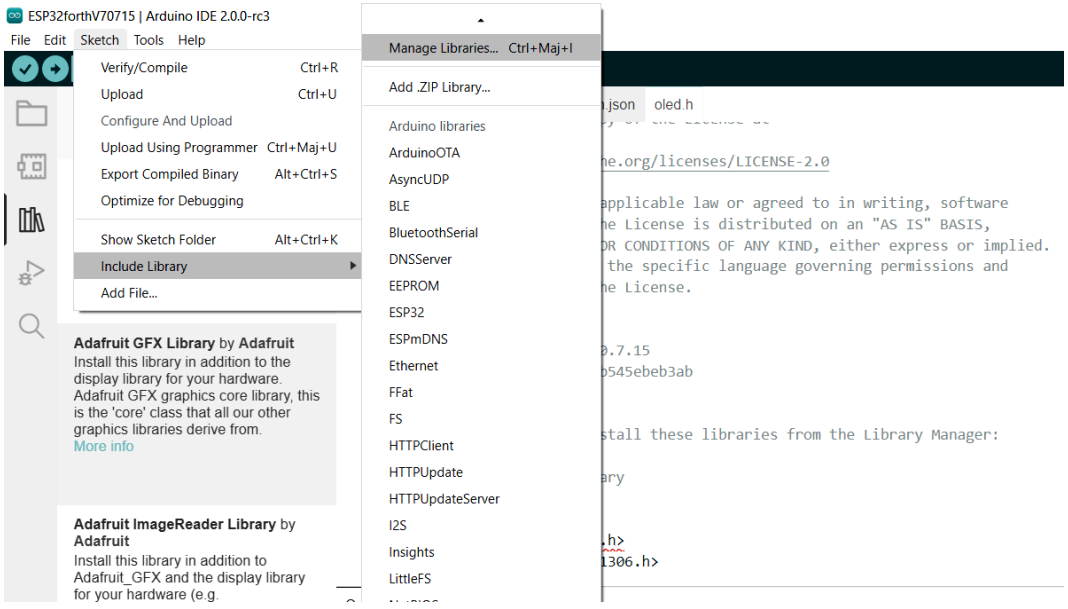
Desde ESP32 en adelante versión 7.0.7.15, las opciones están disponibles en la carpeta **optional**:



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 <= r2.
- **f<>** (r1 r2 -- fl) fl es verdadero si r1 <> 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 >= 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               \ pilas 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 +!                 \ incrémente #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 **LINE1** 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

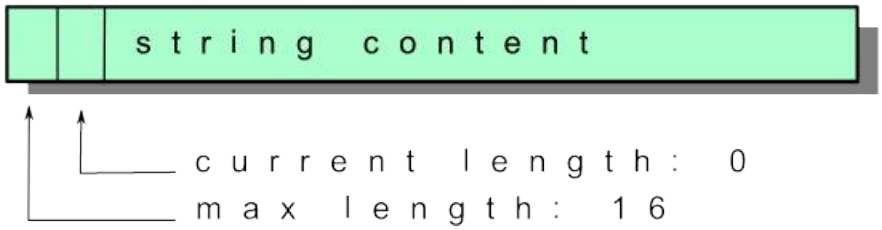
Las variables de texto alfanuméricas no existen de forma nativa en ESP32 en adelante. Aquí está el primer intento de definir la palabra **string** :

```
\ 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 frop 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 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" Saterday" ;
: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: miércoles
```

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 :

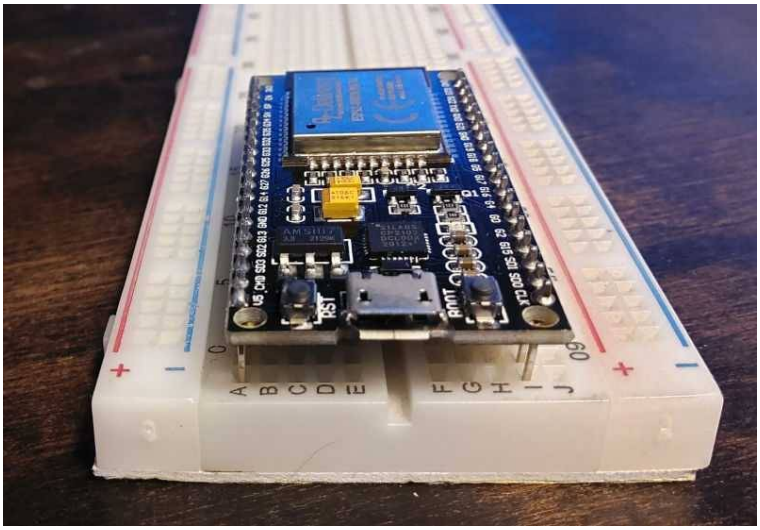
```
Zao An        \ display 早安
```

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:

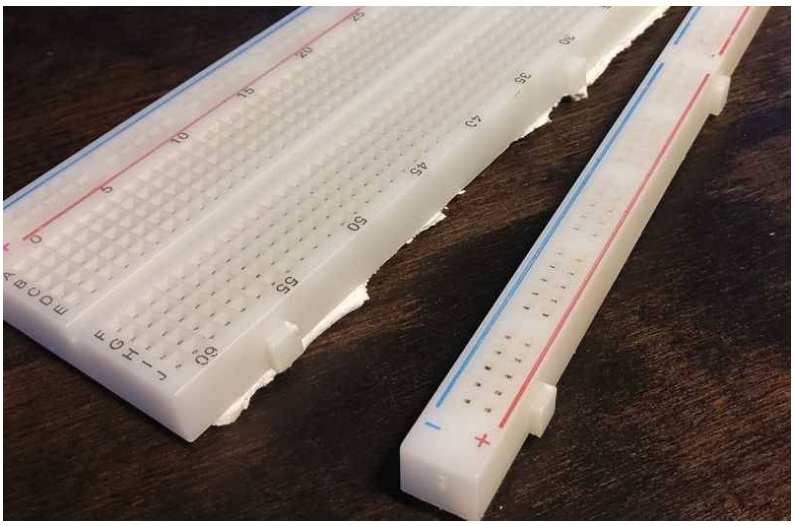


No existe una placa de pruebas específicamente adaptada a las placas ESP32.

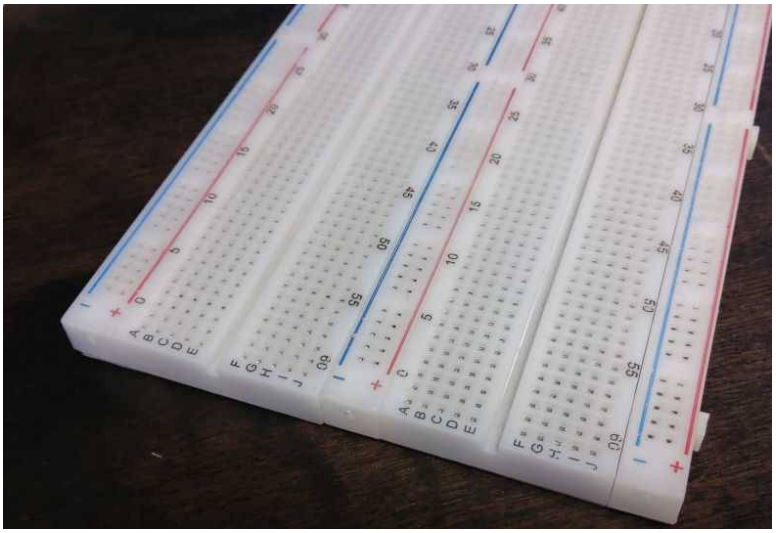
Construya una placa de pruebas adecuada para la placa ESP32

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

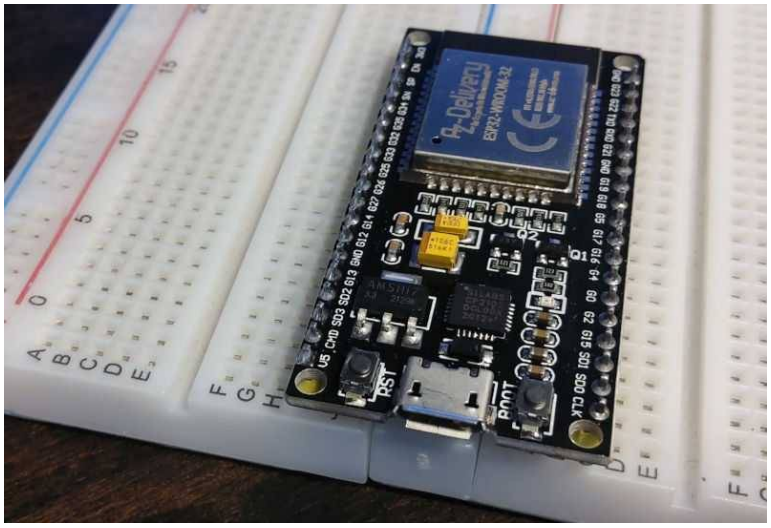
En una de las placas quitaremos un cable eléctrico. Para hacer esto, use un cortador y corte desde abajo. Debería poder separar esta línea eléctrica de esta manera:



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.

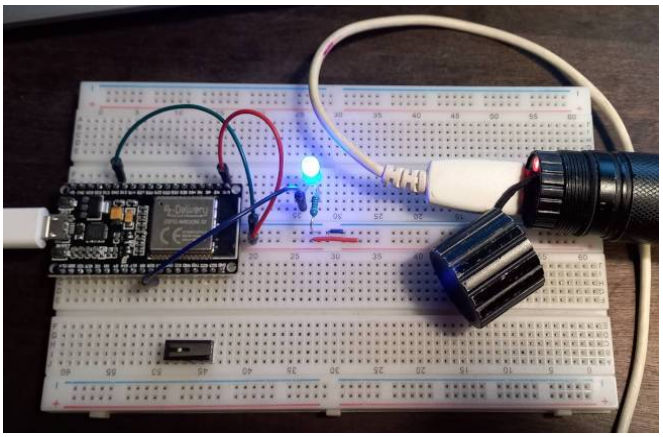
Alimentando la placa ESP32

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).

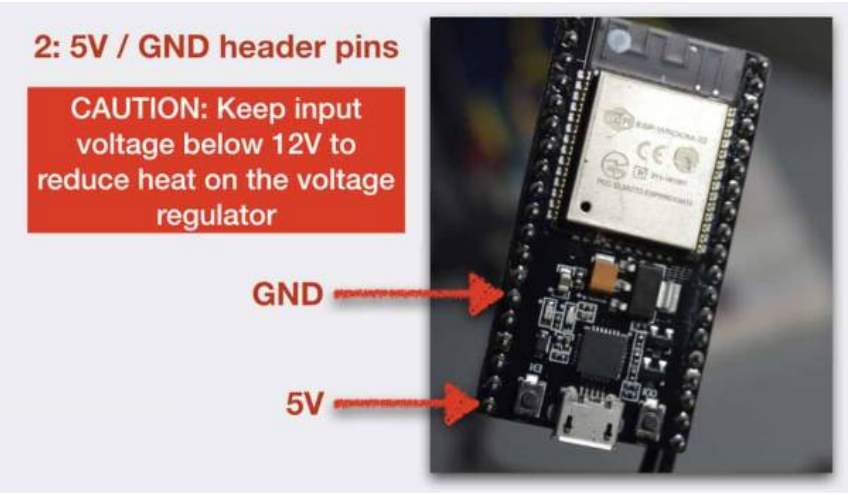
Aquí alimentamos nuestra placa ESP32 con una batería de respaldo para dispositivos móviles.

Alimentación mediante pin de 5V

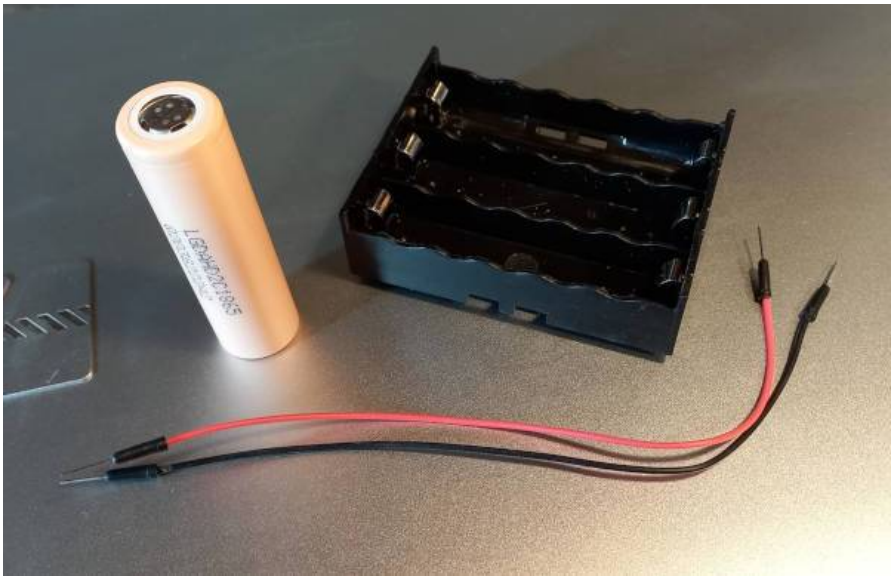
La segunda opción es conectar una fuente de alimentación externa no regulada al pin de 5V y a tierra. Cualquier voltaje entre 5 y 12 voltios debería funcionar.

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.

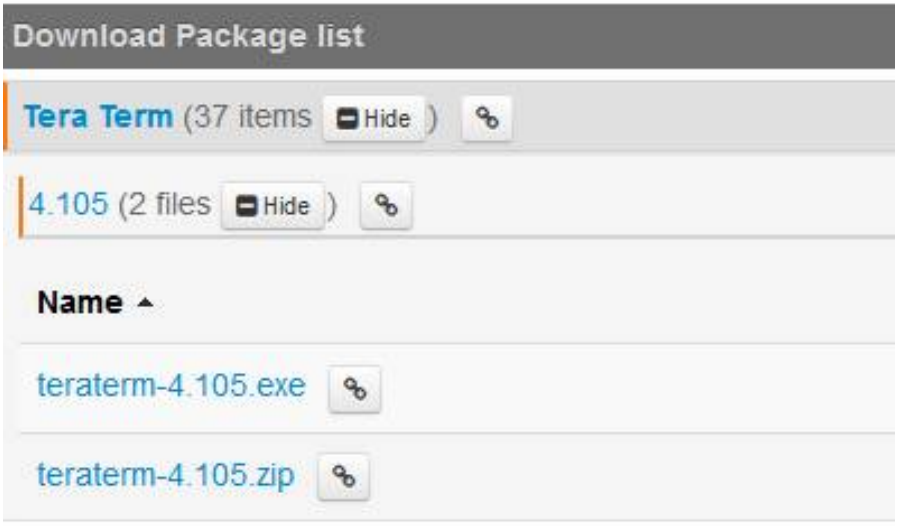
Instale y use la terminal Tera Term en Windows

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:

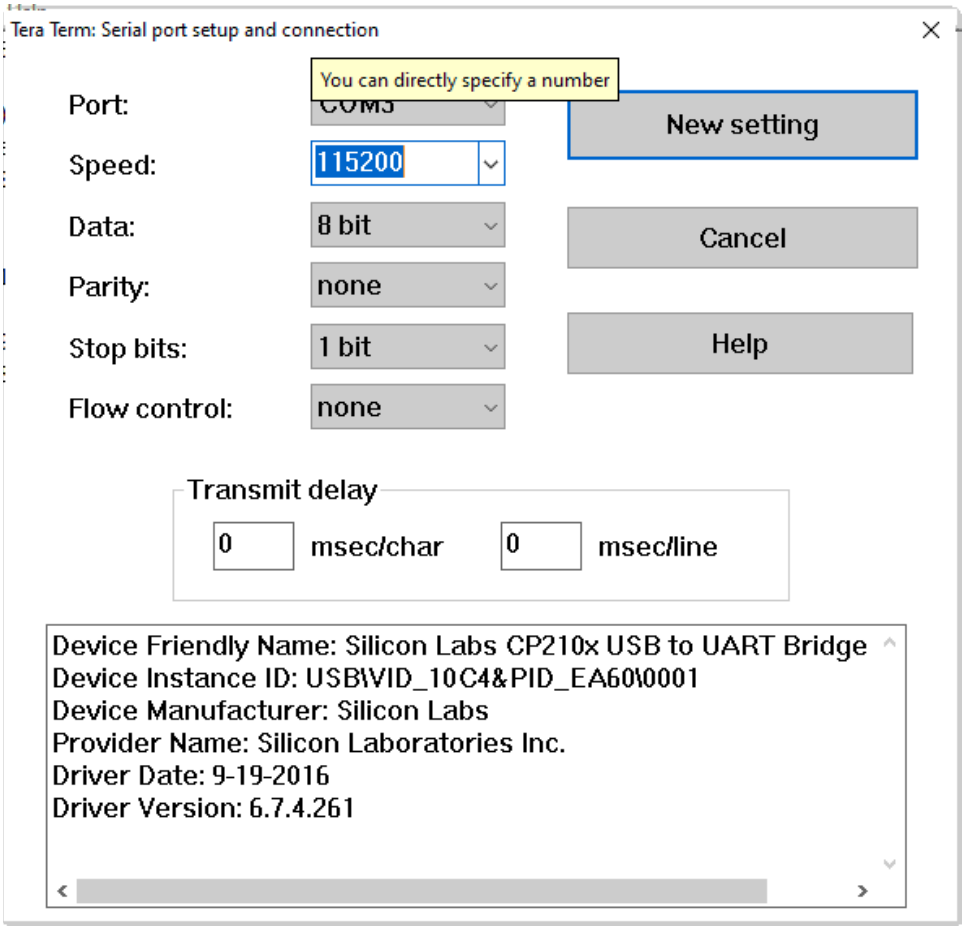


Instale Tera Term. La instalación es rápida y facil.

Configuración de Tera Term

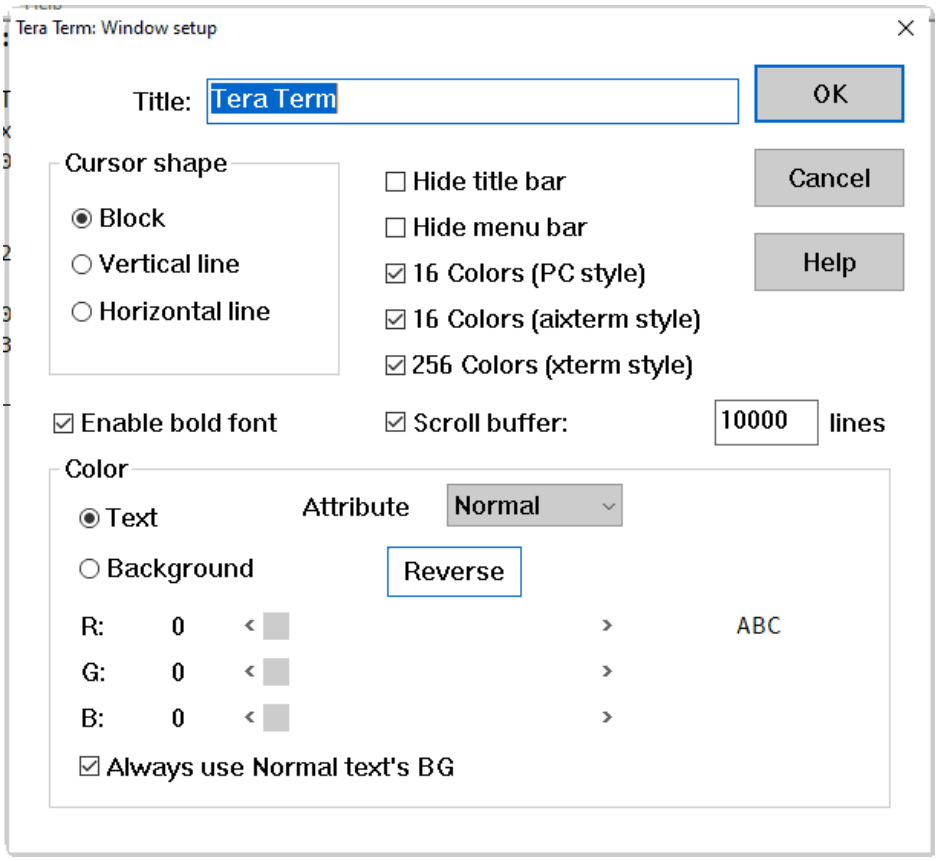
Para comunicarse con la tarjeta ESP32, debes ajustar ciertos parámetros:

- haga clic en Configuración -> puerto serie



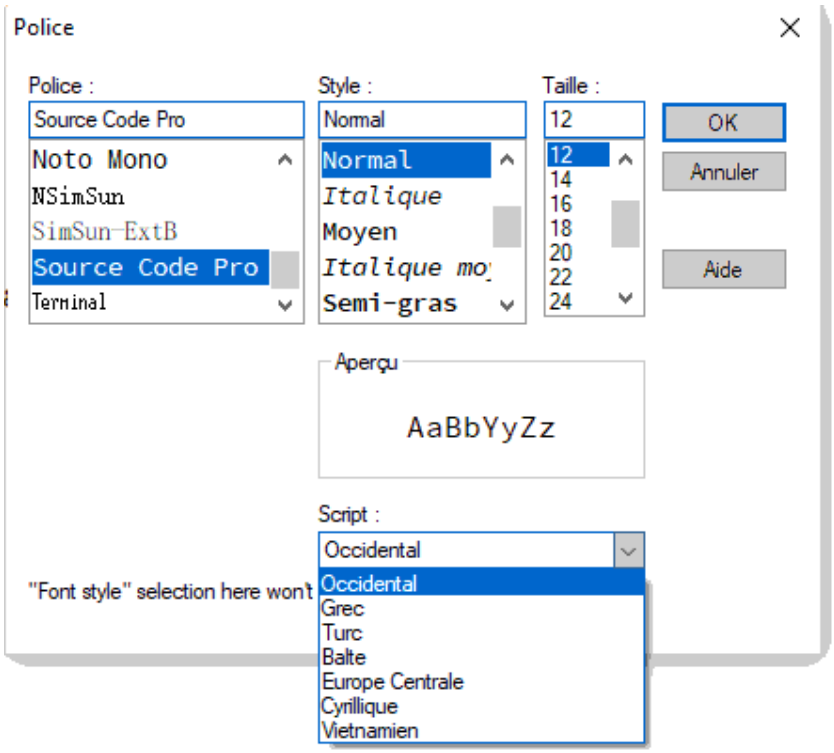
Para una visualización cómoda:

- 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** .

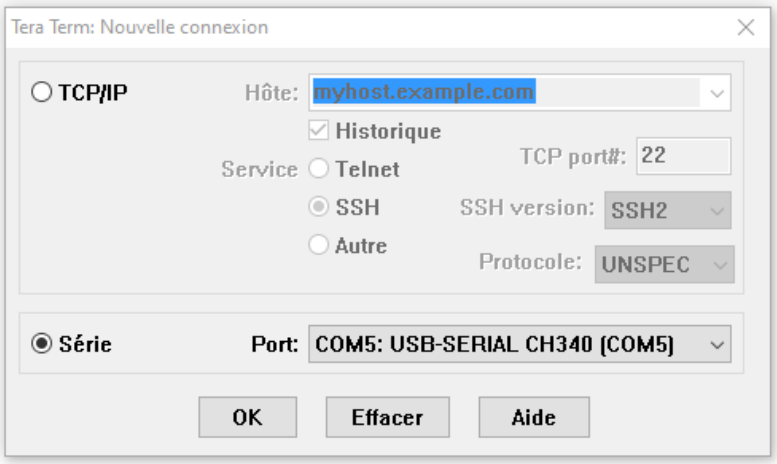
Usando el término Tera

Una vez configurado, cierre Tera Term.

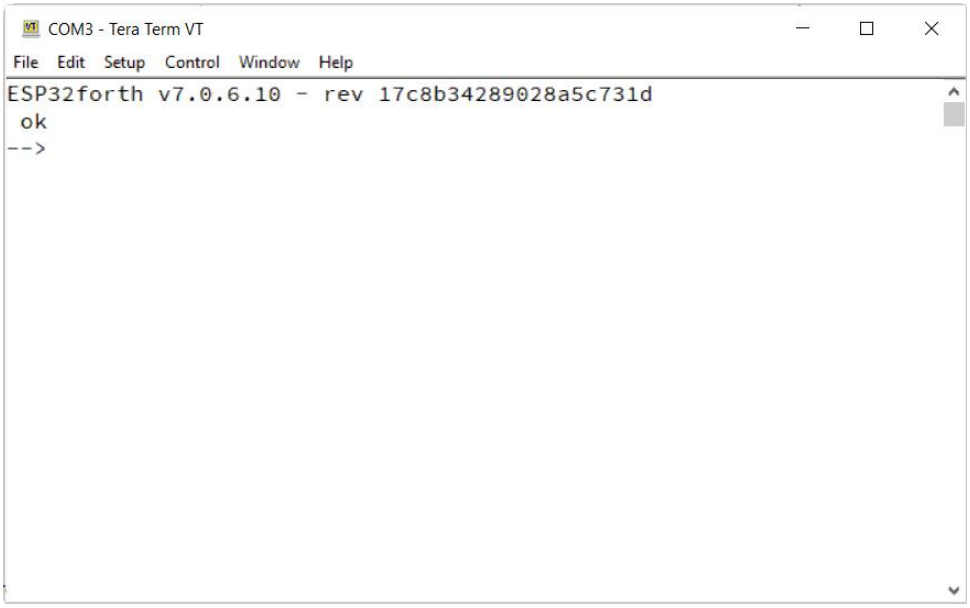
Conecte su placa ESP32 a un puerto USB disponible en su PC.

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

Seleccione el puerto serie :



Si todo ha ido bien deberías ver esto:

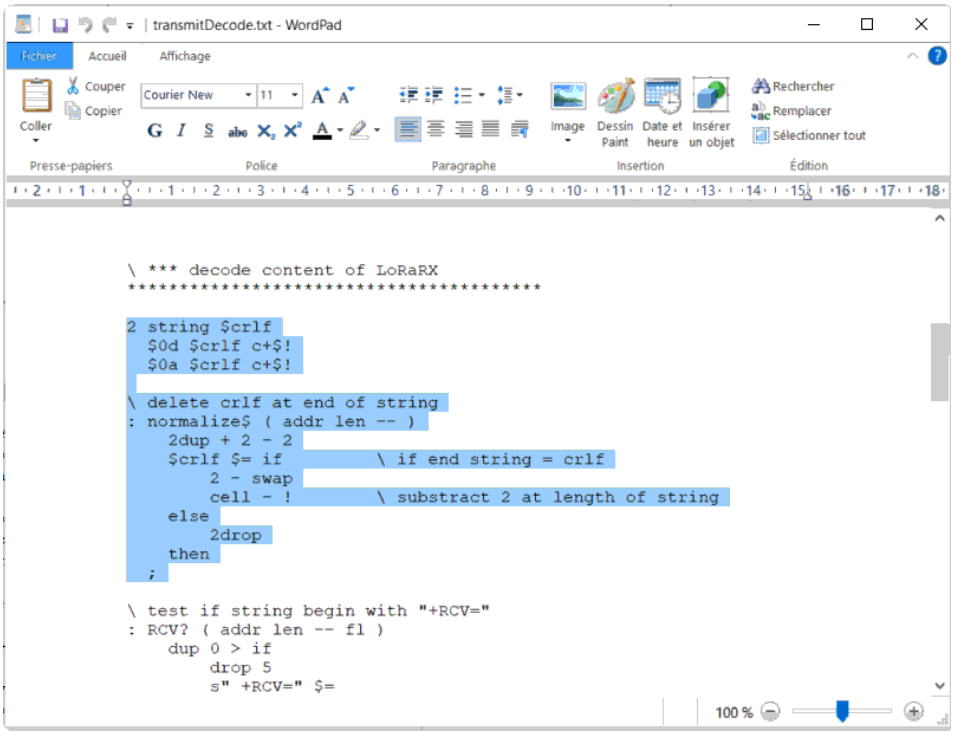


Compilar código fuente en lenguaje Forth

En primer lugar, ¡recordemos 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:

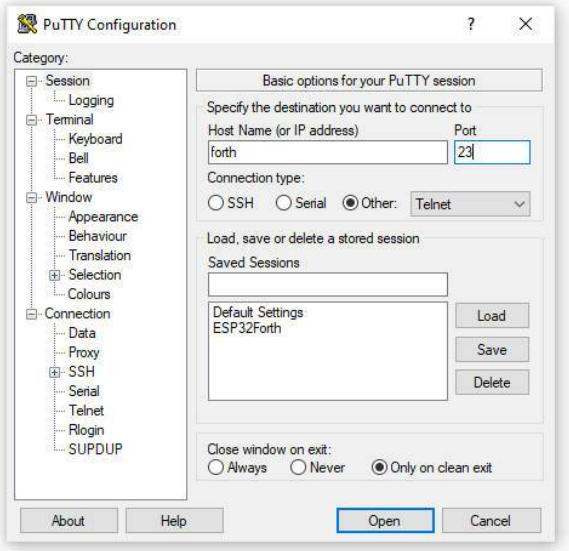


Figure 7: utiliser le nom DNS avec PuTTY

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` :

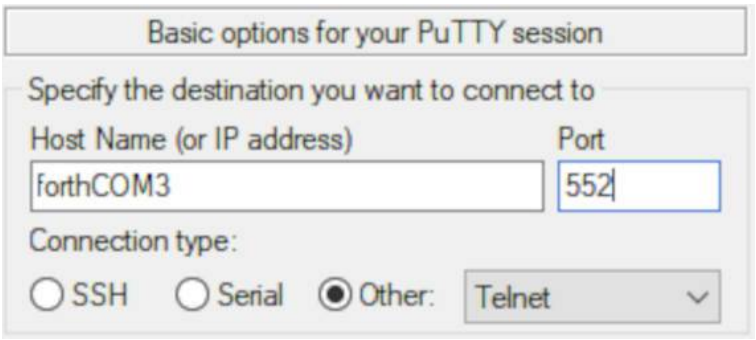


Figure 8: accès de PuTTY à forthCOM3

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:

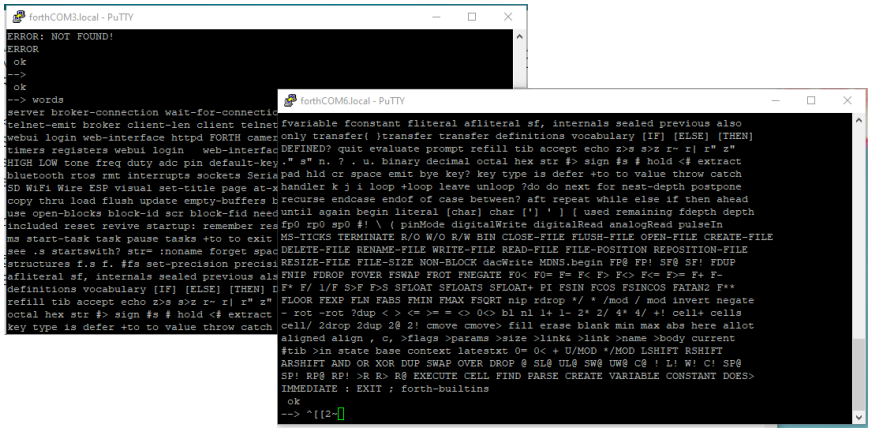


Figure 9: PuTTY accede a dos placas ESP32 separadas

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, ¡pero 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.

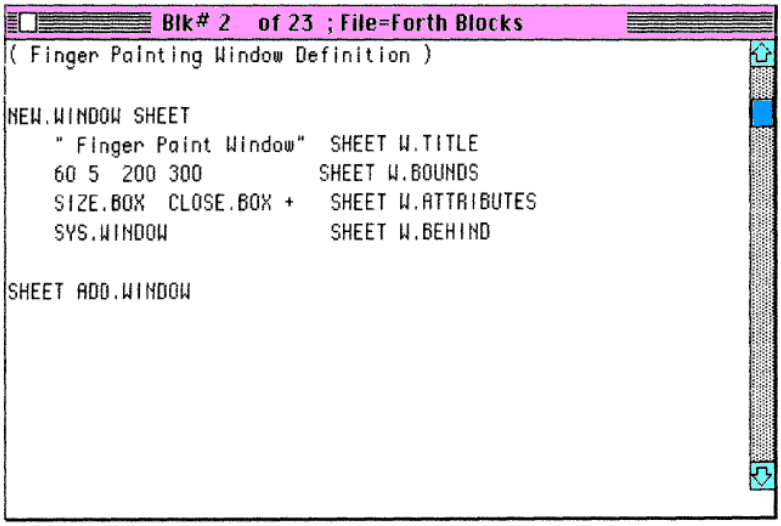
NOTA: Se puede acceder a las tarjetas ESP32 configuradas de esta forma desde Internet si la configuración del router WiFi lo permite.

Gestión de archivos fuente por bloques.

Utilice el `editor` solo para editar archivos de bloques en el sistema de archivos SPIFFS. Utilice `RECORDFILE` primero . Consulte el capítulo *RECORDFILE* .

Los bloques

Aquí un bloque en una computadora vieja:



Un bloque es un espacio de almacenamiento cuya unidad tiene 16 líneas de 64 caracteres. Por tanto, el tamaño de un bloque es 16x64=1024 bytes. ¡Es exactamente del tamaño de un kilobyte!

Abrir un archivo de bloque

Un archivo ya está abierto de forma predeterminada cuando se inicia ESP32forth.

archivo **blocks.fb** .

En caso de duda, ejecute `default-use` .

Para saber qué hay en este archivo, use los comandos del editor escribiendo primero `editor` .

Aquí están nuestros primeros comandos que debemos saber para administrar el contenido de los bloques:

- `l` enumera el contenido del bloque actual
- `n` selecciona el siguiente bloque

- `p` selecciona el bloque anterior

ATENCIÓN: un bloque siempre tiene un número entre 0 y n. Si termina con un número de bloque negativo, esto genera un error.

Editar el contenido de un bloque

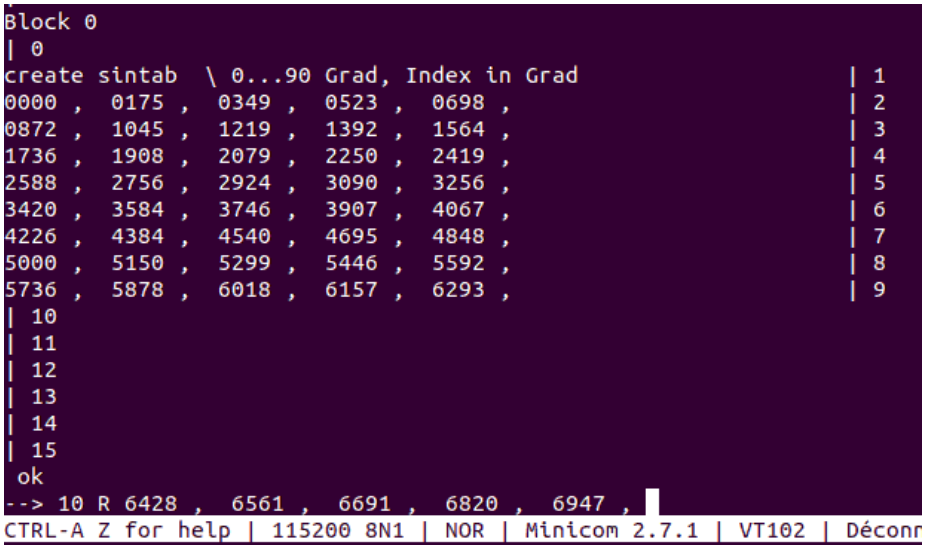
Ahora que sabemos cómo seleccionar un bloque en particular, veamos cómo insertar código fuente en el lenguaje FORTH...

Una estrategia es crear un archivo fuente en su computadora usando un editor de texto. Luego sólo necesitarás copiar/pegar tu código fuente por línea en los archivos de bloque.

Estos son los comandos esenciales para gestionar el contenido de un bloque:

- `wipe` vacía el contenido del bloque actual
- `d` elimina la línea n. El número de línea debe estar en el rango de 0 a 14. Las siguientes líneas se mueven hacia arriba. Ejemplo: `3 D` borra el contenido de la línea 3 y recupera el contenido de las líneas 4 a 15.
- `e` borra el contenido de la línea n. El número de línea debe estar en el rango de 0 a 15. Las otras líneas no suben.
- `a` inserta una línea n. El número de línea debe estar en el rango de 0 a 14. Las líneas ubicadas después de la línea insertada retroceden. Ejemplo: 3 Una prueba inserta la prueba en la línea 3 y baja el contenido de las líneas 4 a 15.
- `r` reemplaza el contenido de la línea n. Ejemplo: 3 R test reemplaza el contenido de la línea 3 con test

Aquí está nuestro bloque 0 actualmente en edición:



En la parte inferior de la pantalla, la línea `10 R 6428c, 6561c, .....` se está integrando en nuestro bloque en la línea 10.

Observa que la línea 0 no tiene contenido. Esto genera un error al compilar el código FORTH. Para solucionar este problema, simplemente escriba `0 R` seguido de dos espacios.

Con un poco de práctica, en unos minutos habrás insertado tu código FORTH en este bloque.

Haga lo mismo con los siguientes bloques si es necesario. Al pasar al siguiente bloque, fuerza que se guarde el contenido de los bloques escribiendo `flush` .

Compilando el contenido del bloque

Antes de compilar el contenido de un archivo de bloque, comprobaremos que su contenido está bien guardado. Por eso:

- escriba `flush` y luego desconecte la placa ESP32;
- espere unos segundos y vuelva a conectar la placa ESP32;
- `editor` de tipos y `l .` Debes encontrar tu bloque 0 con el contenido que editaste.

Para recopilar el contenido de tus bloques, tienes dos palabras:

- `load` precedido del número del bloque cuyo contenido queremos ejecutar y/o compilar. Para compilar el contenido de nuestro bloque 0, ejecutaremos `0 load` ;
- `thru` precedido por dos números de bloque ejecutará y/o compilará el contenido de los bloques como si estuviéramos ejecutando una sucesión de palabras `de load` . Ejemplo: `0 2 thru` ejecuta y/o compila el contenido de los bloques 0 a 2.

La velocidad de ejecución y/o compilación del contenido del bloque es casi instantánea.

Ejemplo práctico paso a paso

Veremos, con un ejemplo práctico, cómo insertar código fuente en el bloque 1. Cogemos un código listo para integrar en nuestro bloque:

```
1 list
editor
 0 r \ tools for REGISTERS definitions and manipulations
 1 r : mclr { mask addr -- }    addr @  mask invert and addr !  ;
 2 r : mset { mask addr -- }    addr @  mask or addr !  ;
 3 r : mtst { mask addr -- x }  addr @  mask and ;
 4 r : defREG: \ define a register, similar as constant
 5 r     create ( addr1 -- <name> )  ,
 6 r     does>  ( -- regAddr )       @  ;
 7 r : .reg ( reg -- ) \ display reg content
 8 r     base @ >r binary @ <#
 9 r     4 for aft 8 for aft # then next
10 r     bl hold  then next  #>
11 r     cr space ." 33222222 22221111 11111100 00000000"
12 r     cr space ." 10987654 32109876 54321098 76543210"
13 r     cr type  r> base !  ;
14 r : defMASK:  create ( mask0 position -- )    lshift ,
15 r     does> ( -- mask1 )              @  ;
save-buffers
```

Simplemente copie/pegue partes del código anterior y ejecute este código a través de ESP32 Forth:

- `1 list` para seleccionar y ver qué contiene el bloque 1
- `editor` para seleccionar `editor de vocabulario`
- copie las líneas `n r....` en paquetes de tres y ejecútelas
- `save-buffers` guarda el código en un archivo de bloque

Apague la placa ESP32. Reinícialo. Si escribe `1 list,` debería ver el código editado y guardado.

Para compilar este código, simplemente escriba `1 load`.

Conclusión

El espacio de archivo disponible para ESP32forth está cerca de 1,8 MB. Por lo tanto, puede administrar sin preocupaciones cientos de bloques de archivos fuente en el idioma FORTH.

Se recomienda instalar códigos fuente de partes de código estable. Así, durante la fase de desarrollo del programa, será mucho más fácil integrarlo en tu código en la fase de desarrollo:

```
2 5 thru \ integrate pwm commands for motors
```

en lugar de recargar sistemáticamente este código a través de una línea serie o WiFi.

La otra ventaja de los bloques es que permiten la incorporación in situ de parámetros, tablas de datos, etc. que luego pueden ser utilizados por sus programas.

Edición de archivos fuente con VISUAL Editor

Utilice **`visual edit`** solo para editar archivos fuente en el sistema de archivos SPIFFS.
Utilice **`RECORDFILE`** primero . Consulte el capítulo *RECORDFILE*.

Editar un archivo fuente FORTH

Para editar un archivo fuente FORTH con ESP32forth, usaremos el editor **`visual`** .

Para editar un archivo **`dump.fs`**, proceda así desde el terminal conectado a una tarjeta ESP32 que contenga ESP32forth:

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

`DUMP` completo está disponible aquí:

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

La palabra **`edit`** va seguida del directorio donde se almacenan los archivos fuente:

- Si el archivo no existe, se crea;
- si el archivo existe, se recupera en el editor.

Anote el nombre del archivo que creó.

fs como extensión de archivo, para **F**orth **S**ource.

Editando el código FORTH

En el editor, mueva el cursor con las flechas izquierda-derecha-arriba-abajo disponibles en el teclado.

El terminal actualiza la pantalla cada vez que se mueve el cursor o se modifica el código fuente.

Para salir del editor:

- CTRL-S: guarda el contenido del archivo que se está editando actualmente
- CTRL-X: sale de la edición:
  - N: sin guardar los cambios del archivo
  - Y: con guardado de cambios

Compilando el contenido del archivo

La compilación del contenido de nuestro archivo **dump.fs** se realiza así:

```
include /spiffs/dump.fs
```

La compilación es mucho más rápida que a través del terminal.

Los archivos fuente integrados en la tarjeta ESP32 con ESP32 en adelante son persistentes. Después de apagar la alimentación y volver a conectar la tarjeta ESP32, el archivo guardado permanece disponible inmediatamente.

Puede definir tantos archivos como sea necesario.

Por lo tanto, es fácil integrar en la tarjeta ESP32 una colección de herramientas y rutinas de las que puede extraer según sea necesario.

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 **/sipffs/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

encontrar todas las definiciones de palabras del proyecto sin necesidad de una nueva transferencia a través del terminal.

La noción de caja negra

Es un concepto antiguo, que data de la época en que desarrollábamos principalmente en ensamblador tarjetas de microcontroladores. También lo encontramos con clases de programación de objetos. En el concepto de "caja negra", debemos considerar una subrutina, una función, un método como una caja negra. Sabemos lo que ponemos allí. Sabemos qué puede salir de ella o cómo funciona esta caja, pero no nos preocupamos de su funcionamiento interno. Confiamos en quienes programaron la "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.

Mi consejo para marcar código verificado es simplemente poner una línea de comentario justo antes de la definición. Ejemplo de código no verificado:

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

El código se probará en un *entorno limitado* o en un archivo de proyecto. Poco importa. Una vez probado con diferentes valores, modifico el código fuente:

```
\ multiplicar fn por pi
: fpi* ( f
    pi f*
  ;
```

Si dudas de la

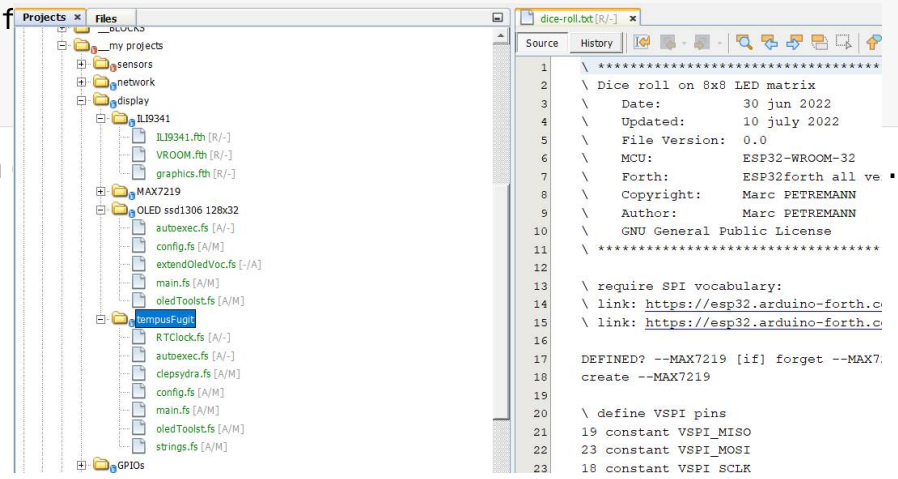


Figure 10: Estructuración de proyectos con el IDE de Netbeans

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/MPETREMANN11/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` .

El sistema de archivos SPIFFS

ESP32Forth contiene un sistema de archivos rudimentario en la memoria Flash interna. Se puede acceder a los archivos a través de una interfaz en serie llamada SPIFFS para Serial Peripheral Interface Flash File System.

Si bien el sistema de archivos SPIFFS es sencillo, aumenta considerablemente la flexibilidad de tus desarrollos con ESP32Forth:

- administrar archivos de configuración
- integrar extensiones de software accesibles bajo petición
- modularizar los desarrollos en módulos funcionales reutilizables

Y muchos otros usos que te dejaremos descubrir...

Acceso al sistema de archivos SPIFFS

Para compilar el contenido de un archivo fuente editado mediante edición visual, escriba:

```
include /spiffs/dumpTool.fs
```

La palabra `incluir` siempre debe utilizarse desde el terminal.

Para ver la lista de archivos SPIFFS , utilice la palabra `ls` :

```
ls /spiffs/
\ cartel:
\ dumpTool.fs
```

Aquí, se ha guardado el archivo **dumpTool.fs .** Para SPIFFS, las extensiones de archivo son irrelevantes. Los nombres de archivos no deben contener caracteres de espacio ni el carácter /.

Editemos y guardemos un nuevo archivo **myApp.fs** con `editor visual`. Ejecutemos `ls` :

```
ls /spiffs/
\ display:
\ dumpTool.fs
\ myApp.fs
```

El sistema de archivos SPIFFS no administra subcarpetas como en una computadora con Linux. Para crear un pseudodirectorio , simplemente indíquelo al crear un nuevo archivo.

Por ejemplo, editemos el archivo **other/myTest.fs** . Una vez editado y guardado, ejecutemos `ls` :

```
ls /spiffs/
\ affiche:
\ dumpTool.fs
\ myApp.fs
\ other/myTest.fs
```

Si desea ver solo los archivos en este **other**, debe seguir **/spiffs/** con el nombre de este pseudodirectorio:

```
ls /spiffs/other
\ affiche:
\ myTest.fs
```

No hay ninguna opción para filtrar nombres de archivos o pseudodirectorios.

Manejo de archivos

Para eliminar completamente un archivo, utilice la palabra `rm` seguida del nombre del archivo que desea eliminar:

```
rm /spiffs/other/myTest.fs
ls /spiffs/
\ affiche:
\ dumpTool.fs
\ myApp.fs
```

Para cambiar el nombre de un archivo, use la palabra `mv` :

```
mv /spiffs/myApp.fs /spiffs/main.fs
ls /spiffs/
\ affiche:
\ dumpTool.fs
\ main.fs
```

Para copiar un archivo, use la palabra `cp` :

```
cp /spiffs/main.fs /spiffs/mainTest.fs
ls /spiffs/
\ affiche:
\ dumpTool.fs
\ main.fs
\ mainTest.fs
```

Para ver el contenido de un archivo, utilice la palabra `cat` :

```
cat /spiffs/dumpTool.fs
```

```
\ muestra el contenido de dumpTool.fs
```

Para guardar el contenido de una cadena en un archivo, actúe en dos fases:

- crear un nuevo archivo con **touch**
- guardar el contenido de la cadena con **dump-file**

```
touch /spiffs/mTest,fs  \ crée nouveau fichier mtest,fs
ls /spiffs/             \ affiche:
\ dumpTool.fs
\ main.fs
\ mainTest.fs
\ mTests

\ enregistre chaîne "Insère mon texte dans mTest" dans mTest
r| ." Insère mon texte dans mTest" |   s" /spiffs/mTest"   dump-file

include /spiffs/mTest   \ affiche: Insert my text in mTest
```

Organiza y compila tus archivos en la tarjeta ESP32

Veremos cómo administrar archivos para una aplicación que se está desarrollando en una placa ESP32 con ESP32forth instalado.

Se acuerda que todos los archivos utilizados estén en formato de texto ASCII.

Las siguientes explicaciones se dan sólo como consejo. Provienen de cierta experiencia y tienen como objetivo facilitar el desarrollo de grandes aplicaciones con ESP32 en adelante.

Edición y transmisión de archivos fuente.

Todos los archivos fuente de su proyecto están en su computadora. Es recomendable tener una subcarpeta dedicada a este proyecto. Por ejemplo, está trabajando en una pantalla OLED SSD1306. Entonces crea un directorio llamado SSD1306.

En cuanto a las extensiones de nombres de archivos, recomendamos utilizar la extensión **fs** .

La edición de archivos en una computadora se realiza con cualquier editor de archivos de texto.

En estos archivos fuente, no utilice ningún carácter que no esté incluido en el código ASCII. Algunos códigos extendidos pueden interrumpir la compilación del programa.

Estos archivos fuente luego se copiarán o transferirán a la tarjeta ESP32 a través del enlace serie y un programa tipo terminal:

- copiando/pegando usando visual en ESP32 en adelante, para reservarlo para archivos pequeños;
- con un procedimiento específico que se detallará más adelante para archivos importantes.

Organiza tus archivos

En lo sucesivo, todos nuestros archivos tendrán la extensión **fs** .

Comencemos desde nuestro directorio SSD1306 en nuestra computadora.

que crearemos en este directorio será el archivo **main.fs.** Este archivo contendrá las llamadas para cargar todos los demás archivos de nuestra aplicación en desarrollo.

Ejemplo de contenido de nuestro archivo **main.fs** :

```
\ Pruebas de visualización y desarrollo OLED SSD1306 128x32
s" /spiffs/config.fs" included
```

En la fase de desarrollo, el contenido de este archivo **main.fs** se cargará manualmente ejecutando `include` de esta manera:

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

Esto hace que se ejecute el contenido de nuestro archivo **main.fs.** La carga de otros archivos se ejecutará desde este archivo **main.fs.** Aquí cargamos el archivo **config.fs** del cual aquí hay un extracto:

```
\ ***********************************************
*********************
\ Configuración para pantalla OLED SSD1306 128x32
\ ***********************************************
*********************

\ pour SSD1306_128_32
    128 constant SSD1306_LCDWIDTH
     32 constant SSD1306_LCDHEIGHT
```

En este archivo **config.fs** pondremos todos los valores constantes y varios parámetros utilizados por los demás archivos.

Nuestro próximo archivo será **SSD10306commands.fs** . Aquí se explica cómo cargar su contenido desde **main.fs**:

```
\ Pruebas de visualización y desarrollo OLED SSD1306 128x32
s" /spiffs/config.fs" included
s" /spiffs/SSD10306commands.fs" included
```

El contenido del archivo **SSD10306commands.fs** tiene casi 230 líneas de código. No es posible copiar el contenido de este archivo línea por línea en el editor visual ESP32forth. Aquí hay un método para copiar y guardar el contenido de este archivo grande en ESP32 de una sola vez.

Conclusión

Los archivos guardados en el sistema de archivos ESP32forth SPIFFS están disponibles permanentemente.

Si saca la placa ESP32 de servicio y luego la vuelve a conectar, los archivos estarán disponibles inmediatamente.

El contenido de los archivos se puede modificar in situ con `visual edit`.

Esta comodidad hará que los desarrollos sean mucho más rápidos y sencillos.

Edición y gestión de archivos fuente para ESP32forth

Como ocurre con la gran mayoría de los lenguajes de programación, los archivos fuente escritos en el lenguaje FORTH están en formato de texto simple. La extensión de archivos en el idioma ADELANTE es gratuita:

- **txt** extensión genérica para todos los archivos de texto;
- **forth** utilizado por algunos programadores de FORTH;
- **fth** forma comprimida para FORTH;
- **4th** otra forma comprimida para FORTH;
- **fs** nuestra extensión favorita…

Editores de archivos de texto

En Windows, el editor **edit** de edición de archivos es el más sencillo:

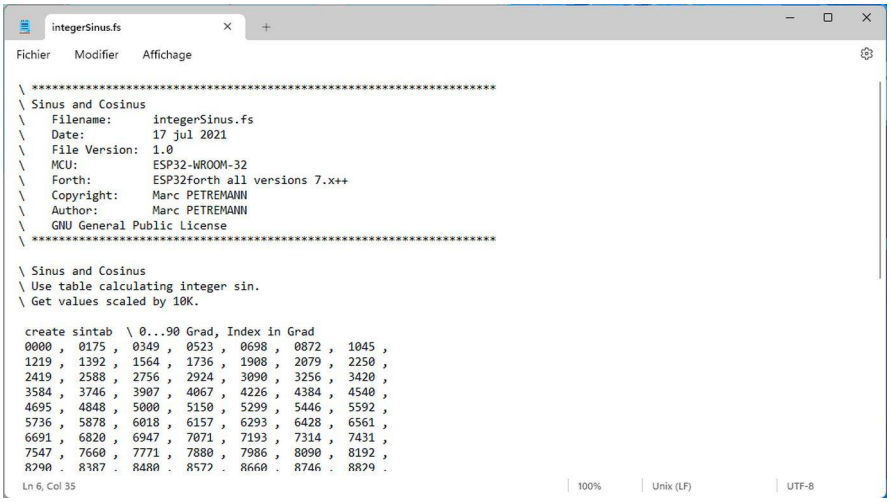


Figure 11: edición con edición en Windows 11

No se recomiendan otros editores, como **WordPad**, porque corre el riesgo de guardar el código fuente del idioma FORTH en un formato de archivo que no es compatible con ESP32forth.

En Linux, el equivalente se llama **gEdit**. MacOS también tiene un editor de texto sencillo.

Si utiliza una extensión de archivo personalizada, como **fs** , para sus archivos fuente de idioma FORTH, su sistema debe reconocer esta extensión de archivo para permitir que el editor de texto los abra.

Utilice un IDE

Nada le impide utilizar un IDE [4]. Por mi parte, tengo preferencia por **Netbeans** que también uso para PHP, MySQL, Javascript, C, ensamblador… Es un IDE muy potente y tan eficiente como **Eclipse** :

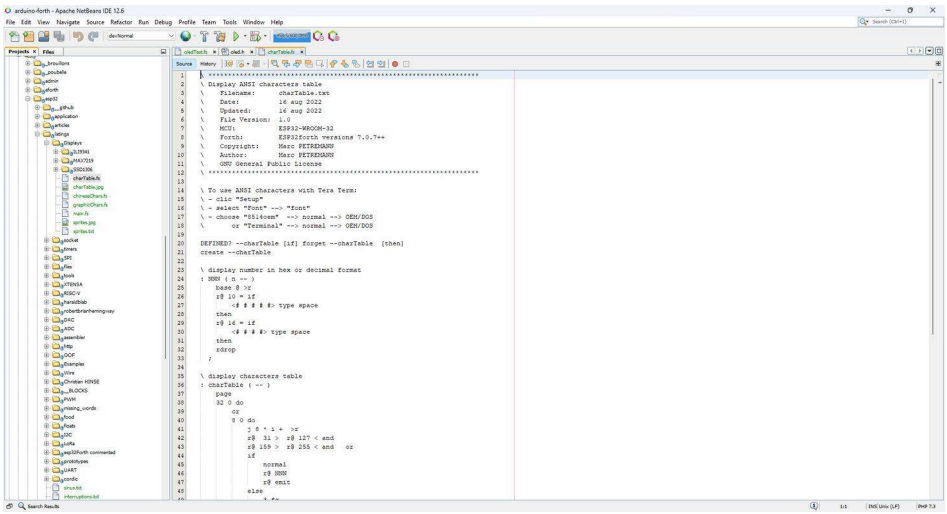


Figure 12: editando con Netbeans

Netbeans ofrece varias características interesantes:

- gestión de versiones con **GIT** ;
- recuperación de versiones anteriores de archivos modificados;
- comparación de archivos con **Diff** ;
- **FTP** con un solo clic al alojamiento en línea de su elección;

Opción **GIT** , posibilidad de compartir archivos en un repositorio y gestionar colaboraciones en proyectos complejos. De forma local o colaborativa, **GIT** le permite gestionar diferentes versiones del mismo proyecto y luego fusionar estas versiones. Puede crear su repositorio GIT local. Cada *vez que* se confirma un archivo o un directorio completo, los desarrollos se mantienen tal cual. Esto le permite encontrar versiones antiguas del mismo archivo o carpeta de archivos.

4 Entorno de desarrollo integrado

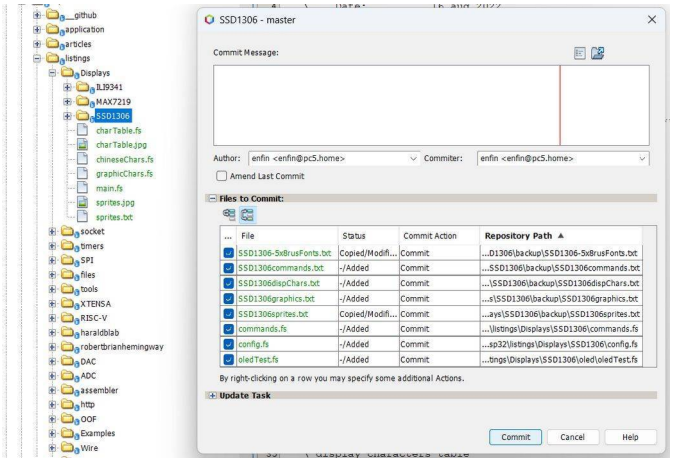


Figure 13: Operación GIT en Netbeans

Con NetBeans, puedes definir una rama de desarrollo para un proyecto complejo. Aquí creamos una nueva rama:

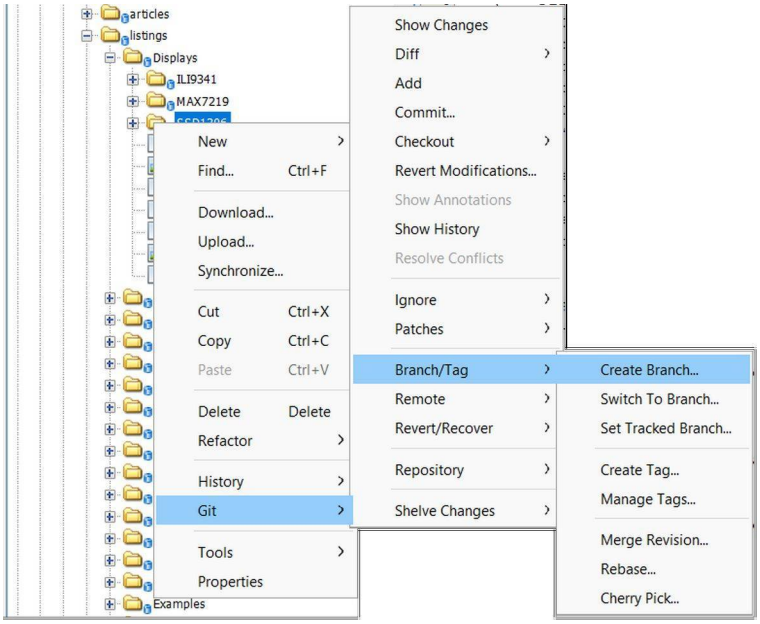


Figure 14: creando una rama en un proyecto

Ejemplo de situación que justifica la creación de una sucursal:

- tienes un proyecto funcional;
- planeas optimizarlo;
- crea una rama y haz las optimizaciones en esta rama...

Los cambios a los archivos fuente en una rama no tienen influencia en los archivos en el tronco *main* .

Por cierto, es más que recomendable disponer de medios físicos de copia de seguridad. Un disco duro SSD cuesta alrededor de 50€ por 300 Gb de espacio de almacenamiento. ¡La velocidad de acceso de lectura o escritura de los medios SSD es simplemente asombrosa!

Almacenamiento en GitHub

Sitio web **de GitHub**[5]es, junto con **SourceForge**[6], uno de los mejores lugares para almacenar archivos fuente. En GitHub, puedes compartir una carpeta de trabajo con otros desarrolladores y gestionar proyectos complejos. El editor de Netbeans puede conectarse al proyecto y le permite pasar o recuperar cambios de archivos.

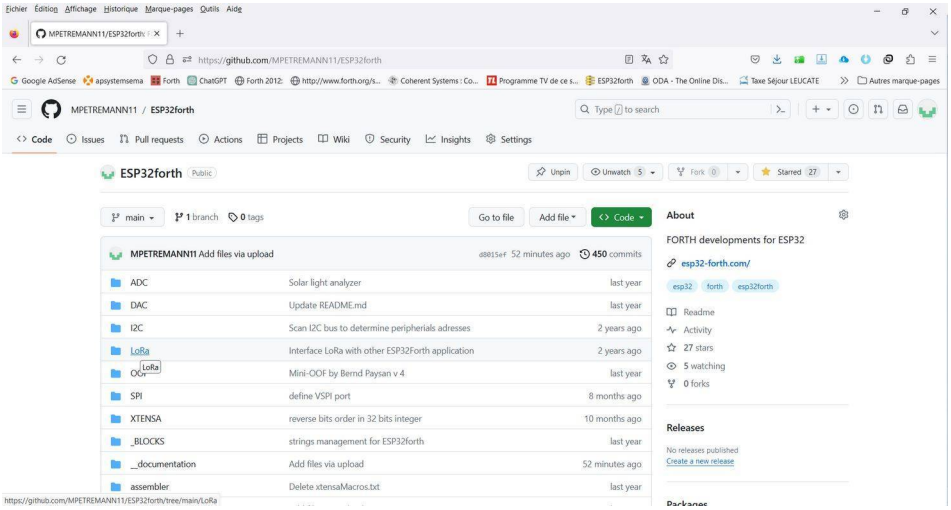


Figure 15: almacenar archivos en Github

En **GitHub** , puedes gestionar *las bifurcaciones de proyectos* . También puedes hacer que ciertas partes de tus proyectos sean confidenciales. Aquí las ramas en los proyectos flagxor/ueforth:

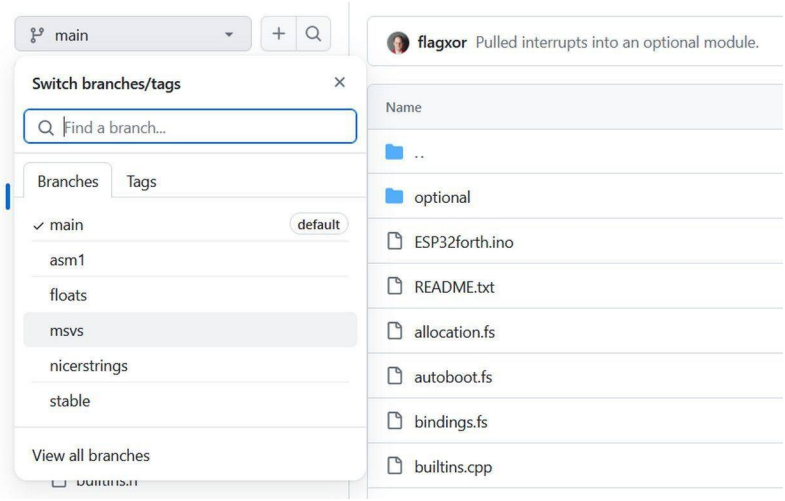


Figure 16: acceso a una rama del proyecto

Algunas buenas practicas

La primera buena práctica es nombrar correctamente los archivos y carpetas de trabajo. Estás desarrollando para ESP32Forth, así que crea una carpeta llamada **ESP32forth** .

Para varias pruebas, cree una subcarpeta **sandbox** zona de pruebas en esta carpeta .

5 https://github.com/
6 https://sourceforge.net/

Para proyectos bien construidos, cree una carpeta por proyecto. Por ejemplo, si desea controlar un robot, cree una subcarpeta **robot** .

Si tiene scripts de uso general, cree una carpeta **tools** . Si está utilizando un archivo de esta carpeta **tools** en un proyecto, copie y pegue ese archivo en la carpeta de ese proyecto. Esto evitará que una modificación de un archivo en **tools** interrumpa posteriormente su proyecto.

La segunda mejor práctica es distribuir el código fuente de un proyecto en varios archivos:

- **config.fs** para almacenar la configuración del proyecto;
- directorio **documentation** para almacenar archivos en el formato de su elección, relacionados con la documentación del proyecto;
- **myApp.fs** para las definiciones de su proyecto. Elija un nombre de archivo bastante explícito. Por ejemplo, para administrar un robot, tome el nombre **robot-commands.fs** .

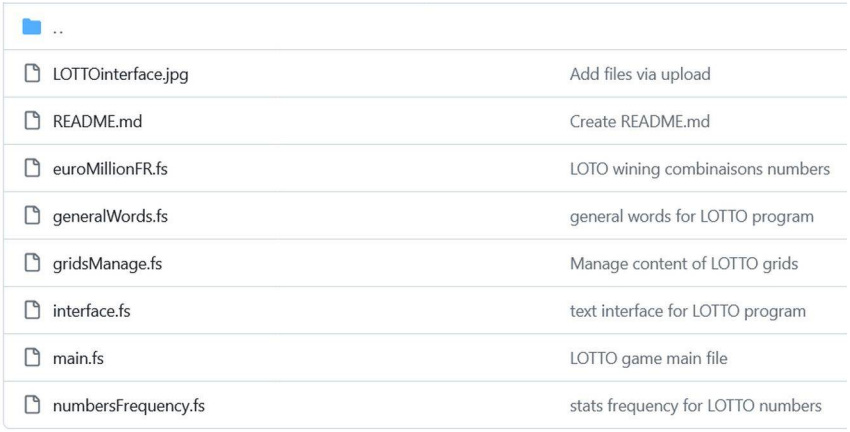


Figure 17: Cuarto ejemplo de nomenclatura de archivo Forth

Es el contenido de estos archivos el que debe transferirse a través del terminal a la tarjeta ESP32 para que ESP32forth interprete y compile el código FORTH.

El archivo main.fs

ESP32forth gestiona un sistema de archivos SPIFFS [7]. Consulte el capítulo *El sistema de archivos SPIFFS* .

Por lo tanto, estos archivos se almacenan en la tarjeta ESP32 y pueden ser leídos por ESP32 en adelante. Si escribió un archivo **config.fs** en el sistema de archivos SPIFFS, aquí está la línea de código que debe escribir en **main.fs** para acceder al contenido de **config.fs** :

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

A partir de este momento, tiene dos opciones para interpretar el contenido de **config.fs** . Desde la terminal:

7 Sistema de archivos Flash de interfaz periférica serie

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

O

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

El caso es que **main.fs** puede llamar a otros archivos. Ejemplo :

```
\ Pruebas de visualización y desarrollo OLED SSD1306 128x32
s" /spiffs/config.fs" included
s" /spiffs/SSD10306commands.fs" included
```

Procesar muchos archivos lleva menos de un segundo. Esta estrategia evita la transmisión repetida del código fuente por enlace serie a través del terminal.

Y cuando administras múltiples proyectos en múltiples placas ESP32, es más fácil probar cada proyecto con un simple comando `include /spiffs/main.fs` .

La carga de archivos se gestionará entonces con RECORDFILE. Consulte el capítulo denominado *RECORDFILE* .

Gestionar un semáforo con ESP32

Puertos GPIO en la placa ESP32

Los puertos GPIO (General Purpose Input/Output) son puertos de entrada-salida muy utilizados en el mundo de los microcontroladores.

El chip ESP32 viene con 48 pines que tienen múltiples funciones. No todos los pines se utilizan en las placas de desarrollo ESP32 y algunos pines no se pueden utilizar.

Hay muchas preguntas sobre cómo utilizar los GPIO ESP32. ¿Qué conectores debería utilizar? ¿Qué conectores deberías evitar utilizar en tus proyectos?

Si miramos con lupa una tarjeta ESP32, vemos esto:

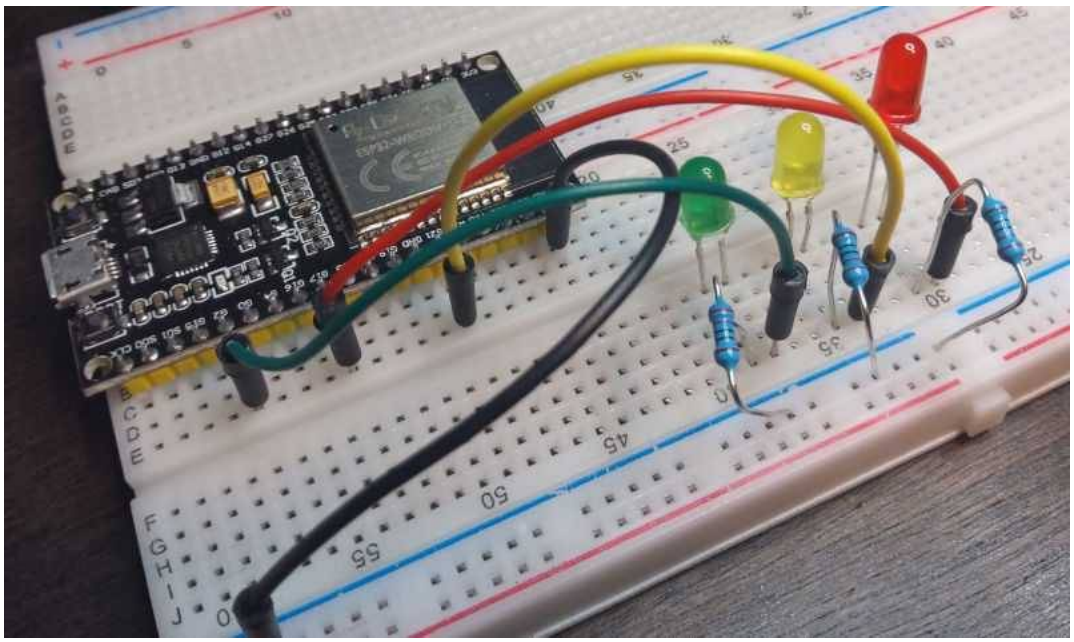
Cada conector está identificado por una serie de letras y números, aquí de izquierda a derecha en nuestra foto: G22 TXD RXD G21 GND G19 G18, etc...

Los conectores que nos interesan para este manejo están precedidos de la letra G seguida de uno o dos números. Por ejemplo, G2 corresponde a GPIO 2.

Definir y operar un conector GPIO en modo de salida es bastante simple.

Montaje de los LED

El montaje es bastante sencillo y sólo basta con una foto:



- LED verde conectado a G2 - cable verde
- LED amarillo conectado a G21 - cable amarillo
- LED rojo conectado a G17 - cable rojo
- cable negro conectado a GND

Nuestro código utiliza la palabra **incluir** seguida del archivo a cargar.

Definimos nuestros LED con **defPin:**

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

 2 defPIN: ledGREEN
21 defPIN: ledYELLOW
17 defPIN: ledRED

: LEDinit
    ledGREEN    output pinMode
    ledYELLOW   output pinMode
    ledRED      output pinMode
  ;
```

Muchos programadores tienen la mala costumbre de nombrar los conectores por su número. Ejemplo :

```
17 defPin: pin17
```

O

```
17 defPin: GPIO17
```

Para que sea efectivo, debes nombrar los conectores por su función. Aquí definimos los conectores **`ledRED`** o **`ledGREEN`** .

¿Por qué? Porque el día que necesites agregar accesorios y liberar por ejemplo el conector G21, simplemente redefine **`21 defPIN: ledYELLOW`** con el nuevo número de conector. El resto del código no se modificará y podrá utilizarse.

Gestión de semáforos.

Aquí está la parte del código que controla nuestros LED en nuestra simulación de semáforo:

```
\ trafficLights ejecuta un ciclo de luz
: trafficLights ( ---)
    high ledGREEN   pin     3000 ms    low ledGREEN   pin
    high ledYELLOW  pin      800 ms    low ledYELLOW  pin
    high ledRED     pin     3000 ms    low ledRED     pin
    ;

\ bucle de semáforo clásico
: lightsLoop ( ---)
    LEDinit
    begin
        trafficLights
    key? until
    ;

\ estilo semáforo alemán
: Dtraffic ( ---)
    high ledGREEN   pin    3000 ms    low ledGREEN   pin
    high ledYELLOW  pin     800 ms    low ledYELLOW  pin
    high ledRED     pin    3000 ms
    ledYELLOW  high    800 ms
    \ simultaneous red and yellow ON
    high ledRED     pin  \ simultaneous red and yellow OFF
    high ledYELLOW  pin
    ;

\ bucle de semáforo alemán
: DlightsLoop ( ---)
    LEDinit
    begin
```

```
        Dtraffic
    key? until
    ;
```

Conclusión

Este programa de gestión de semáforos perfectamente podría haber sido escrito en lenguaje C. Pero la ventaja del lenguaje FORTH es que da control, a través del terminal, para analizar, depurar y modificar funciones muy rápidamente (en FORTH decimos palabras).

Administrar los semáforos es un ejercicio fácil en lenguaje C. Pero cuando los programas se vuelven un poco más complejos, el proceso de compilación y carga rápidamente se vuelve tedioso.

Simplemente actúe a través de la terminal y simplemente copie/pegue cualquier fragmento del código del lenguaje FORTH para que sea compilado y/o ejecutado.

Si está utilizando un programa de terminal para comunicarse con la placa ESP32, simplemente escriba `DlightsLoop` o `LightsLoop` para probar cómo funciona el programa. Estas palabras utilizan un bucle condicional. Simplemente presione una tecla en el teclado y la palabra dejará de reproducirse al final del bucle.

Acceso directo a los registros GPIO

En algunas situaciones, es mucho más beneficioso tener acceso directo a los registros GPIO en la placa ESP32. Por ejemplo, para gestionar secuencias complejas de activación o desactivación.

¡Con ESP32 en adelante, el acceso a un registro ESP32 se realiza usando m palabras `!` y M `@` . Estas palabras permiten en particular el acceso directo a los registros GPIO.

El registro GPIO que gestiona las primeras 32 entradas/salidas se encuentra en la dirección hexadecimal 3ff44004:

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

Si conectamos un LED al puerto GPIO2, podemos encenderlo y apagarlo así:

```
0 GPIO_OUT_REG m!   \ apagar el LED encendido G2
4 GPIO_OUT_REG m!   \ enciende el LED en G2
```

¡La desventaja, en secuencia **`4 GPIO_OUT_REG m!`** , esto es lo que sólo activa el puerto G2. Si otros puertos estuvieran activos, se desactivarán. Por lo tanto, la solución que se me ocurre sería leer el estado del registro **`GPIO_OUT_REG`** usando `m@` y realizar operaciones lógicas con el valor antes de reinyectarlo con `m!` .

Resulta que la tarjeta ESP32 tiene registros dedicados que realizan estas operaciones selectivas de activación y desactivación sin pasar por estas operaciones lógicas en este registro **`GPIO_OUT_REG`**.

Uso de palabras m! y m@

Estas dos palabras se definen en el vocabulario **`de registers`**. Estas son también las únicas palabras definidas en este vocabulario:

- `m!` (val shift mask addr -)
  modifica el contenido de un registro señalado por **`addr`** , aplica una máscara lógica con `mask` y desplaza **`val en n bits`** según el **`shift`** ;
- `m@` (shift mask addr - val)
  lee el contenido de un registro señalado por **`addr`** , aplica una máscara lógica con **`mask`** y cambia n bits según el **`shift`** .

Para comprender completamente cómo funcionan estas dos palabras, primero definiremos una palabra que muestra el contenido de 32 bits de cualquier dirección:

```
\ mostrar n en formato bbbbbbbb bbb.....
: .binDisp ( n -- )
    base @ >r  binary   \ select binary base
    <#                  \ start num formating
```

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

Definamos cualquier espacio de memoria, inicializado con un valor cero:

```
create myReg
    0 ,
```

A continuación se explica cómo mostrar el contenido de **myReg** con **.binDisp** :

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

El bit b22 ha sido resaltado. Aquí está en cero. ATENCIÓN: los bits están numerados del 0 al 31 en la visualización representada por **.binDisp** . A continuación se explica cómo configurar este único bit b22 en uno:

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

Está hecho. Pero tomamos el valor de 32 bits **$ffffffff** como máscara .
Lamentablemente, la elección de esta máscara permite una acción sobre todos los bits del contenido de **myReg** . Si queremos actuar sólo sobre uno o más bits, debemos elegir una máscara que limite la acción de la palabra **m!**. Por ejemplo, para establecer el bit b07 en 1, necesitará utilizar una máscara. Aquí está esta máscara en binario:

```
00000000 00000000 00000000 10000000
```

Lo que se traduce en hexadecimal a **$00000080** . Si tienes dudas, puedes consultar con **.binDisp** :

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

Esta máscara binaria corresponde al bit b07. Ahora pondremos este bit a 1 en **myReg** sin modificar el otro bit b22 ya puesto a uno:

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

Entre los parámetros necesarios para **m!** , tenemos el par shift addr. Para simplificar el código ESP32forth, crearemos una palabra **defMASK:**:

```
\ definir una máscara para los registros
: defMASK: ( comp: mask0 position -- <name> | exec: -- position
mask1 )
    create
        dup ,
        lshift ,
    does>
        dup @
        swap cell + @
  ;
```

Para definir una máscara, sólo necesitas dos parámetros:

- **mask0** : 1 es la máscara mínima en un bit, 3 en 2 bits, 7 en 3 bits, etc...
- **position** : indica la posición, en el intervalo [0..31]

Para modificar, por ejemplo, el bit b12, podemos definir nuestra máscara de la siguiente manera:

```
1 12 defMASK: mB12
```

Para establecer este único bit b12 en 1:

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

Para restablecer este bit b12:

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

Máscaras definidas por `defMASK:` son aplicables a cualquier registro. Esto es lo que verás más adelante. Estas máscaras también son muy útiles para determinar el estado de uno o varios bits específicos. Probemos el estado de nuestro bit b12 con `m@` :

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

Restablezcamos este bit b12 a 1 y probémoslo nuevamente:

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

Ya tenemos las claves para actuar poco a poco sobre el contenido de cualquier registro. En este capítulo, nos centraremos particularmente en el registro GPIO_OUT_REG:

```
\ Registro de salida GPIO 0-31 R/W
$3FF44004 defREG: GPIO_OUT_REG
```

El registro GPIO_OUT_REG

Este registro le permite controlar los puertos GPIO G0 a G31. Cablearemos tres diodos de colores en los pines G25, G26 y G27. Por tanto, definimos tres constantes adjuntas a estos pines:

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

En el capítulo *Gestionar un semáforo con ESP32* , definimos estas mismas palabras con `defPIN:`. Aquí lo hacemos con `constant`, que tiene el mismo comportamiento. Usaremos estas constantes para definir las máscaras:

```
\ definir máscaras para LED rojos, amarillos y verdes
1 ledRED     defMASK: mLED_RED
1 ledYELLOW  defMASK: mLED_YELLOW
1 ledGREEN   defMASK: mLED_GREEN
```

Para activar el LED rojo en G25, ingrese:

```
registers
```

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

Para evitar la selección recurrente del vocabulario `de registers`, definimos dos palabras que nos simplificarán la programación:

```
\ establecer máscara en dirección
: regSet ( val shift mask addr -- )
    [ registers ] m! [ forth ]
  ;

\ máscara de prueba en dirección
: regTst ( shift mask addr -- val )
    [ registers ] m@ [ forth ]
  ;
```

Activación del LED rojo en G25:

```
1 mLED_RED GPIO_OUT_REG regSet
```

Esto funcionará, siempre que los pines GPIO estén inicializados correctamente. Esto es lo que vamos a discutir.

Registros de activación y desactivación

Los nombres de los registros se toman de la documentación *del Manual de referencia técnica de ESP32* (pdf): https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf

Antes de encender y apagar nuestros LED conectados a los puertos GPIO G25 G26 y G27, comenzaremos inicializando estos puertos. Esto se hace actuando sobre el registro `GPIO_ENABLE_REG`:

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

Al ejecutar la palabra `GPIO.init` se inicializan los puertos de salida G25, G26 y G27. Probemos la iluminación de los LED:

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

Si los LED están conectados correctamente, deberían encenderse. Aquí el encendido de los LED se realiza de forma secuencial mediante la palabra `regSet` repetida tres veces.

Actuando directamente sobre el contenido del registro **GPIO_OUT_REG** , podemos encender los tres LED en una sola ejecución de **regSet** :

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

Veamos cómo utilizar los dos registros **GPIO_OUT_W1TS_REG** y **GPIO_OUT_W1TC_REG** para actuar indirectamente sobre el estado de uno o más puertos GPIO:

- **GPIO_OUT_W1TS_REG** : *GPIO Output Write to Set Register* se utiliza para configurar los bits correspondientes a los pines GPIO en el modo "Salida" en lógica alta (1). Le permite activar las salidas GPIO especificadas, configurando los bits correspondientes en 1.
- **GPIO_OUT_W1TC_REG** : *GPIO Output Write to Clear Register* se utiliza para borrar (cero) bits específicos en el registro de salida GPIO. Cada bit de este registro está asociado con un pin GPIO particular, y **al escribir un 1** en un bit determinado de este registro, puede poner a cero (borrar) la salida correspondiente del pin GPIO.

Por lo tanto, podemos encender todos nuestros LED en una única secuencia **regSet** como esta:

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

Y para apagar todos los LED en una sola secuencia **regSet** , ejecutaremos esto:

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

Si queremos gestionar un ciclo de encendido y apagado temporizado de un LED crearemos una palabra que gestione un ciclo de encendido y apagado:

```
\ definir una secuencia de ENCENDIDO y APAGADO para un LED
: GPIO.on.off.sequence { position mask delay -- }
    1 position mask GPIO_OUT_W1TS_REG regSet
    delay ms
    1 position mask GPIO_OUT_W1TC_REG regSet
  ;
```

Vamos a probar nuestra palabra **GPIO.on.off.sequence** :

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

Esta secuencia debería encender el LED rojo durante un segundo. Ahora definamos un ciclo completo simulando un incendio en un cruce de carreteras:

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

`traffic-light` en funcionamiento simulará una luz de carretera clásica. Sin embargo, no podemos simular un semáforo en el que las luces roja y naranja están encendidas al mismo tiempo. Por tanto, primero escribiremos la palabra `TRAFFIC.sequence` que utiliza el código `GPIO.on.off.sequence`, con la diferencia de que también tendremos que utilizar un valor además de los demás parámetros:

```
\ definir una secuencia de ENCENDIDO y APAGADO
: TRAFFIC.sequence { val position mask delay -- }
    val position mask GPIO_OUT_W1TS_REG regSet
    delay ms
    val position mask GPIO_OUT_W1TC_REG regSet
  ;
```

A continuación definimos estas cuatro palabras que permitirán gestionar un ciclo complejo de incendios en las carreteras:

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

Ahora definimos un ciclo de luz de carretera tal como lo podríamos encontrar en Alemania:

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

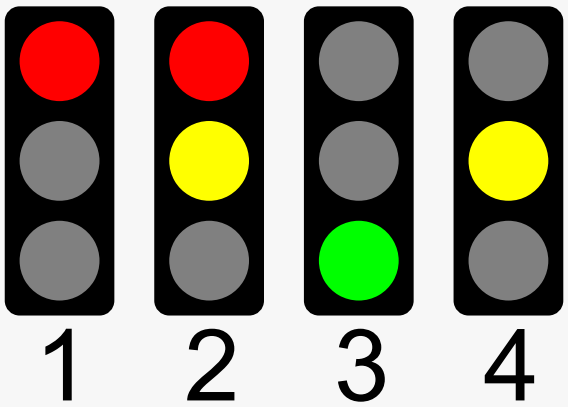


Figure 18: cycle de feux à quatre états

En Alemania los semáforos tienen cuatro ciclos. La particularidad de estas luces es encender simultáneamente las luces roja y amarilla antes de volver a pasar a la luz verde.

Y finalmente, probamos nuestro ciclo de luces de carretera en un bucle:

```
: TRAFFIC.loop ( -- )
```

```
    begin
        TRAFFIC.german.cycle
    key? until
;
```

Ejecutar **`TRAFFIC.loop`** simulará nuestra luz de carretera. Cuando llega la secuencia **`TRAFFIC.red-yellow`**, podemos ver que las luces amarilla y roja se encienden simultáneamente.

Vimos en este capítulo cómo actuar sobre varias salidas GPIO al mismo tiempo actuando directamente sobre los registros GPIO.

Sin embargo, con las tarjetas ESP32 no se recomienda gestionar demasiados LED simultáneamente. La tarjeta ESP32 debe reservarse para gestionar señales a accesorios muy eficientes energéticamente. Evitaremos por tanto conjuntos de 8 a 16 LED para crear un efecto tipo persecución por ejemplo, salvo que utilicemos elementos de control que utilicen una fuente de alimentación independiente.

Interrupciones de hardware con ESP32forth

Interrupciones

Cuando queremos gestionar eventos externos, un pulsador por ejemplo, tenemos dos soluciones:

- Pruebe el estado del botón con la mayor regularidad posible, a través de un bucle. Actuaremos según el estado de este botón.
- utilizar una interrupción. Asignamos el código de ejecución a una interrupción adjunta a un pin. El botón está conectado a este pin y el cambio de estado ejecutará esta palabra.

La solución de interrupción es la más elegante. Le permite aliviar el programa principal evitando monitorear el botón en bucle.

En su documentación ESP32forth, Brad NELSON da un ejemplo simple de manejo de interrupciones:

```
17 input pinMode
: test ." pinvalue: " 17 digitalRead . cr ;
' test 17 pinchange
```

Excepto que este ejemplo, tal como está escrito, tiene muchas posibilidades de no funcionar. Veremos por qué y proporcionaremos los elementos para que funcione.

Montaje de un pulsador

El botón está conectado como entrada a la fuente de alimentación de 3,3 V de la placa ESP32.

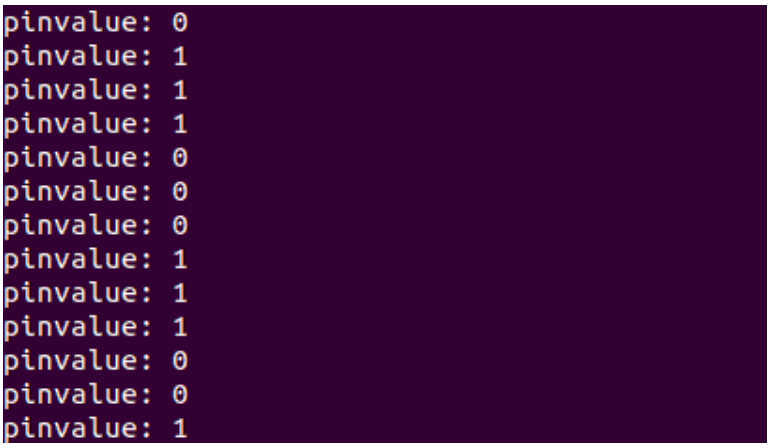
La salida del botón pulsador está conectada al pin GPIO17. Es todo.

Para que el ejemplo de Brad NELSON sea funcional, debe seleccionar el vocabulario `de interrupts` antes de configurar la interrupción usando `pinchange` . En el camino, definiremos la constante `button` :

```
17 constant button
button input pinMode
: test ." pinvalue: "
    button digitalRead . cr
  ;
interrupts
```

```
' test button pinchange
forth
```

Funciona, pero hay un efecto inesperado que hace que la interrupción se active inesperadamente:



La solución hardware consistiría en poner una resistencia de alto valor en la salida del botón y conectarla a GND.

Consolidación de software de la interrupción.

En la tarjeta ESP32, puedes activar una resistencia en cualquier pin GPIO. Esta activación se realiza mediante la palabra `gpio_pulldown_en` . Esta palabra acepta como parámetro el número del pin GPIO cuya resistencia se debe activar. A cambio, esta palabra devuelve 0 si la acción fue exitosa, un código de error en caso contrario:

```
17 constant button
button input pinMode
: test ." pinvalue: "
    button digitalRead . cr
  ;
interrupts
button gpio_pulldown_en drop
' test button pinchange
forth
```

El resultado de ejecutar la interrupción es significativamente mejor:

```
ok     button digitalRead . cr
ok   ;
ok interrupts
ok button gpio_pulldown_en drop
ok ' test button pinchange
ok forth
--> pinvalue: 1
pinvalue: 0
pinvalue: 1
pinvalue: 0
pinvalue: 1
pinvalue: 0
```

En cada cambio de estado, tenemos una interrupción. En la captura de pantalla anterior, cada cambio de estado muestra `pinvalue: 1` y luego `pinvalue: 0` .

Es posible tener en cuenta una interrupción únicamente en el flanco ascendente. Esto es posible indicando:

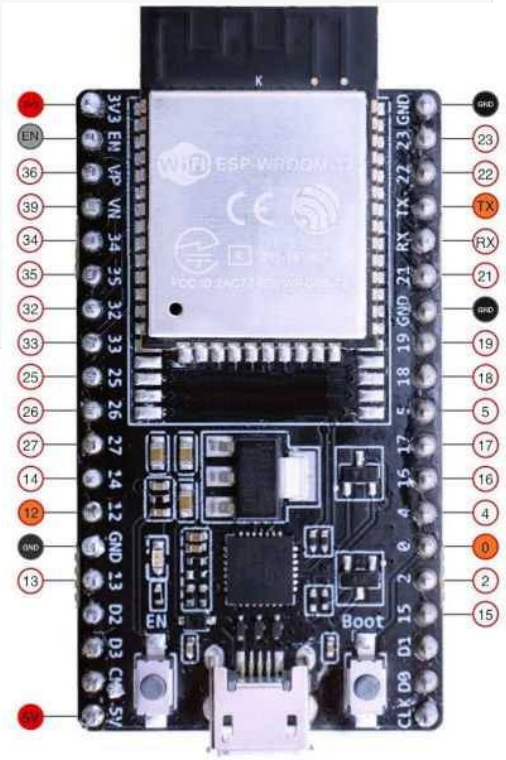
```
button GPIO_INTR_POSEDGE gpio_set_intr_type drop
```

La palabra `gpio_set_intr_type` acepta estos parámetros:

- `GPIO_INTR_ANYEDGE` para gestionar interrupciones de flanco ascendente o descendente
- `GPIO_INTR_NEGEDGE` para manejar interrupciones solo en el flanco descendente
- `GPIO_INTR_POSEDGE` para gestionar interrupciones solo en el flanco ascendente
- `GPIO_INTR_DISABLE` para desactivar las interrupciones

Complete el código FORTH con detección de flanco ascendente:

```
17 constant button
0 constant GPIO_PULLUP_ONLY
button input pinMode
: test ." pinvalue: "
    button digitalRead . cr
  ;
interrupts
button gpio_pulldown_en drop
button GPIO_INTR_POSEDGE
gpio_set_intr_type drop
' test button pinchange
forth
```



Informaciones complementarias

Para ESP32, todos los pines GPIO se pueden usar como interrupción, excepto GPIO6 a GPIO11.

No utilice alfileres de color naranja o rojo. Su programa podría comportarse inesperadamente al utilizarlos.

Usando el codificador rotatorio KY-040

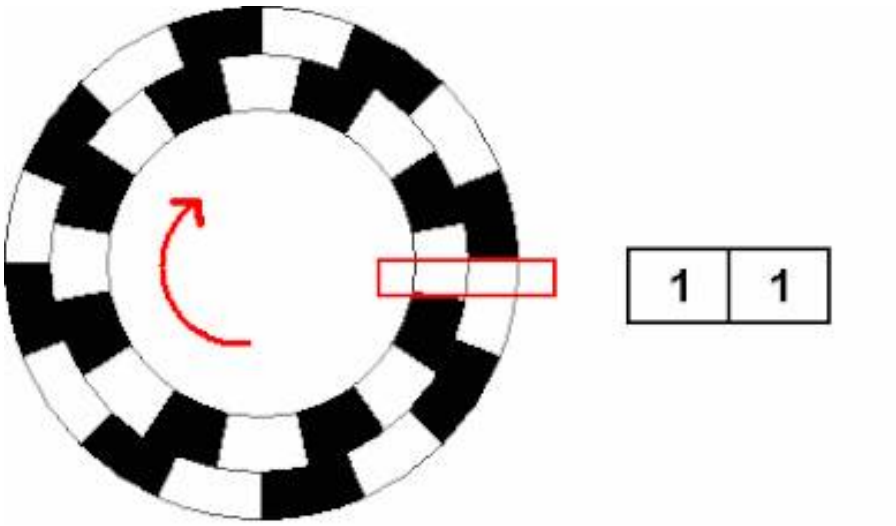
Descripción general del codificador

Para variar una señal, tenemos varias soluciones:

- una resistencia variable en un potenciómetro
- dos botones que gestionan la variación por software
- un codificador rotatorio

El codificador rotatorio es una solución interesante. Se puede hacer que actúe como un potenciómetro, con la ventaja de no tener inicio y final de carrera.

Su principio es muy simple. Aquí están las señales emitidas por nuestro codificador rotatorio:



Aquí está nuestro codificador:

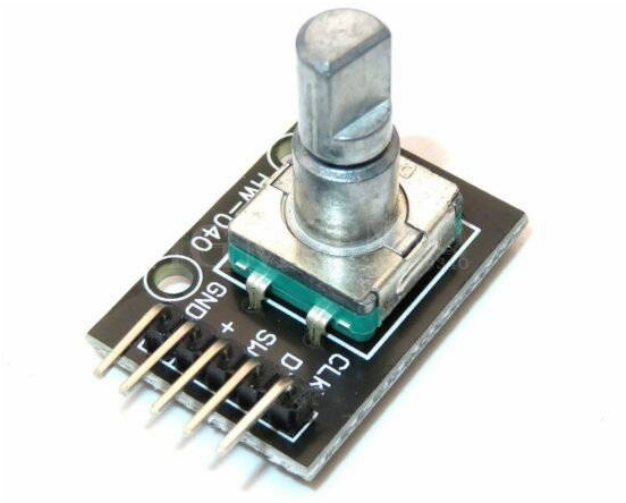


Diagrama de funcionamiento interno:

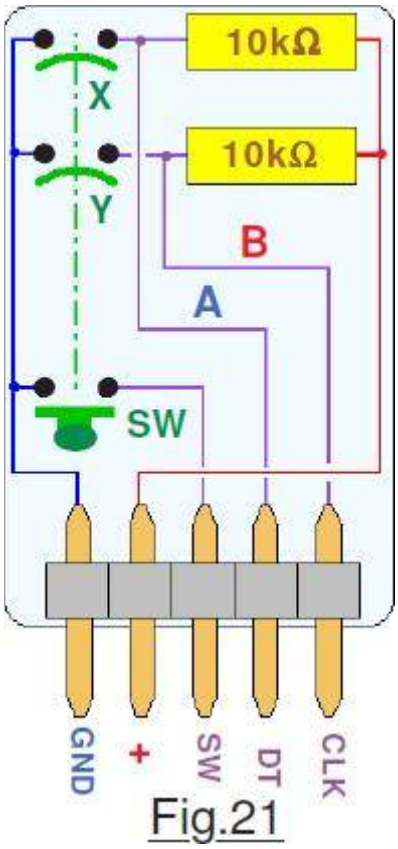


Fig.21

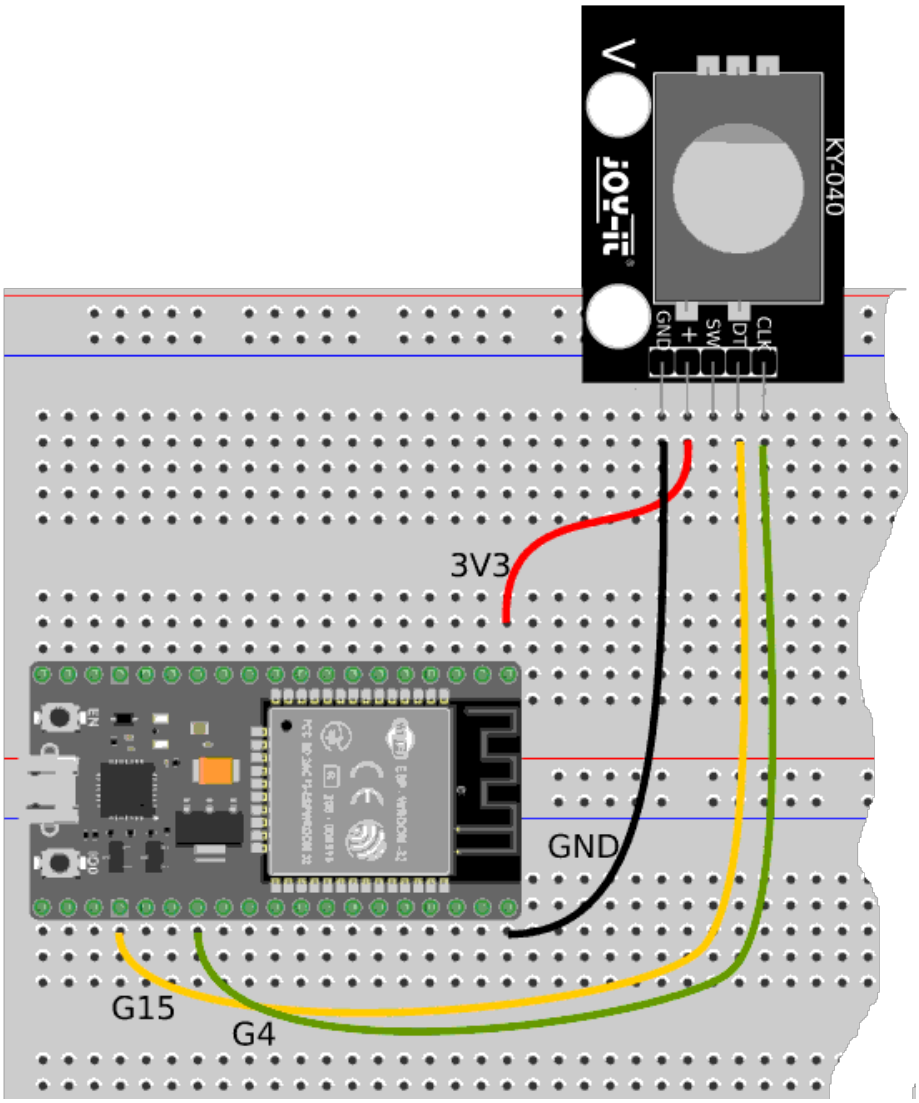
Según este diagrama nos interesan dos terminales:

- A (DT) -> cambiar X
- B (CLK) -> cambiar Y

Este codificador se puede alimentar con 5V o 3,3V. Esto nos conviene porque la tarjeta ESP32 tiene una salida de 3,3V.

Montaje del codificador en la placa de pruebas

Cablear nuestro codificador a la placa ESP32 solo requiere 4 cables:

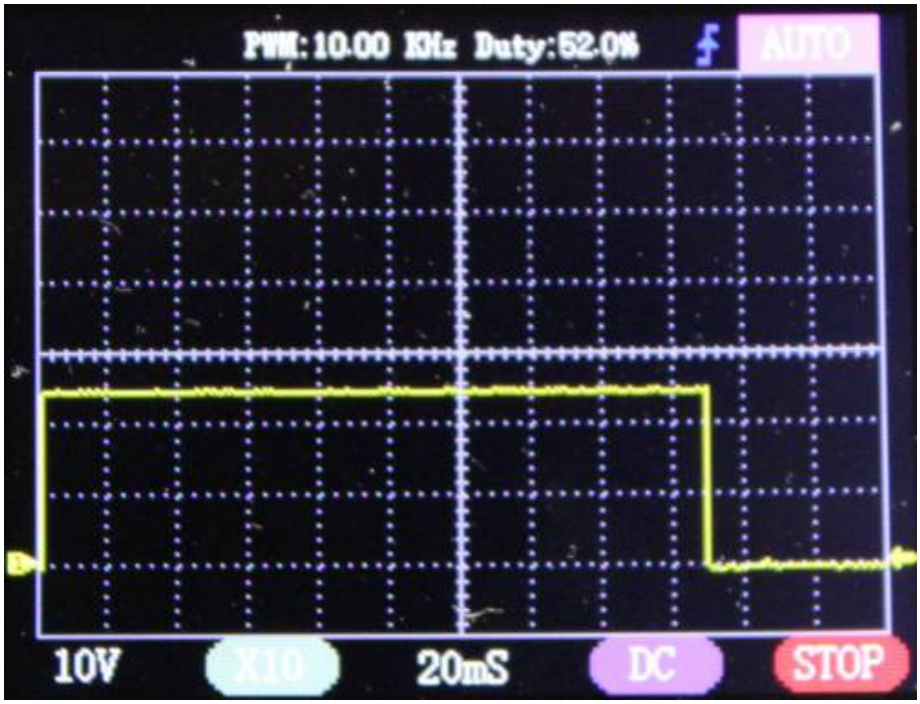


TENGA EN CUENTA : la posición de los pines G4 y G15 puede variar según la versión de su tarjeta ESP32.

Análisis de señales de codificador.

Mientras nuestro codificador está conectado, cada terminal A o B recibe un voltaje, aquí 3,3 V, cuya intensidad está limitada por una resistencia de 10 Kohms.

El análisis de la señal en el terminal G15 muestra claramente la presencia de la tensión de



3,3V:

En esta captura de señal, el nivel bajo en el terminal G15 aparece cuando se opera la varilla de control del codificador. Cuando está inactivo, la señal en el terminal G15 está en nivel alto.

Esto cambia todo, porque a nivel de programación debemos procesar la interrupción G15 como un flanco descendente.

Programación de codificadores

El codificador será gestionado por interrupción. Las interrupciones activan el programa sólo si una señal particular alcanza un nivel bien definido.

Gestionaremos una única interrupción en el terminal GPIO G15:

```
interrupts

\ enable interrupt on GPIO G15
: intG15enable ( -- )
    15 GPIO_INTR_POSEDGE gpio_set_intr_type drop
  ;
```

```
\ disable interrupt on GPIO G15
: intG15disable ( -- )
    15 GPIO_INTR_DISABLE gpio_set_intr_type drop
  ;

: pinsInit ( -- )
    04 input pinmode            \ GO4 as an input
    04 gpio_pulldown_en drop    \ Enable pull-down on GPIO 04
    15 input pinmode            \ G15 as an input
    15 gpio_pulldown_en drop    \ Enable pull-down on GPIO 15
    intG15enable
  ;
```

En la palabra **`pinsInit`** , inicializamos los pines GPIO G4 y G15 como entrada. Luego determinamos el modo de interrupción de G15 en el flanco descendente con **`15 GPIO_INTR_POSEDGE gpio_set_intr_type drop`**.

Probando la codificación

Esta parte del código no se debe utilizar en un ensamblaje final. Sólo se utiliza para comprobar que el codificador está correctamente conectado y funciona correctamente:

```
: test ( -- )
    cr ." PIN: "
    cr ."  - G15: " 15 digitalRead .
    cr ."  - G04: " 04 digitalRead .
  ;

pinsInit    \ initialise G4 and G15
' test 15 pinchange
```

Es la secuencia **`de ' test 15 pinchange`** que le dice a ESP32Forth que ejecute el código de prueba si se activa una interrupción por la acción del terminal G15.

Resultado de la acción en nuestro codificador. Solo guardamos los resultados de las acciones que llegaron a la parada, una vez en el sentido contrario a las agujas del reloj y luego en el sentido de las agujas del reloj:

```
PIN:
 - G15: 1  \ reverse clockwise turn
 - G04: 1
PIN:
 - G15: 0  \ clockwise turn
 - G04: 1
```

Incrementar y disminuir una variable con el codificador

Ahora que hemos probado el codificador mediante interrupción de hardware, podremos gestionar el contenido de una variable. Para ello definimos nuestra variable `KYvar` y las palabras que nos permitan modificar su contenido:

```
0 value KYvar   \ content is incremented or decremented

\ increment content of KYvar
: incKYvar ( n -- )
    1 +to KYvar
  ;

\ decrement content of KYvar
: decKYvar ( n -- )
    -1 +to KYvar
  ;
```

La palabra `incKYvar` incrementa el contenido de `Kyvar`. La palabra `decKYvar` disminuye el contenido de `KYvar`.

Probamos la modificación del contenido de la variable `KYvar` mediante esta palabra `testIncDec` definida de la siguiente manera:

```
\ used by interruption when G15 activated
: testIncDec ( -- )
    intG15disable
    15 digitalRead  if
        04 digitalRead  if
            decKYvar
        else
            incKYvar
        then
        cr ." KYvar: " KYvar .
    then
    1000 0 do loop  \ small wait loop
    intG15enable
  ;

pinsInit
' testIncDec 15 pinchange
```

Al girar el control del codificador hacia la derecha (en el sentido de las agujas del reloj), se incrementará el contenido de la variable KYvar. Una rotación hacia la izquierda disminuye el contenido de la variable KYvar:

```
pinsInit
' testIncDec 15 pinchange
-->
KYvar: 1    \ rotate Clockwise
KYvar: 2
KYvar: 3
KYvar: 4
KYvar: 3    \ rotate Contra Clockwise
KYvar: 2
KYvar: 1
KYvar: 0
KYvar: -1
KYvar: -2
```

Listado completo: usando el codificador rotatorio KY-040 @TODO: coloque el listado completo en la sección de listado

Parpadeo de un LED por temporizador

Comenzando con la programación FORTH

Cualquier principiante en programación conoce muy bien este ejemplo más que clásico: el parpadeo de un LED. Aquí está el código fuente, en lenguaje C para ESP32:

```
/*
 * This ESP32 code is created by esp32io.com
 * This ESP32 code is released in the public domain
 * For more detail (instruction and wiring diagram),
 *  visit https://esp32io.com/tutorials/esp32-led-blink
 */

// the code in setup function runs only one time when ESP32 starts
void setup() {
  // initialize digital pin GIOP18 as an output.
  pinMode(18, OUTPUT);
}

// the code in loop function is executed repeatedly infinitely
void loop() {
  digitalWrite(18, HIGH); // turn the LED on
  delay(500);             // wait for 500 milliseconds
  digitalWrite(18, LOW);  // turn the LED off
  delay(500);             // wait for 500 milliseconds
}
```

En FORTH idioma, no es muy diferente:

```
18 constant myLED

: led.blink ( -- )
    myLED output pinMode
    begin
        HIGH myLED pin
        500 ms
        LOW myLED  pin
        500 ms
    key? until
  ;
```

Si compila este código FORTH con ESP32forth instalado en su placa ESP32 y escribe `led.blink` desde el terminal, el LED conectado al puerto GPIO18 parpadeará.

Para inyectar código escrito en lenguaje C, será necesario compilarlo en la PC y luego cargarlo en la tarjeta ESP32, operaciones que llevan algún tiempo. Mientras que con el lenguaje FORTH, el compilador ya está operativo en nuestra placa ESP32. El compilador compilará el programa escrito en lenguaje FORTH en dos o tres segundos y permitirá su ejecución inmediata simplemente escribiendo la palabra que contiene este código, aquí `led.blink` para nuestro ejemplo.

En el lenguaje FORTH, podemos compilar cientos de palabras y probarlas inmediatamente, todo individualmente, lo cual no es posible en absoluto en el lenguaje C.

Factorizamos nuestro código FORTH de esta manera:

```
18 constant myLED

: led.on ( -- )
    HIGH myLED pin
  ;

: led.off ( -- )
    LOW myLED pin
  ;

: waiting ( -- )
    500 ms
  ;

: led.blink ( -- )
    myLED output pinMode
    begin
        led.on      waiting
        led.off     waiting
    key? until
  ;
```

Desde el terminal, podemos simplemente encender el LED escribiendo `led.on` y apagarlo escribiendo `led.off` . La ejecución de `led.blink` sigue siendo posible.

El objetivo de la factorización es dividir una función compleja y difícil de leer en un conjunto de funciones más simples y legibles. Con FORTH, se recomienda la factorización, por un lado para permitir una depuración más sencilla y, por otro lado, para permitir la reutilización de palabras factorizadas.

Estas explicaciones pueden parecer triviales para quienes conocen y dominan el idioma FORTH. Esto está lejos de ser obvio para las personas que programan en C, quienes se ven obligadas a agrupar llamadas a funciones en la función `loop()` general.

Ahora que esto está explicado, ¡nos olvidaremos de todo! Porque...

Intermitente por TIMER

Olvidaremos todo lo explicado anteriormente. Porque este ejemplo de LED parpadeante tiene una gran desventaja. Nuestro programa hace precisamente eso y nada más. En resumen, es un verdadero desperdicio de hardware y software hacer parpadear un LED en nuestra tarjeta ESP32. Veremos una forma muy diferente de producir este flasheo, en CUARTO idioma exclusivamente.

ESP32forth tiene dos palabras que serán muy útiles para gestionar este parpadeo del LED: `interval` y `rerun`.

Pero antes de discutir cómo funcionan estas dos palabras, echemos un vistazo a la noción de interrupción...

Interrupciones de hardware y software

Si planea administrar microcontroladores sin preocuparse por las interrupciones de hardware o software, ¡abandone el desarrollo informático por placas ESP32!

Tienes derecho a empezar y no sufrir interrupciones. Y le explicaremos las interrupciones y cómo utilizar las interrupciones del temporizador.

A continuación se muestra un ejemplo no informático de lo que es una interrupción:

- estás esperando un paquete importante;
- bajas a la puerta de tu casa cada minuto para ver si ha llegado el cartero.

En este escenario, en realidad pasas tu tiempo bajando, mirando y retrocediendo. De hecho, casi no tienes tiempo para hacer nada más...

En realidad, esto es lo que debería suceder:

- te quedas en tu casa;
- llega el cartero y toca el timbre;
- bajas y recoges tu paquete...

Un microcontrolador, que incluye la tarjeta ESP32, tiene dos tipos de interrupciones:

- **interrupciones de hardware** : se activan mediante una acción física en una de las entradas GPIO de la tarjeta ESP32;

- **Interrupciones de software** : se activan si ciertos registros alcanzan valores predefinidos.

Este es el caso de las interrupciones del temporizador, que definiremos como interrupciones de software.

Utilice las palabras intervalo y vuelva a ejecutar

La palabra `interval` se define en el vocabulario de los `timers` . Acepta tres parámetros:

- **xt** que ejecuta el código de la palabra que se lanzará cuando se active la interrupción;
- **usec** es el tiempo de espera, en microsegundos, antes de activar la interrupción;
- **t** es el número del temporizador que se activará. Este parámetro debe estar en el rango [0..3]

Tomemos parcialmente el código factorizado de nuestro LED parpadeante:

```
18 constant myLED

0 value LED_STATE

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

: led.off ( -- )
    LOW  dup myLED pin
    to LED_STATE
  ;

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

' led.toggle 500000 0 interval

: led.blink
```

```
    myLED output pinMode
    led.toggle
  ;
```

La palabra **rerun** está precedida por el número de temporizador activado antes de la definición de **interval** . La palabra **rerun** debe usarse en la definición de la palabra ejecutada por el temporizador.

La palabra **led.blink** inicializa la salida GPIO utilizada por el LED y luego ejecuta **led.toggle**.

En esta secuencia FORTH **' led.toggle 500000 0 interval**, inicializamos el temporizador 0 recuperando el código de ejecución de la palabra usando **rerun** , seguido del intervalo de tiempo, aquí 500 milisegundos, luego el número del temporizador que se activará.

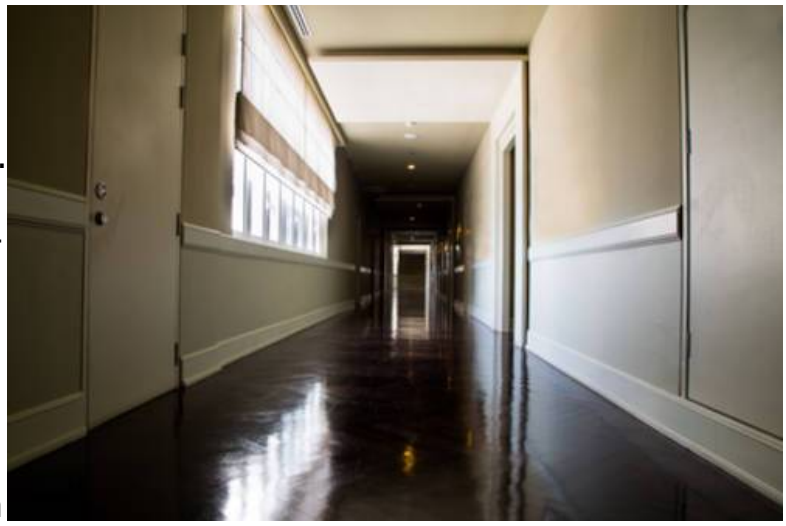
El parpadeo del LED comienza inmediatamente después de la ejecución de la palabra **led.blink**.

El intérprete FORTH de ESP32forth permanece accesible mientras el LED parpadea, ¡algo imposible en lenguaje C!

Temporizador de ama de llaves

Preámbulo

Estamos en 1990. Es un programador informático que trabaja mucho. Por eso a veces sale un poco tarde de su oficina.



Y fue durante una de sus últimas salidas de la oficina que entró al pasillo, uno de esos pasillos con un botón cronómetro en cada extremo. La luz ya está encendida. Pero por reflejo, nuestro amigo programador presiona el interruptor y se pincha el dedo. Se inserta una punta de madera en el interruptor para bloquear el temporizador.

Es la señora de la limpieza que está limpiando el suelo quien le explica: "Sí. El cronómetro sólo dura un minuto. Y a menudo me encuentro a oscuras. Como estoy cansado de presionar de nuevo sin parar el interruptor del cronómetro, bloqueo el botón con esta puntita de madera"…

Una solución

Esta anécdota despertó una idea en la cabeza de nuestro programador. Como tenía algunos conocimientos sobre microcontroladores, se propuso encontrar una solución para la señora de la limpieza.

La historia no dice en qué idioma programó su solución. Ciertamente en ensamblador.

Derivó el control de las luces a su circuito:

- una pulsación normal pone en marcha el cronómetro durante un minuto;
- si la luz está encendida, cualquier pulsación breve de un botón reduce el retardo de encendido a un minuto;
- El secreto de nuestro programador es haber previsto una pulsación larga de 3 segundos o más. Esta pulsación larga inicia el temporizador de 10 minutos de iluminación;

- si el temporizador está en circuito largo, otra pulsación larga reduce el retraso del temporizador a un minuto;
- un pitido corto reconoce la activación o desactivación de un ciclo de temporizador largo.

La señora de la limpieza realmente apreció esta mejora en el cronómetro. Ya no necesitaba bloquear el botón de ninguna manera.

¿Qué pasa con los demás trabajadores? Como nadie estaba al tanto de esta característica, continuaron usando el temporizador presionando brevemente el interruptor de activación.

Un temporizador FORTH para ESP32Forth

Ya comprende, vamos a utilizar `timers` para administrar un temporizador integrando el escenario descrito anteriormente.

```
\ myLIGHTS connecté à GPIO18
18 constant myLIGHTS

\ définit temps max pour cycle normal ou étendu, en secondes
 60 constant MAX_LIGHT_TIME_NORMAL_CYCLE
600 constant MAX_LIGHT_TIME_EXXTENDED_CYCLE

\ temps max pour cycle normal ou étendu, en secondes
0 value MAX_LIGHT_TIME

timers
\ coupe éclairage si MAX_LIGHT_TIME égal 0
: cycle.stop ( -- )
    -1 +to MAX_LIGHT_TIME       \ décrémente temps max de 1 seconde
    MAX_LIGHT_TIME 0 = if
        LOW myLIGHTS pin        \ coupe éclairage
    else
        0 rerun
    then
  ;

\ initialise timer 0
' cycle.stop 1000000 0 interval

\ démarre un cycle d’éclairage, n est délai en secondes
: cycle.start ( n -- )
    1+ to MAX_LIGHT_TIME        \ sélect. Temps max
    myLIGHTS output pinMode
```

```
    HIGH myLIGHTS pin           \ active éclairage
    0 rerun
  ;
```

Ya podemos probar nuestro temporizador:

```
 3 cycle.start  \ enciende las luces durante 3 segundos
10 cycle.start  \ enciende las luces durante 10 segundos
```

Si reiniciamos `cycle.start` mientras la luz está encendida, comenzamos nuevamente para un nuevo ciclo de iluminación de n segundos.

Por lo tanto todavía tenemos que gestionar la activación de estos ciclos desde un interruptor.

Gestión del botón de encendido de luz.

No es brujería. Gestionaremos un pulsador. Como tenemos a mano una tarjeta ESP32, programable con ESP32Forth, aprovecharemos para gestionar este botón por interrupciones. Las interrupciones que gestionan los terminales GPIO en la placa ESP32 son interrupciones de hardware.

Nuestro botón está montado en el terminal GPIO17 (G17).

Definimos dos palabras, `intPosEdge` e `intNegEdge` , que determinan el tipo de activación de la interrupción:

- `intPosEdge` para activar la interrupción en el flanco ascendente;
- `intNegEdge` para activar la interrupción en el flanco descendente.

```
17 constant button  \ mount button on GPIO17

interrupts                          \ select interrupts vocabulary

\ interrupt activated for upraising signal
: intPosEdge ( -- )
    button #GPIO_INTR_POSEDGE gpio_set_intr_type drop
  ;

\ interrupt activated for falldown signal
: intNegEdge ( -- )
    button #GPIO_INTR_NEGEDGE gpio_set_intr_type drop
  ;
```

Luego necesitamos definir algunas variables y constantes:

- dos constantes, **CYCLE_SHORT** y **CYCLE_LONG** que se utilizarán para definir la duración del encendido de las luces. Aquí elegimos 3 y 10 segundos para hacer nuestras pruebas.
- la variable **msTicksPositiveEdge** que almacena la posición del contador de espera entregado por ms-ticks
- la constante **DELAY_LIMIT** que determina el umbral para determinar una pulsación corta o larga del pulsador. Aquí son 3000 milisegundos o 3 segundos. Un usuario normal NUNCA presionará el botón de encendido de la luz durante 3 segundos. Sólo la señora de la limpieza conoce la maniobra para tener una iluminación larga y continua...

```
03 constant CYCLE_SHORT     \ durée éclairage pour appui bref, en
secondes
10 constant CYCLE_LONG      \ durée éclairage pour appui long

\ mémorise valeur de ms-ticks sur front montant
variable msTicksPositiveEdge

\ délai limite: si delai < DELAY_LIMIT, cycle court
3000 constant DELAY_LIMIT
```

La palabra **getButton** se lanza en cada interrupción activada al presionar el botón conectado a GPIO17 (G17) en nuestra placa ESP32.

La ejecución **de getButton** , las interrupciones en G17 están deshabilitadas. Esta interrupción se reactivará al finalizar la definición. Esta desactivación es necesaria para evitar el apilamiento de interrupciones.

A la desactivación le sigue el **70000 0 do loop**. Este bucle se utiliza para gestionar los rebotes de contactos. Aquí gestionamos el rebote por software.

```
\ palabra ejecutada por interrupción
: getButton ( -- )
    button gpio_intr_disable drop
    70000 0 do loop  \ anti rebond
    button digitalRead 1 =
    if
        ms-ticks msTicksPositiveEdge !
        intNegEdge
    else
        intPosEdge
        ms-ticks msTicksPositiveEdge @ -
        DELAY_LIMIT >
        if      CYCLE_LONG  cr ." BEEP"
        else    CYCLE_SHORT cr ." ----"
```

```
    then
    cycle.start
    button gpio_intr_enable drop
  ;
```

En el flanco ascendente, la palabra `getButton` registra el estado del contador de retraso y posiciona las interrupciones en el flanco descendente. Luego dejamos esta palabra reactivando las interrupciones.

En el flanco descendente, la palabra `getButton` calcula el tiempo transcurrido desde el flanco ascendente. Si este retraso es mayor que `DELAY_LIMIT` , se inicia un ciclo de encendido largo. De lo contrario, se inicia un breve ciclo de encendido.

La activación de un ciclo de encendido largo se indica mediante la visualización de "BEEP" en el terminal.

En el escenario original, esto se materializa mediante un breve pitido.

Finalmente, inicializamos el botón y la interrupción de hardware en este botón:

```
\ botón de inicialización y vectores de interrupción
button input pinMode            \ sélectionne G17 en mode entrée
button gpio_pulldown_en drop    \ active résistance interne de G17
' getButton button pinchange
intPosEdge

forth
```

Conclusión

Mira el vídeo de montaje: https://www.youtube.com/watch?v=OHWMh_bIWz0

Este caso de estudio muy sencillo muestra cómo gestionar simultáneamente el temporizador y una interrupción de hardware.

Estos dos mecanismos son muy poco preventivos. El cronómetro deja disponible el acceso al intérprete FORTH. La interrupción de hardware está operativa incluso si FORTH está ejecutando otro proceso.

No realizamos múltiples tareas. ¡Es importante decirlo!

Sólo espero que este caso de libro de texto te proporcione muchas ideas para tus desarrollos...

Reloj en tiempo real

La palabra MS-TICKS

La palabra `MS-TICKS` se utiliza en la definición de la palabra `ms`:

```
DEFINED? ms-ticks [IF]
  : ms ( n -- )
      ms-ticks >r
      begin
          pause ms-ticks r@ - over
      >= until
      rdrop drop
    ;
[THEN]
```

Esta palabra `MS-TICKS` está en el centro de nuestras investigaciones. Si ponemos en marcha la tarjeta ESP32 su ejecución restablece el número de milisegundos transcurridos desde que se puso en marcha la tarjeta ESP32. Este valor sigue creciendo. El valor de saturación de este conteo es 2^{32}-1, o 4294967295 milisegundos, o aproximadamente 49 días...

Cada vez que se reinicia la tarjeta ESP32, este valor se reinicia desde cero.

Administrar un reloj de software

A partir de los datos **de HH MM SS** (Horas, minutos, segundos), es fácil reconstruir un valor entero, en milisegundos, correspondiente al tiempo transcurrido desde las 00:00:00. Si a este tiempo le restamos el valor de `MS-TICKS` , tenemos un valor de tiempo de inicio para determinar el tiempo real . Por lo tanto, inicializamos un contador básico `currentTime` a partir de la palabra `RTC.set-time` :

```
0 value currentTime

\ almacenar la hora actual
: RTC.set-time { hh mm ss -- }
    hh 3600 *
    mm 60 *
    ss + +  1000 *
    MS-TICKS - to currentTime
  ;
```

Ejemplo de inicialización: `22 52 00 RTC.set-time` inicializa la base de tiempo para 22:52:00...

Para inicializar correctamente, prepare los tres valores **HH MM SS** seguidos de la palabra **`RTC.set-time`** , observe su reloj. Cuando llegue el momento esperado, ejecute la secuencia de inicialización.

La operación inversa recupera los valores **de HH MM** y **SS** de la hora actual, utilizando esta palabra:

```
\ obtener la hora actual en segundos
: RTC.get-time ( -- hh mm ss )
    currentTime MS-TICKS + 1000 /
    3600 /mod swap 60 /mod swap
  ;
```

Finalmente, definimos la palabra **`RTC.display-time`** que le permite mostrar la hora actual después de la inicialización de nuestro reloj de software:

```
\ utilizado para visualización de tiempo parcial SS y MM
: :## ( n -- n' )
    #  6 base ! #  decimal  [char] : hold
  ;

\ mostrar hora actual
: RTC.display-time ( -- )
    currentTime MS-TICKS + 1000 /
    <# :## :## 24 MOD #S #> type
  ;
```

El siguiente paso sería conectarnos a un servidor horario, con protocolo NTP, para inicializar automáticamente nuestro reloj software.

Medir el tiempo de ejecución de una CUARTA palabra

Medir el desempeño de las definiciones FORTH

Comencemos definiendo la palabra **measure:** la cual realizará estas mediciones de tiempo de ejecución:

```
: measure: ( exec: -- <word> )
    ms-ticks >r
    ' execute
    ms-ticks r> -
    cr ." execution time: "
    <# # # # [char] . hold #s #> type ." sec." cr
  ;
```

En esta palabra, recuperamos la hora mediante **ms-ticks** , luego recuperamos el código de ejecución de la palabra que sigue **measure:,** ejecutamos esta palabra, recuperamos el nuevo valor de tiempo mediante **ms-ticks** . Hacemos la diferencia, que corresponde al tiempo transcurrido, en milisegundos, que tarda la palabra en ejecutarse. Ejemplo :

```
measure: words
\ display: execution time: 0.210sec.
```

La palabra **words** se ejecutó en 0,2 segundos. Este tiempo no tiene en cuenta los retrasos en la transmisión por parte del terminal. Este tiempo tampoco tiene en cuenta el retraso que tarda la **measure:** recuperar el código de ejecución de la palabra a medir.

Si hay parámetros para pasar a la palabra a medir, estos deben apilarse antes de llamar **a medida:** seguido de la palabra a medir:

```
: CUADRADO (n -- n-exp2)
doble*
;
3 medida: CUADRADO
\ cartel:
\tiempo de ejecución: 0.000seg.
```

Este resultado significa que nuestra definición **de CUADRADO** se ejecuta en menos de un milisegundo.

Repetiremos esta operación un número determinado de veces:

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

```
3 measure: SQUARE
\ display:
\ execution time: 0.000sec.
```

Ejecutando la palabra **SQUARE** 1000 veces , precedida de un apilamiento de valores y desapilado del resultado, llegamos a un tiempo de ejecución de 1 milisegundo. ¡Podemos deducir razonablemente que **SQUARE** se ejecuta en menos de un microsegundo!

Probando algunos bucles

Vamos a probar algunos bucles, con 1 millón de iteraciones. Comencemos con un bucle **do loop**:

```
: test-loop ( -- )
    1000000 0 do
    loop
   ;
measure: test-loop
\ display:
\ execution time: 1.327sec.
```

Ahora veamos con un bucle **for-next** :

```
: test-for ( -- )
    1000000 for
    next
   ;
measure: test-for
\ display:
\ execution time: 0.096sec.
```

El bucle **for-next** se ejecuta casi 14 veces más rápido que el **bucle do** .

Veamos qué nos depara un bucle **begin-until :**

```
: test-begin ( -- )
    1000000 begin
        1- dup 0=
    until
   ;
measure: test-begin
\ display:
\ execution time: 0.273sec.
```

Esto es más eficiente que el **bucle do** , pero aún así tres veces más lento que el bucle **for-next** .

Ahora está equipado para crear programas FORTH aún más eficientes.

Programar un analizador de luz

Preámbulo

Como parte de un proyecto solar que utiliza varios paneles solares y su microinversor, surgen algunos problemas con la gestión de la energía eléctrica producida.

La principal preocupación es activar los grandes aparatos de consumo sólo si los paneles solares funcionan a pleno sol. Un dispositivo en particular se refiere, el cúmulo de agua caliente:

- activar el dispositivo cuando los paneles estén expuestos a la luz solar directa;
- desactivar el dispositivo cuando pasen nubes.

Los microinversores inyectan energía a la red eléctrica general. Si un dispositivo que consume mucha electricidad está activo cuando pasan las nubes, este dispositivo se alimentará principalmente de la red general.

En este artículo presentamos una solución que permite la detección de nubes mediante un panel solar en miniatura y una tarjeta ESP32.

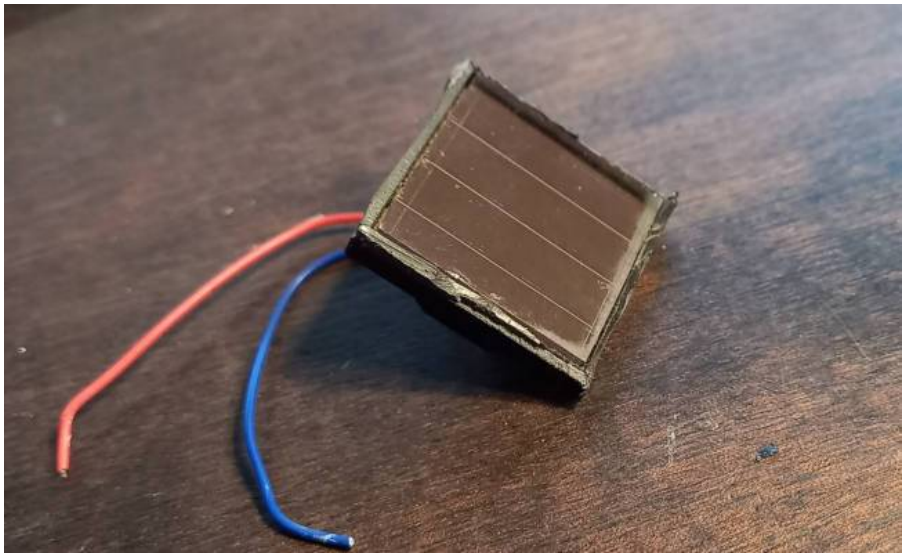
Código completo disponible aquí:
https://github.com/MPETREMANN11/ESP32forth/blob/main/ADC/solarLightAnalyzer.txt

El panel solar en miniatura

Para crear nuestro detector de nubes, usaremos un panel solar muy pequeño, aquí un panel de 25 mm x 25 mm.

Recuperación de un panel solar en miniatura

Este panel solar en miniatura se recupera de una lámpara de jardín averiada:



Aquí está nuestro mini panel solar extraído de esta lámpara de jardín:

Sacrificamos dos conectores Dupont para permitir realizar varias mediciones en nuestra placa prototipo. Estos conectores están soldados a los dos cables rojo y azul que salen del mini panel solar.

Medición del voltaje del panel solar.

Comenzamos midiendo el voltaje sin carga de nuestro mini panel solar, aquí con un osciloscopio. Esta medida de tensión también se puede realizar con un voltímetro:

¡Con mucha luz, la tensión medida es de 14,2 voltios!

Bajo luz difusa, el voltaje cae a 5,8 voltios.

Al cubrir el mini panel solar con la mano, el voltaje cae a casi 0 Voltios.

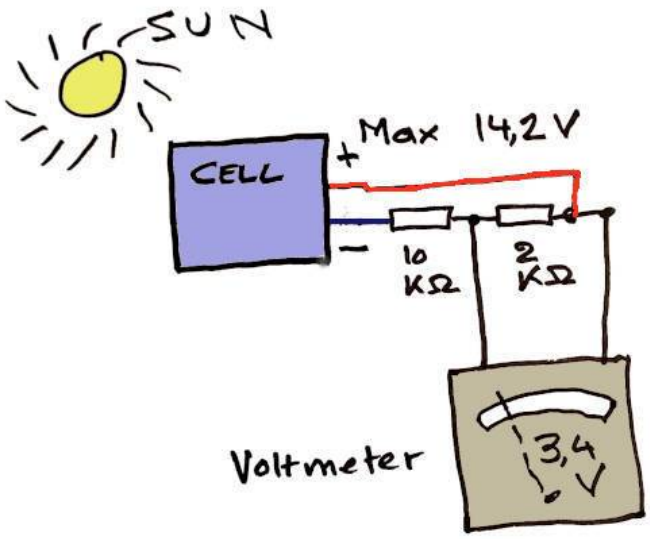
Medición de corriente del panel solar

La corriente, es decir, la intensidad, debe medirse con un amperímetro. La función de amperímetro de un controlador universal será adecuada. Al cortocircuitar el minipanel solar en condiciones de mucha luz se puede medir una corriente de 10 mA.

Por tanto, nuestro mini panel solar tiene una potencia aproximada de 0,2 vatios.

Antes de conectar nuestro mini panel solar a la tarjeta ESP32, es imprescindible bajar el voltaje de salida. No es posible inyectar este voltaje de 14,2 Voltios en una entrada de la tarjeta ESP32. Tal voltaje destruiría el circuito interno de la placa ESP32.

Bajar el voltaje del panel solar



La idea es reducir el voltaje en nuestro mini panel solar. Después de algunas pruebas, elegimos dos resistencias, una de 220 ohmios y la otra de 1K ohmios. Montaje de estas resistencias:

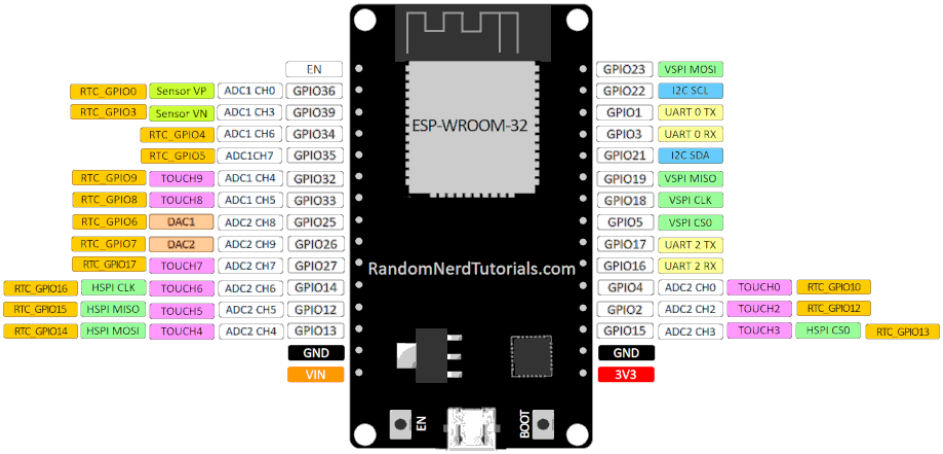
La medida de voltaje se toma entre las dos resistencias y el terminal positivo del panel solar.

El voltímetro ahora indica un voltaje máximo de 3,2 V en luz brillante, un voltaje de 0,35 V en luz difusa.

Programación del analizador solar

La placa ESP32 tiene 18 canales de 12 bits que permiten la conversión de analógico a digital (ADC). Para analizar el voltaje de nuestro mini panel solar, un solo canal ADC es necesario y suficiente.

Sólo están disponibles 15 canales ADC :



Usaremos uno, el canal **ADC1_CH6** que está conectado al pin **G34:**

```
34 constant SOLAR_CELL

: init-solar-cell ( -- )
    SOLAR_CELL input pinMode
  ;

init-solar-cell
```

Para leer el voltaje en el punto entre las dos resistencias, simplemente ejecute `SOLAR_CELL analogRead` . Esta secuencia cae un valor entre 0 y 4095. El valor más bajo corresponde a voltaje cero. El valor más alto corresponde a una tensión de 3,3 voltios.

Definición `de solar-cell-read` para recuperar este voltaje:

```
: solar-cell-read ( -- n )
    SOLAR_CELL analogRead
  ;
```

Probemos esta definición en un bucle:

```
: solar-cell-loop ( --)
    init-solar-cell
    begin
        solar-cell-read cr .
        200 ms
    key? until
  ;
```

Cuando se ejecuta `solar-cell-loop` , cada 200 milisegundos, se muestra el valor de conversión de voltaje del ADC:

```
...
322
331
290
172
39
0
0
0
0
19
79
86
...
```

Aquí los valores se obtuvieron iluminando el mini panel solar con una lámpara de alta potencia. Los valores cero corresponden a la ausencia de iluminación.

Las pruebas con el sol real muestran mediciones superiores a 300.

Gestionar la activación y desactivación de un dispositivo

Para empezar, definiremos dos pines, un pin reservado para gestionar una señal de activación y el otro para una señal de desactivación:

- Pin G17 conectado a un LED verde. Este pin se utiliza para activar un dispositivo.
- Pin G16 conectado a un LED rojo. Este pin se utiliza para desactivar un dispositivo.

```
17 constant DEVICE_ON      \ green LED
16 constant DEVICE_OFF     \   red LED

: init-device-state ( -- )
    DEVICE_ON  output pinMode
    DEVICE_OFF output pinMode
  ;
```

Podríamos haber usado un solo pin para administrar el dispositivo remoto. Pero algunos dispositivos, como los relés biestables, tienen dos bobinas:

- la primera bobina se alimenta para que los contactos cambien. El estado no cambia cuando la bobina ya no está energizada;
- para volver al estado inicial, se alimenta la segunda bobina.

Por este motivo nuestra programación tendrá en cuenta este tipo de dispositivos.

```
\ definir disparador de retardo de estado alto
500 value DEVICE_DELAY

\ establecer nivel ALTO de activación
: device-activation { trigger -- }
    trigger HIGH digitalWrite
    DEVICE_DELAY ?dup
    if
        ms
        trigger LOW  digitalWrite
    then
  ;
```

Aquí, la pseudoconstante **`DEVICE_DELAY`** se utiliza para indicar el retraso durante el cual la señal de control debe mantenerse alta. Después de este tiempo, la señal de control vuelve al estado bajo.

Si el valor de **`DEVICE_DELAY`** es cero, la señal de control permanece alta.

Es la palabra **`trigger-activation`** la que gestiona la activación del pin correspondiente:

- **TRIGGER_ON trigger-activation** establece de forma permanente o transitoria el pin conectado al LED verde en alto;
- **TRIGGER_OFF trigger-activation** establece el pin conectado al LED rojo de forma permanente o transitoria en alto.

Definimos ahora dos palabras, **device-ON** y **device-OFF**, responsables respectivamente de activar y desactivar el dispositivo que se pretende controlar mediante los pines G16 y G17:

```
\ definir el estado del dispositivo: 0=BAJO, -1=ALTO
0 value DEVICE_STATE

: enable-device ( -- )
    DEVICE_STATE invert
    if
        DEVICE_OFF LOW  digitalWrite
        DEVICE_ON  device-activation
        -1 to DEVICE_STATE
    then
  ;

: disable-device ( -- )
    DEVICE_STATE
    if
        DEVICE_ON  LOW  digitalWrite
        DEVICE_OFF device-activation
         0 to DEVICE_STATE
    then
  ;
```

El estado del dispositivo se almacena en **DEVICE_STATE** . Este estado se prueba antes de intentar cambiar de estado. Si el dispositivo está activo, no se reactivará repetidamente. Lo mismo si el dispositivo está inactivo.

```
\ definir valor de activación para cielo soleado o nublado
300 value SOLAR_TRIGGER

\ si luz solar > SOLAR_TRIGGER, activa la acción
: action-light-level ( -- )
    solar-cell-read SOLAR_TRIGGER >=
    if
        enable-device
    else
        disable-device
    then
  ;
```

Activado por interrupción del temporizador

La forma más elegante es utilizar una interrupción del temporizador. Usaremos el temporizador 0:

```
  0 to DEVICE_DELAY
200 to SOLAR_TRIGGER
init-solar-cell
init-device-state

timers
: action ( -- )
    action-light-level
    0 rerun
  ;

' action 1000000 0 interval
```

A partir de ahora, el temporizador analizará el flujo luminoso cada segundo y actuará en consecuencia. Enlace al vídeo: https://youtu.be/lAjeev2u9fc

Para este vídeo, actuamos sobre dos parámetros:

- `0 to DEVICE_DELAY` enciende los LED de forma permanente. El LED rojo indica que el dispositivo está desactivado. El LED verde indica la activación del dispositivo;
- `200 to SOLAR_TRIGGER` determina el umbral para activar el estado de luz solar. Este parámetro es ajustable para adaptarse a las características del mini panel solar.

La palabra `de acción` funciona mediante interrupción del temporizador. Por tanto, no es necesario disponer de un bucle general para que funcione el detector.

Dispositivos controlados por el sensor de luz solar.

En resumen, tenemos dos cables de control, un cable correspondiente al LED verde en el video y el otro cable correspondiente al LED rojo. El programa está diseñado para que ambos cables de control no puedan estar activos al mismo tiempo.

Para tener una señal continua en cualquiera de los cables de control, el valor `DEVICE_DELAY` solo necesita ser cero. A continuación se explica cómo inicializar este escenario:

```
\ comenzar con señal de comando constante
: start-CCS ( -- )
      0 to DEVICE_DELAY
```

```
    200 to SOLAR_TRIGGER
    init-solar-cell
    init-device-state
    disable-device
    [ timers ] ['] action 1000000 0 interval
  ;
```

Y para tener comandos temporizados asignaremos a **DEVICE_DELAY** el retraso del nivel del comando de activación o desactivación del dispositivo.

```
\ comenzar con señal de comando temporal
: start-TCS ( -- )
    300 to DEVICE_DELAY
    200 to SOLAR_TRIGGER
    init-solar-cell
    init-device-state
    disable-device
    [ timers ] ['] action 1000000 0 interval
  ;
```

Escenario **de start-TCS** es típico de un control de relé biestable operado por impulsos. El relé se activa si recibe un comando de activación. Incluso si la señal de activación cae, el relé biestable permanece activo. Para desactivar el relé biestable se le debe transmitir un comando de desactivación en la línea de desactivación.

En conclusión, nuestro analizador de luz solar puede controlar una amplia variedad de dispositivos. Basta con adaptar las interfaces de control de estos dispositivos a las

El archivo de script FORTH completo **solarLightAnalyzer.fs** está disponible en el archivo **ESP32forth-book.zip**

Gestión de salidas N/A (Digital/Analógica)

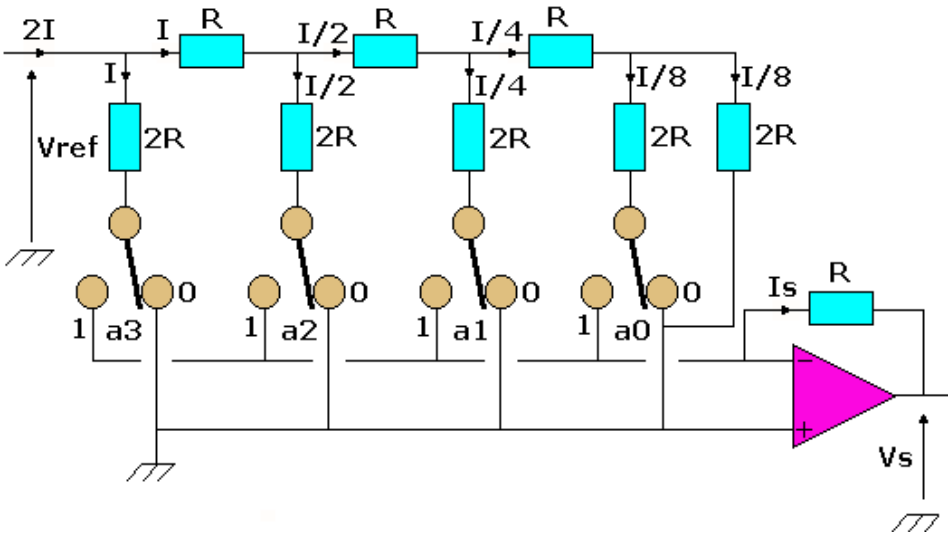
Conversión digital/analógica

La conversión de una cantidad digital en un voltaje eléctrico proporcional a esta cantidad digital es una funcionalidad muy interesante en un microcontrolador.

Cuando utiliza Internet y realiza una llamada telefónica VOIP, su voz se transforma en valores numéricos. La de su corresponsal se transformará inversamente de números a señales sonoras. Este proceso utiliza conversión de analógico a digital y viceversa.

Conversión D/A con circuito R2R

Aquí está el diagrama básico de un convertidor digital a analógico de 4 bits:



El valor a convertir, de 4 bits, se distribuye en 4 pines a0 a a3. El voltaje de referencia se inyecta en la parte superior izquierda del circuito. Este voltaje genera una intensidad 2I si esta corriente no pasa por ninguna resistencia.

Dependiendo de los bits activados, para cada bit el voltaje se divide y se suma al de los demás bits activos. Por ejemplo, si a2 y a0 están activos, la corriente de salida Is será la suma de I/2 e I/8.

Para este circuito de 4 bits, el paso de conversión es I/16. Con ESP32, la conversión se realiza en 8 bits. Por tanto, el paso de conversión será I/256.

Conversión D/A con ESP32

Ninguna placa ARDUINO tiene una salida de conversión D/A. Para realizar una conversión D/A con una placa ARDUINO, debe utilizar un componente externo.

Con la tarjeta ESP32 tenemos dos pines, G25 y G26, correspondientes a las salidas de conversión D/A.

Para nuestro primer experimento de conversión D/A con la placa ESP32, conectaremos dos LED a los pines G25 y G26:

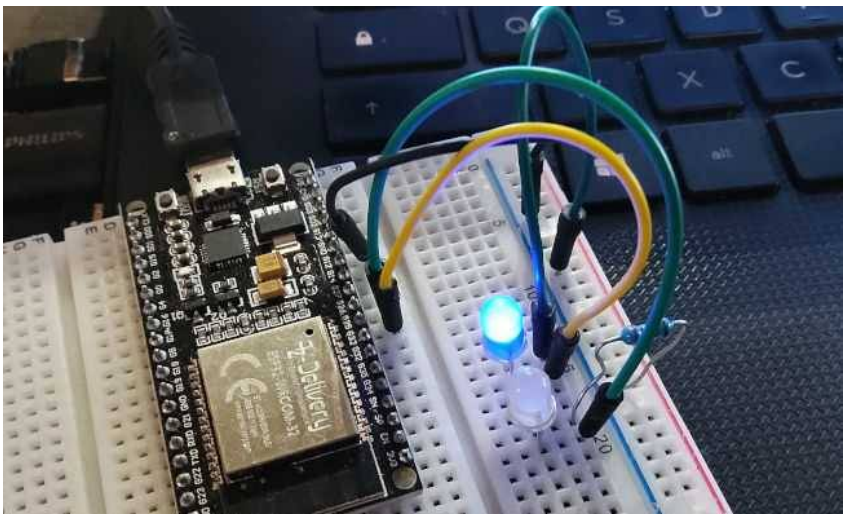
```
\ define Gx a LED
25 constant ledBLUE     \  blue led on G25
26 constant ledWHITE    \ white led on G26
```

Antes de realizar una conversión D/A, planeamos inicializar los pines G25 y G26:

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

Y definimos dos palabras que nos permiten controlar la intensidad de nuestros dos LED:

```
\ establecer intensidad para led AZUL
: BLset ( val -- )
    ledBLUE   swap dacWrite
    ;

\ establecer intensidad para led BLANCO
: WHset ( val -- )
    ledWHITE  swap dacWrite
    ;
```



Las palabras `BLset` y `WHset` aceptan como parámetro un valor numérico en el rango 0..255.

En la foto, después de `initLeds` , la secuencia `200 BLset` enciende el LED azul a potencia reducida.

Para encenderlo a máxima potencia utilizaremos la secuencia `255 BLset`

Para apagarlo por completo enviaremos esta secuencia `0 BLset`

Posibilidades de conversión D/A

Aquí, con nuestros dos LED, hemos creado un montaje sencillo y poco interesante.

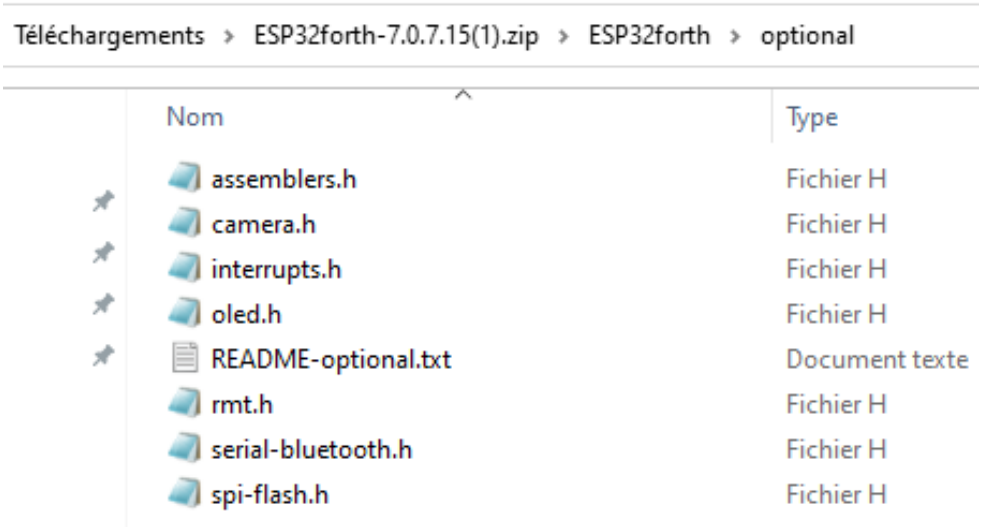
Este montaje tiene el mérito de mostrar que la conversión D/A funciona perfectamente. La conversión D/A permite:

- control de potencia a través de un circuito dedicado, por ejemplo un variador para un motor eléctrico;
- generación de señales: sinusoide, cuadrado, triángulo, etc...
- conversión de archivos de sonido
- síntesis de sonido...

Código completo disponible aquí:
https://github.com/MPETREMANN11/ESP32forth/blob/main/DAC/DAoutput.txt

Instalación de la biblioteca OLED para SSD1306

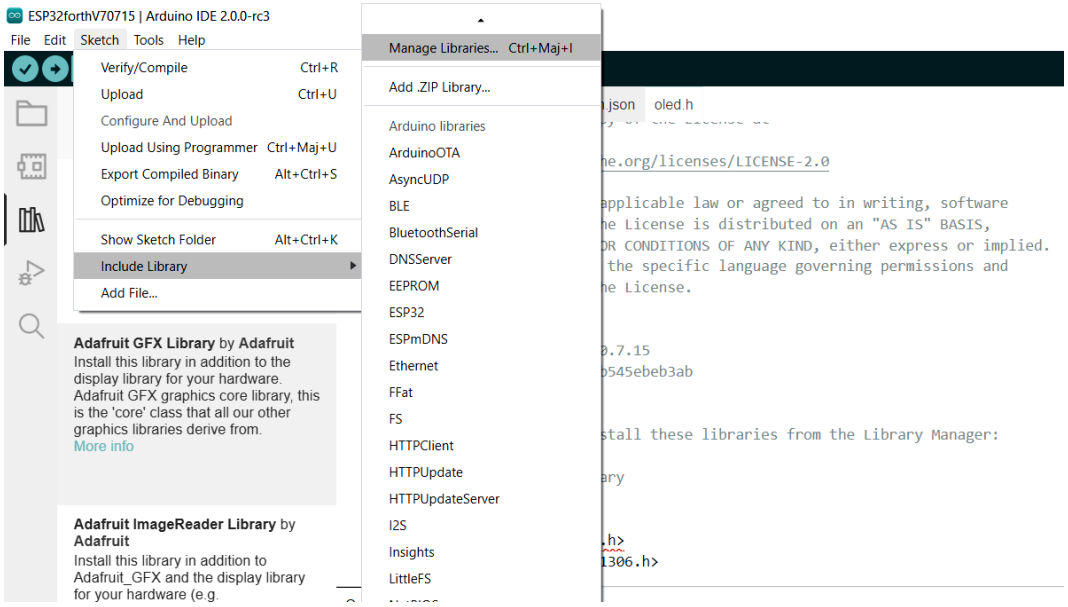
Desde ESP32 en adelante versión 7.0.7.15, las opciones están disponibles en la carpeta **optional**:



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
```

La interfaz I2C en ESP32

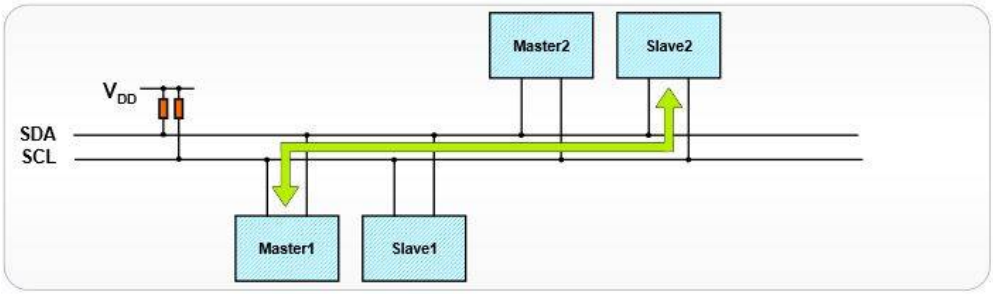
Introducción

I2C (significa: Inter-Integrated Circuit, en inglés) es un bus informático que surgió de la "guerra de estándares" lanzada por los actores del mundo electrónico. Diseñado por Philips para aplicaciones de domótica y electrónica doméstica, permite conectar fácilmente un microprocesador y varios circuitos, en particular los de un televisor moderno: receptor de mando a distancia, ajustes del amplificador de baja frecuencia, sintonizador, reloj, gestión del euroconector, etc. .

Existen innumerables periféricos que utilizan este bus, incluso puede implementarse mediante software en cualquier microcontrolador. El peso de la industria de la electrónica de consumo ha permitido precios muy bajos gracias a estos numerosos componentes.

Algunos fabricantes a veces llaman a este bus TWI (interfaz de dos cables) o TWSI (interfaz serie de dos cables).

Los intercambios se realizan siempre entre un único maestro y uno (o todos) los esclavos, siempre por iniciativa del maestro (nunca de maestro a maestro o de esclavo a esclavo). Sin embargo, nada impide que un componente pase del estado de maestro al de esclavo y viceversa.



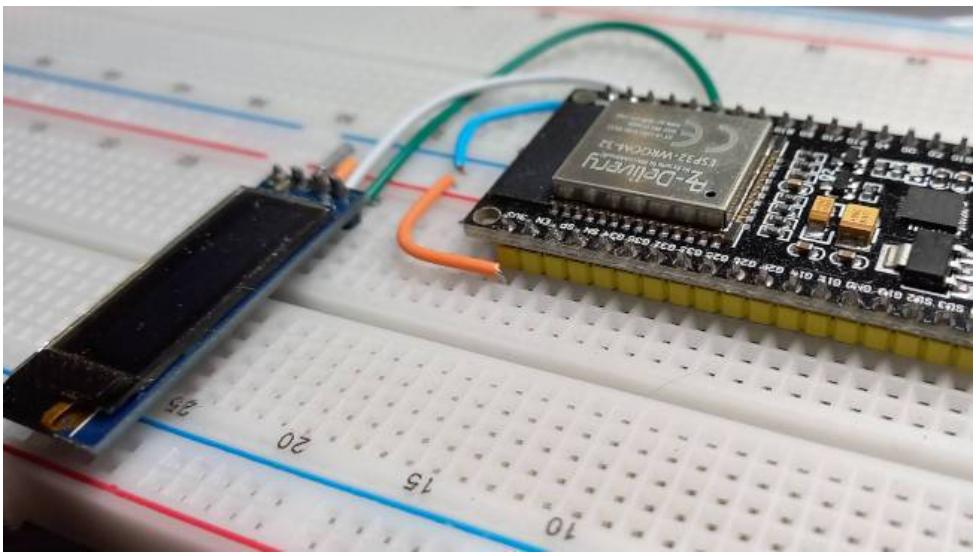
principio de un bus I2C

La conexión se realiza a través de dos líneas:

- SDA (línea de datos serie): línea de datos bidireccional,
- SCL (Serial Clock Line): línea de reloj de sincronización bidireccional.

No debemos olvidar la masa que debe ser común al equipo.

Ambas líneas se tiran al nivel de voltaje VDD a través de resistencias pull-up (RP).



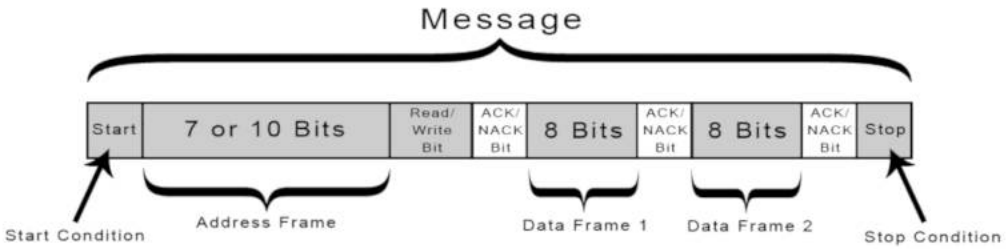
Pantalla OLED conectada al bus I2C

Intercambio de esclavos maestros

El mensaje se puede dividir en dos partes:

- El maestro es el transmisor, el esclavo es el receptor:
  - emisión de una condición de START por el maestro ("S"),
  - transmisión del byte o bytes de dirección por parte del maestro para designar un esclavo, con el bit R/W en 0 (ver la sección sobre direccionamiento a continuación),
  - respuesta del esclavo con un bit de reconocimiento ACK (o bit de no reconocimiento NACK),
  - después de cada reconocimiento, el esclavo puede solicitar una pausa ("PA").
  - emisión de un byte de comando por parte del maestro para el esclavo,
  - respuesta del esclavo con un bit de reconocimiento ACK (o bit de no reconocimiento NACK),
  - emisión de una condición RESTART por parte del maestro ("RS"),
  - transmisión del byte o bytes de dirección por parte del maestro para designar al mismo esclavo, con el bit R/W en 1.
  - respuesta del esclavo con un bit de reconocimiento ACK (o un bit de no reconocimiento NACK).
- El maestro se convierte en receptor, el esclavo en transmisor:
  - emisión de un byte de datos por parte del esclavo para el maestro,

- respuesta del maestro con un bit de reconocimiento ACK (o bit de no reconocimiento NACK),
- transmisión de otros bytes de datos por parte del esclavo con acuse de recibo del maestro,
- para el último byte de datos esperado por el maestro, responde con un NACK para finalizar el diálogo,
- emisión de una condición de STOP por parte del maestro ("P").



Condición de inicio : la línea SDA pasa de un nivel de voltaje alto a un nivel de voltaje bajo antes de que la línea SCL pase de alto a bajo.

Condición de apagado : la línea SDA cambia de un nivel de voltaje bajo a un nivel de voltaje alto después de que la línea SCL cambia de bajo a alto.

Trama de dirección : una secuencia única de 7 o 10 bits para cada esclavo que identifica al esclavo cuando el maestro quiere hablar con él.

Bit de lectura/escritura : un solo bit que especifica si el maestro envía datos al esclavo (nivel de voltaje bajo) o solicita datos (nivel de voltaje alto).

Bit ACK/NACK : cada trama de un mensaje va seguida de un bit de reconocimiento/no reconocimiento. Si se ha recibido correctamente una trama de dirección o una trama de datos, se devuelve un bit ACK al remitente.

Direccionamiento

I2C no tiene líneas de selección de esclavo como SPI, por lo que necesita otra forma de hacerle saber al esclavo que se le están enviando datos a él, y no a otro esclavo. Lo hace dirigiéndose. La trama de dirección es siempre la primera trama después del bit de inicio en un mensaje nuevo.

El maestro envía la dirección del esclavo con el que quiere comunicarse a cada esclavo conectado a él. Luego, cada esclavo compara la dirección enviada por el maestro con la suya. Si la dirección coincide, devuelve un bit ACK de bajo voltaje al maestro. Si la dirección no coincide, el esclavo no hace nada y la línea SDA permanece alta.

Así es como la palabra `Wire.detect` detecta dispositivos conectados al bus i2c.

Puede conectar varios dispositivos diferentes al bus i2c. No se pueden conectar varias copias del mismo dispositivo al mismo bus i2c.

Configuración de puertos GPIO para I2C

Configurar los puertos GPIO para el bus I2C es muy sencillo:

```
\ activar el vocabulario de cables
wire
\ inicia la interfaz I2C usando los pines 21 y 22 en ESP32 DEVKIT V1
\ con 21 utilizados como sda y 22 como scl.
21 22 wire.begin
```

protocolos de bus I2C

El diálogo es sólo entre un amo y un esclavo. Este diálogo siempre lo inicia el maestro (Condición de inicio): el maestro envía la dirección del esclavo con el que quiere comunicarse en el bus I2C.

El diálogo siempre lo finaliza el maestro (condición de parada).

La señal de reloj (SCL) la genera el maestro.

Detectando un dispositivo I2C

Esta parte se utiliza para detectar la presencia de un dispositivo conectado al bus I2C.

Puede compilar este código para probar la presencia de módulos conectados y activos en el bus I2C.

```
\ activa el vocabulario de cables
wire
\ inicia la interfaz I2C usando los pines 21 y 22 en ESP32 DEVKIT V1
\ con 21 para sda y 22 para scl.
21 22 wire.begin

: spaces ( n -- )
    for
        space
    next
  ;

: .## ( n -- )
    <# # # #> type
  ;
```

```
\ no todos los patrones de bits son direcciones i2c de 7 bits
válidas
: Wire.7bitaddr? ( a -- f )
    dup $07 >=
    swap $77 <=  and
  ;

: Wire.detect   ( -- )
    base @ >r hex
    cr
    ."      00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f"
    $80 $00 do
        i $0f and 0=
        if
             cr i .##  ."  : "
        then
        i Wire.7bitaddr? if
            i  Wire.beginTransmission
            -1 Wire.endTransmission 0 =
            if
                i .## space
            else
                ." -- "
            then
        else
            2 spaces
        then
    loop
    cr r> base !
  ;
```

Aquí, ejecutar la palabra **`Wire.detect`** indica la presencia del dispositivo de visualización OLED en la dirección hexadecimal **`3c`** :

```
Wire.detect
     00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
00 :                        -- -- -- -- -- -- -- -- --
10 : -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20 : -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30 : -- -- -- -- -- -- -- -- -- -- -- -- 3c -- -- --
40 : -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50 : -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60 : -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70 : -- -- -- -- -- -- -- --
```

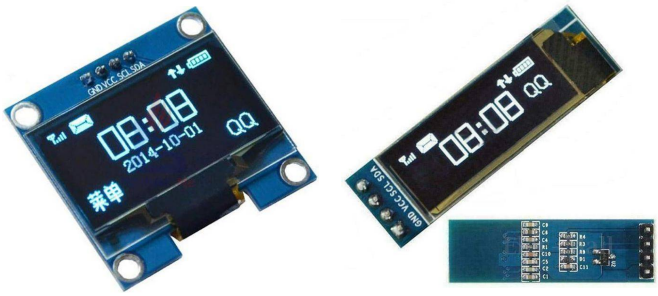
Aquí detectamos un módulo en la dirección hexadecimal `3c` . Esta es la dirección que usaremos para abordar este módulo.

El módulo no se puede detectar a menos que esté físicamente enchufado y encendido.

La pantalla OLED SSD1306

La pantalla OLED existe en dos definiciones:

- Pantalla de 128 x 64 píxeles, monocromática o a color. Si la pantalla es de color, los píxeles permanecen monocromáticos.
- 128 x 32 píxeles, pantalla monocromática.



Estas pantallas están disponibles en interfaz SPI o I2C.

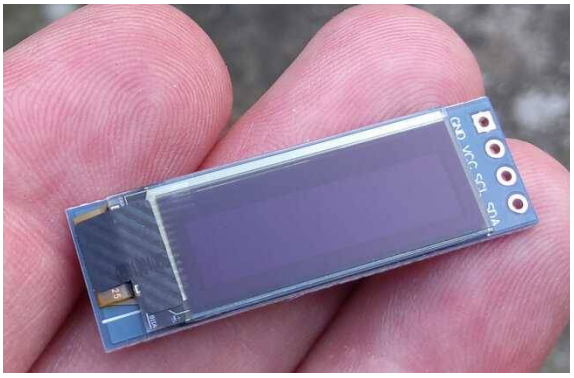
Favorezca la interfaz I2C que permite la conexión de varias interfaces I2C en el mismo dispositivo I2C. El vocabulario **OLED** está diseñado para gestionar la transmisión vía I2C a estas pantallas OLED.

Elegir una interfaz de visualización

La elección de una interfaz de visualización está sujeta a varias condiciones:

- su precio;
- su consumo de electricidad;
- su robustez;
- su facilidad de programación y uso.

Una interfaz de pantalla es muy útil en una configuración independiente para proporcionar información textual o gráfica muy clara.



Después de varias investigaciones, la elección recayó en una pantalla OLED de este tipo

Sólo cuesta unos pocos euros.

Esta pantalla utiliza tecnología OLED, por lo tanto sin retroiluminación.

Su resolución de pantalla es de 128x32 píxeles. Puede mostrar texto e imágenes, pero sólo en monocromo.

En el modo **DISPLAYOFF** , el consumo de energía es casi nulo.

Es un producto muy extendido y bastante bien documentado.

Documentación en línea

- Adafruit: controles y documentación técnica de la pantalla OLED https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf
- Adafruit: biblioteca SSD1306 C https://adafruit.github.io/Adafruit_SSD1306/html/files.html
- Adafruit: SSD1306 microPython https://github.com/adafruit/micropython-adafruit-ssd1306
- Punyforth: SSD1306 SPI en adelante https://github.com/zeroflag/punyforth/blob/master/arch/esp8266/forth/ssd1306-spi.forth
- TG9541: Cuarta pantalla Oled https://github.com/TG9541/forth-oled-display/blob/master/ssd1306.fs
- Yunfan: SSD1306 128x32 i2c en adelante https://gist.github.com/yunfan/2d3ee14697f3ebd3cb43ae411216d9aa

Conexión de la pantalla OLED SSD1306

La pantalla OLED SSD1306 128x32 debe usarse en el bus I2C de la tarjeta ESP32.

Este bus I2C está presente en todas las tarjetas ESP32.

Conexión a una tarjeta ESP32:

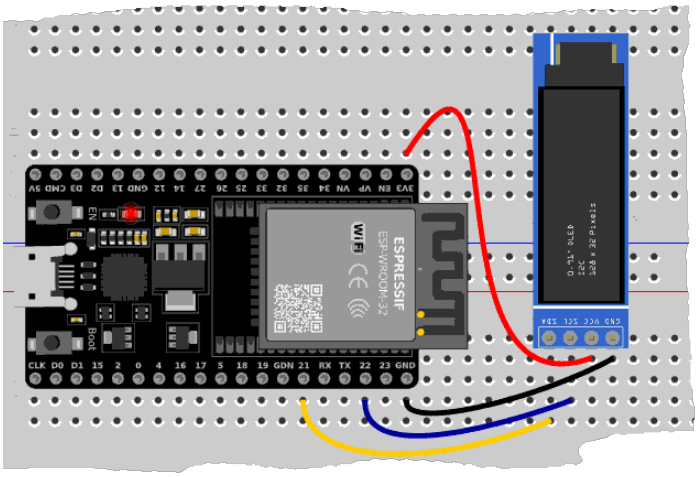


Figure 19: branchement de l'afficheur OLED SSD1306 I2C

Como podemos ver, 4 cables son suficientes: 2 para alimentar la pantalla OLED SSD1306 (cables negro y rojo), 2 para la conexión al bus I2C (cables azul y amarillo).

La fuente de alimentación de la pantalla se toma de la tarjeta ESP32. No es necesario utilizar una fuente de alimentación auxiliar. El bajísimo consumo de energía de esta pantalla lo permite. La pantalla OLED SSD1306 tiene un circuito integrado que devuelve el voltaje de 5V necesario para su funcionamiento.

Organización de la memoria

La pantalla SSD1306 128x32 utiliza el mismo componente interno que la pantalla SSD1306 128x64. La memoria interna es común a ambos modelos, la pantalla de 128x32 sólo utiliza parte de esta memoria.

La memoria interna de la pantalla tiene 1 KB de RAM.

En este diagrama, aquí está la organización de la pantalla para una definición de 128x64 píxeles:

Cada columna contiene 8 bits. Se designa una línea por página:

- la pantalla de 128x64: contiene 8 páginas, numeradas del 0 al 7
- la pantalla de 128x32: contiene 4 páginas, numeradas del 0 al 3

Cada página está dividida en segmentos:

Aquí, en azul en la figura, un segmento representa un byte. La parte menos significativa está arriba.

No necesitamos ir más lejos para gestionar esta pantalla con el vocabulario `OLED` .

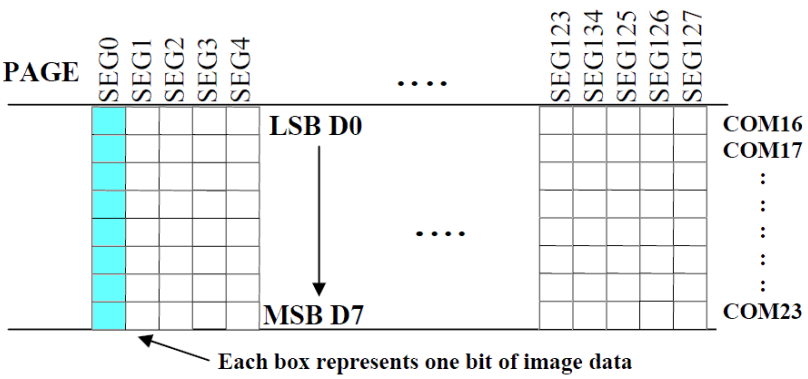


Figure 20: segmentación del espacio de memoria de visualización

Organizar el proyecto SSD1306

Antes de llegar al meollo del asunto, veamos cómo vamos a organizar nuestro proyecto. En su computadora, cree una carpeta de trabajo llamada **display** . En esta carpeta, cree una subcarpeta **SSD1306** .

Aprovecharemos al máximo la gestión de archivos SPIFFS y la construcción de nuestro código FORTH en un proyecto real.

Como requisito previo, debe tener la definición RECORDFILE guardada en el archivo **/spiffs/autoexec.fs** .

Creando el archivo main.fs

subcarpeta **SSD1306 , cree el** archivo **main.fs** y copie este CUARTO código en él:

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

Copie este código nuevamente, inicie el terminal que se comunica con la placa ESP32 y ESP32forth. Copie este código en la terminal. Ejecutarlo. Al final de la ejecución, debería encontrar su archivo main.fs en la tarjeta ESP32:

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

Para verificar que el contenido de **main.fs** se haya guardado en el sistema de archivos SPIFFS:

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

debería mostrar el contenido del archivo **/spiffs/main.fs** .

Creando el archivo config.fs

subcarpeta **SSD1306 , cree el** archivo **config.fs** y copie este CUARTO código en él:

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

Al igual que con **main.fs** , pase este contenido a través de la terminal para guardar el nuevo archivo **config.fs** .

Creando el archivo oledTools.fs

Por ahora, aquí está nuestro archivo **oledTools.fs final** para crear en la computadora y transmitir a la placa ESP32:

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

Pruebe nuestro proyecto SSD1306

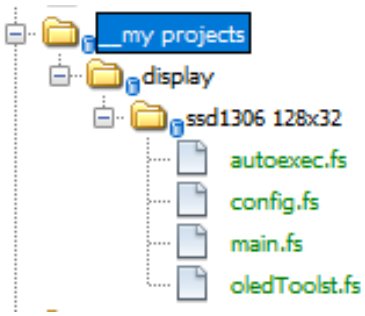


Figure 21: fichiers du projet

Aquí está la estructura de nuestro proyecto en el disco de nuestra computadora.

Todos los archivos de extensión fs se pasaron a la placa ESP32 para guardarlos en el sistema de archivos SPIFFS.

Para compilar el contenido del archivo /SPIFFS/config.fs, puede probarlo desde la ventana del terminal que se comunica con ESP32 en adelante:

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

Si nunca modifica el contenido del archivo **config.fs** , siempre estará disponible para ESP32 tan pronto como se encienda la placa ESP32.

¿Recuerda que también pasamos un archivo **main.fs** ? El contenido de este archivo debe reservarse para llamar a los diferentes archivos del proyecto. Recordatorio del contenido de nuestro archivo main.fs tal como está guardado en la tarjeta ESP32 en el sistema de archivos SPIFFS:

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

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

Las dos primeras líneas le permiten gestionar un marcador. Cada vez que se interpreta el contenido de **main.fs** , ESP32forth probará si hay una palabra `--oledTest` . Si esta

palabra existe, será eliminada del diccionario. Todas las palabras compiladas después de `--oledTest` se eliminarán del diccionario.

En la segunda línea, recreamos la palabra `--oledTest` . No es sorprendente eliminar esta palabra y recrearla. De esta forma, cada vez que se interpreta el contenido de **main.fs** , el diccionario de ESP32 en adelante se reinicia con contenido que no interrumpirá nuestro proyecto.

Finalmente, en las dos últimas líneas de **main.fs** , se le pide a ESP32forth que procese el contenido de los archivos **config.fs** y **oledTools.fs** . Entonces, para iniciar este procesamiento global, podemos escribir:

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

Cuando tienes un proyecto complejo, es posible que tengas que escribir este tratamiento docenas o incluso cientos de veces. ¿Recuerdas que en el archivo **autoexec.fs** definimos la palabra **MAIN** ? Recordatorio de la definición de esta palabra:

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

Demasiado bien ! Entonces podemos simplemente escribir `MAIN` en lugar de `incluir /spiffs/main.fs…`

¿Hacemos la prueba? DE ACUERDO. Apague la placa ESP32. Vuelva a encenderlo. Abra la terminal que se comunica con ESP32 en adelante y escriba MAIN. ¡Todo el contenido del proyecto se ejecuta y compila casi al instante! Escriba `words`. Todas las palabras de nuestro proyecto están en el `forth` vocabulario.

Comprobamos que funciona tecleando ahora:

```
Oled128x32Init
```

Si la pantalla OLED SSD1306 de 128x32 píxeles está conectada correctamente, debería mostrar `*Esp32forth*` :

A partir de este momento podemos escribir y realizar todas las demás palabras de nuestro

A partir de este momento podemos escribir y realizar todas las demás palabras de nuestro proyecto.

Utilice vocabulario OLED

Las palabras que usaremos para usar nuestra pantalla OLED SSD1306 128x32 están disponibles en el vocabulario **OLED** :

```
--> oled vlist
OledInit SSD1306_SWITCHCAPVCC SSD1306_EXTERNALVCC WHITE BLACK OledReset
HEIGHT WIDTH OledAddr OledNew OledDelete OledBegin OledHOME OledCLS OledTextc
OledPrintln OledNumln OledNum OledDisplay OledPrint OledInvert OledTextsize
OledSetCursor OledPixel OledDrawL OledFastHLine OledFastVLine OledCirc
OledCircF OledRect OledRectF OledRectR OledRectRF oled-builtins
```

Inicializando el bus I2C para la pantalla OLED SSD1306

Nuestra pantalla OLED está conectada al bus I2C. Por tanto, está disponible en la dirección hexadecimal 3c en este bus I2C. Por tanto, lo primero que hay que hacer es definir una constante:

```
\ establecer dirección de pantalla OLED SSD1306 128x32 píxeles
$3c constant I2C_SSD1306_ADDRESS
```

En el vocabulario oled existe una variable **OledAddr** encargada de memorizar esta dirección 3c. Si esta dirección contiene un valor cero es porque el bus I2C no ha establecido conexión con nuestro display SSD1306. Es este valor nulo el que condiciona la inicialización de esta conexión:

```
    OledAddr @ 0=
    if
        \ in
    then
```

Aquí está la parte us I2C a la pantalla OLED SS

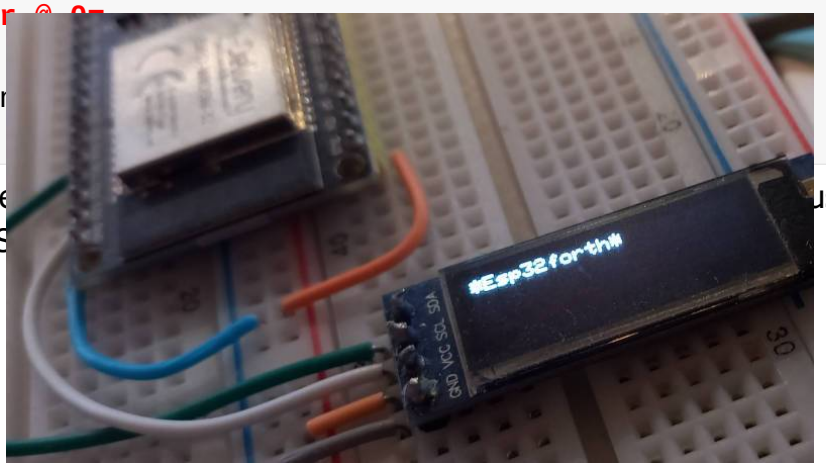


Figure 22: resultado de ejecutar la palabra Oled128x32Init

```
WIDTH HEIGHT OledReset OledNew
SSD1306_SWITCHCAPVCC I2C_SSD1306_ADDRESS OledBegin drop
```

- secuencia **`WIDTH HEIGHT OledReset OledNew`** crea una instancia de una nueva sesión OLED para nuestra pantalla SSD1306;
- la palabra **`OledBegin`** irá precedida de dos parámetros:
  - **`SSD1306_SWITCHCAPVCC`** que confirma el suministro de 3.3V de la tarjeta ESP32, como es el caso en nuestro ensamblaje. Si hubiéramos utilizado una fuente de alimentación externa, reemplazamos esta palabra por **`SSD1306_EXTERNALVCC`** .
  - **`I2C_SSD1306_ADDRESS`** que indica la dirección de la pantalla OLED en el bus I2C.

Esta inicialización del bus I2C sólo se realiza una vez para nuestra pantalla OLED SSD1306.

Inicializando la pantalla para SSD1306

Para iniciar una visualización, inicializaremos la visualización:

- **`OledCLS`** que solicita la eliminación del contenido de la pantalla;
- **`1 OledTextsize`** que indica el tamaño del texto a mostrar;
- **`WHITE OledTextc`** que indica el color del texto a mostrar. Para la pantalla SSD1306, solo hay colores **`WHITE`** y **`BLACK`** .

Aquí está la palabra **`Oled128x32Init`** que permite una inicialización adecuada:

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

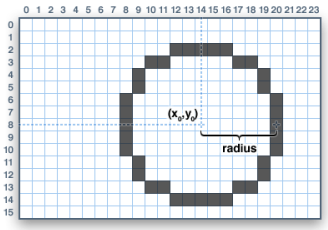
Al ejecutar **`Oled128x32Init`** debería mostrarse el texto `*Esp32forth*` en la pantalla OLED.

A continuación se muestra un resumen de los comandos de administración de visualización de texto para la pantalla OLED:

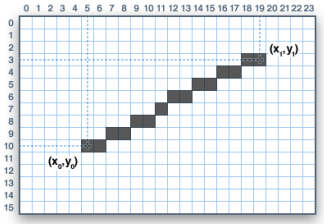
- **`OledCLS`** (--)
  Borra el contenido de la pantalla OLED
- **`OledDisplay`** (--)
  transmite a la pantalla OLED los comandos en espera de visualización
- **`OledHOME`** (--)
  Coloca el cursor en la fila 0, columna 0 de la pantalla OLED. Esta posición es perfecta en píxeles.
- **`OledInvert`** (--)
  Invierte la visualización de la pantalla OLED
- **`OledNum`** (n --)
  Muestra el número n como una cadena en la pantalla OLED
- **`OledNumln`** (n --)
  Muestra un número entero en la pantalla OLED y pasa a la siguiente línea
- **`OledPrint`** (z-string --)
  Muestra el texto de la cadena z en la pantalla OLED
- **`OledPrintln`** (z-string --)
  Imprime el texto de la cadena z en la pantalla OLED y pasa a la siguiente línea
- **`OledTextc`** (BLANCO|NEGRO --)
  Establece el color del texto que se mostrará
- **`OledSetCursor`** (xy --)
  Establece la posición del cursor
- **`OledTextsize`** (size -)
  Establece el tamaño del texto que se mostrará en la pantalla OLED. El valor de n debe estar en el intervalo [1..3]. Para texto de tamaño normal, tamaño=1. Si excede el valor 4, el texto se truncará en una visualización de 4 líneas.

Y aquí hay un resumen de los comandos de gestión de la pantalla gráfica:

- **`OledCirc`** (color del radio xy --)
  Dibuja un círculo centrado en xy, con radio radio y color color (0|1)



- **`OledCircF`** (color del radio xy --)
  Dibuja un círculo completo centrado en xy, con radio radio y color color (0|1)
- **`OledDrawL`** (x0 y0 x1 y1 color --)
  Dibuja una línea de x0 y0 a x1 y1 de color color.



- **`OledFastHLine`** (color de longitud xy --)
  Dibuja una línea horizontal desde xy de longitud de dimensión y color de color.
- **`OledFastVLine`** (color de longitud xy --)
  Dibuja una línea vertical desde xy de longitud de dimensión y color de color.
- **`OledPixel`** (color xy)
  Activa un píxel en la posición x y. El parámetro de color determina el color del píxel.
- **`OledRect`** (x y ancho alto color -)
  Dibuja un rectángulo vacío desde la posición xy de tamaño ancho alto y color color.
- **`OledRectF`** (x y ancho alto color -)
  Dibuja un rectángulo sólido desde la posición xy de tamaño ancho alto y color color.
- **`OledRectR`** (x y ancho alto radio color -)
  Dibuja un rectángulo vacío con esquinas redondeadas, desde la posición xy, de dimensión ancho alto, en el color del color, con un radio radio.
- **`OledRectRF`** (x y ancho alto radio color -)
  Traza un rectángulo sólido con esquinas redondeadas, desde la posición xy, de dimensión ancho alto, en el color del color, con un radio radio.

La pantalla OLED permite ejecutar palabras de gestión de texto y gráficos en el mismo modo de visualización. En resumen, puedes mezclar texto y gráficos.

Ampliar el vocabulario OLED

Después de algunos fallos, descubrí que no se puede definir una extensión `OledTriangle` como esta en lenguaje C para ampliar las definiciones en el archivo **oled.h** :

```
YV(oled, OledTriangle, oled_display->drawTriangle(n5, n4, n3, n2, n1, n0); DROPn(6)) \
```

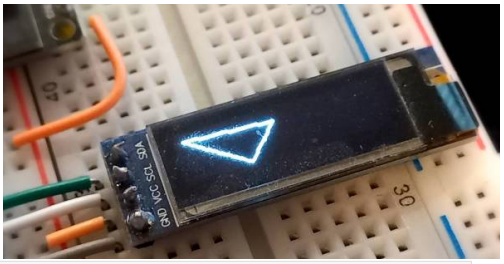
pero eso sin tener en cuenta que programamos en CUARTO lenguaje y que con este lenguaje podemos ampliar nuestro vocabulario OLED. Por lo tanto, crearemos un nuevo archivo en nuestra computadora, en el directorio de nuestro proyecto, con el nombre de archivo **extendOledVoc.fs** . Contenido :

```
RECORDFILE /spiffs/extendOledVoc.fs
oled definitions
: OledTriangle { x0 y0 x1 y1 x2 y2 color -- }
    x0 y0 x1 y1 color OledDrawL
    x1 y1 x2 y2 color OledDrawL
    x2 y2 x0 y0 color OledDrawL
  ;
forth definitions
<EOF>
```

Luego copiamos y pegamos este código y lo transmitimos a ESP32 a través del terminal que se comunica con la tarjeta ESP32. Deberíamos terminar con un archivo **extendOledVoc.fs** en el espacio de memoria SPIFFS. Modificamos ahora el contenido del archivo **main.fs** :

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

Desconectamos y volvemos a conectar la tarjeta ESP32. Escribimos `MAIN` en la terminal. Si todo ha ido bien, deberías encontrar la palabra `OledTriangle` simplemente escribiendo `oled vlist` .



Prueba de nuestra nueva palabra `OledTriangle` :

```
oled
OledCLS OledDisplay
5 5 60 8 40 30 WHITE OledTriangle OledDisplay
```

TEMPVS FVGIT[8]

¿Y si los romanos hubieran podido programar la visualización del tiempo en forma digital?

Este es un proyecto interesante que combina varios archivos. En este capítulo, no proporcionaremos todo el código utilizado aquí. Sería demasiado largo.

Los códigos fuente de este capítulo se encuentran en este archivo:

- **ESP32forth-book.zip** → proyectos → tempusFugit

Enlace: https://github.com/MPETREMANN11/ESP32forth/blob/main/__documentation/ESP32forth-book.zip

Todo el proyecto contiene estos archivos:

- **autoexec.fs** cargado cuando se inicia ESP32fortth
- **clepsydra.fs** conversión de números a números romanos
- **config.fs** ajustes de configuración global
- **main.fs** cargando los otros archivos del proyecto
- **oledTools.fs** completa el vocabulario `oled`
- **RTClock.fs** gestiona el reloj en tiempo real
- **strings.fs** maneja el procesamiento de cadenas alfanuméricas

La secuencia de carga de archivos de proyecto está escrita en **main.fs** :

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

La mayoría de los archivos de este proyecto son independientes, excepto **clepsydra.fs** , que depende de **strings.fs** .

Romaní non ustulo nulla[9]

Los romanos no conocían el número 0. Entonces, ¿cómo podemos mostrar **las 13:00** o **las 00:15** en números romanos?

Para solucionar el problema de las horas posteriores a la medianoche, por ejemplo las 00:15, los japoneses

8 Tempus fugit = el tiempo pasa
9 Romani non ustulo nulla = Los romanos no conocían el cero

(residentes en JAPÓN) nos serán de gran ayuda. Si alguna vez vas a este país, ¡te sorprenderá ver las tiendas abiertas hasta las **25:00** !

¡Esta tienda está abierta de 09:00 a 25:00! Oh sí. Sin embargo, los relojes de JAPÓN también son de 24 horas. Sabíamos que los japoneses eran trabajadores, pero en cuanto a trabajar jornadas de 25 horas, tenemos derecho a tener algunas dudas...

De hecho, hay una explicación muy lógica. Después de las 12:00, son las 12:01, etc... Y así, después de las 23:59, son las 24:00, luego las 24:01. Entonces, si una tienda cierra a las 25:00, debemos entender que para nosotros cierra a la 01:00.

Si transponemos esto a nuestro reloj romano, cuando sean las **00:00** , podremos mostrar **XXIV** o mejor **XXIII:LX** (23:60).

Horas y minutos en romaní[10]

Para resolver el caso de horas como 01:00, 02:00... 23:00, lógicamente, después de las 12:59, podemos mostrar muy bien las 12:60, luego el minuto después de las 13:01... 12:60 en Números romanos: **XXII:LX** .

Esto es lo que se consigue en la palabra `tempusTo$` :

```
: tempusTo$ { HH MM -- }
    HH 0 =  MM 0=  AND if
        60  to MM
        23  to HH
    THEN
    HH 0 >  MM 0=  AND if
        60  to MM
        -1 +to HH
    then
    HH 0 <= if
        24 to HH
    then
    HH roman tempus $!
    [char] : tempus c+$!            \ add char :
    MM roman tempus append$
    tempus
  ;
```

En la primera prueba si...entonces, probamos si estamos a las **00:00** . En este caso forzamos la hora a **23** y los minutos a **60** .

En la segunda prueba, si la hora es mayor que 00 y los minutos son mayores que 00, disminuimos la hora y forzamos los minutos a **60** . La desventaja es que si la hora es 00, la cambiamos a -1.

10 Romani horas et minuta = horas y minutos romanos

En la última prueba, si la hora es cero o negativa, la forzamos a **24** .

Podemos utilizar la palabra `.tempus` que se utilizó en el desarrollo para comprobar este correcto funcionamiento:

```
--> 23 59 .tempus
XXIII:LIX ok
--> 0 0 .tempus
XXIII:LX ok
--> 0 1 .tempus
XXIV:I ok
--> 1 0 .tempus
XXIV:LX ok
--> 1 1 .tempus
I:I ok
```

Haec omnia integramus pro ESP32forth[11]

En el estado actual del proyecto, debe ingresar manualmente la hora inicial:

```
22 19 start
```

Significa que inicializamos la hora a las **22:19** . Esta inicialización se realiza de forma muy sencilla:

```
0 RTC.setTime
```

Luego inicializamos la pantalla OLED de 128x32:

```
Oled128x32Init
1 OledTextsize
WHITE OledTextc
```

Y finalmente, recuperaremos la hora actual y la mostraremos:

```
OledCLS OledDisplay
16 20 OledSetCursor
RTC.getTime drop tempusTo$ s>z OledPrintln OledDisplay
```

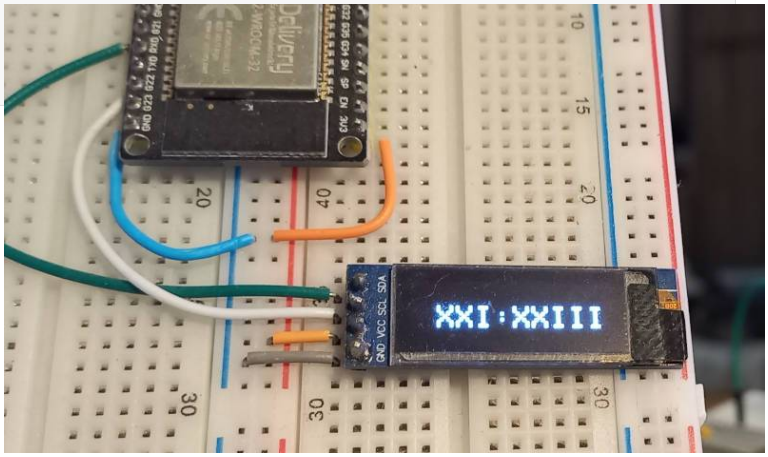
Hice pruebas con una pantalla más grande. El problema, para el canal **XXIII:XXVIII** , no hay suficiente espacio para mostrar este canal.

Aquí está el bucle final que se debe ejecutar para comenzar a mostrar la hora en números romanos:

```
oled
: start ( HH MM -- )
    0 RTC.setTime       \ define current time
    Oled128x32Init
    1 OledTextsize
    WHITE OledTextc
```

11 Haec omnia integramus pro ESP32forth = Integramos todo esto para ESP32forth

```
    begin
        OledCLS OledDisplay
        16 20 OledSetCursor
        RTC.getTime drop tempusTo$ s>z OledPrintln OledDisplay
        1000 ms
    key? until
  ;
forth
```



El programa se puede mejorar recuperando la hora de un servidor de hora. Ver capítulo *Recuperar la hora desde un servidor WEB* .

También es posible utilizar funciones de `timers` para liberar al intérprete. Consulte el capítulo *Cómo hacer parpadear un LED mediante temporizador*.

Finalmente, ensamblar los diferentes archivos para este proyecto y algunas pruebas y ajustes me llevó una tarde.

Sin embargo, me gustaría enfatizar algunos puntos:

- Siempre haga una copia en la carpeta de su proyecto de un archivo de propósito general. Por ejemplo, para el archivo **strings.fs** ;
- Si copia un componente general, por ejemplo **strings.fs** o **RTClock.fs** , solo realice cambios en los archivos copiados en la carpeta de su proyecto. Versione estas modificaciones e indique la fecha de modificación en el comentario del encabezado del archivo modificado.

A medida que realizas tus proyectos, es posible que te encuentres con el mismo archivo copiado en diferentes carpetas y modificado. Esta solución es preferible a un único archivo lleno de parámetros de ajuste.

Agregue la biblioteca SPI

La biblioteca SPI no está implementada de forma nativa en ESP32 en adelante. Para instalarlo, primero debes crear el archivo **spi.h** el cual debe instalarse en la misma carpeta que contiene el archivo **ESP32forth.ino** .

Contenido del archivo **spi.h** (en lenguaje C) :

```
# include <SPI.h>

#define OPTIONAL_SPI_VOCABULARY V(spi)
#define OPTIONAL_SPI_SUPPORT \
  XV(internals, "spi-source", SPI_SOURCE, \
      PUSH spi_source; PUSH sizeof(spi_source) - 1) \
  XV(spi, "SPI.begin", SPI_BEGIN, SPI.begin((int8_t) n3, (int8_t) n2, (int8_t) n1, (int8_t) n0); DROPn(4)) \
  XV(spi, "SPI.end", SPI_END, SPI.end();) \
  XV(spi, "SPI.setHwCs", SPI_SETHWCS, SPI.setHwCs((boolean) n0); DROP) \
  XV(spi, "SPI.setBitOrder", SPI_SETBITORDER, SPI.setBitOrder((uint8_t) n0); DROP) \
  XV(spi, "SPI.setDataMode", SPI_SETDATAMODE, SPI.setDataMode((uint8_t) n0); DROP) \
  XV(spi, "SPI.setFrequency", SPI_SETFREQUENCY, SPI.setFrequency((uint32_t) n0); DROP) \
  XV(spi, "SPI.setClockDivider", SPI_SETCLOCKDIVIDER, SPI.setClockDivider((uint32_t) n0); DROP) \
  XV(spi, "SPI.getClockDivider", SPI_GETCLOCKDIVIDER, PUSH SPI.getClockDivider();) \
  XV(spi, "SPI.transfer",   SPI_TRANSFER, SPI.transfer((uint8_t *) n1, (uint32_t) n0); DROPn(2)) \
  XV(spi, "SPI.transfer8",  SPI_TRANSFER_8,  PUSH (uint8_t)  SPI.transfer((uint8_t) n0); NIP) \
  XV(spi, "SPI.transfer16", SPI_TRANSFER_16, PUSH (uint16_t) SPI.transfer16((uint16_t) n0); NIP) \
  XV(spi, "SPI.transfer32", SPI_TRANSFER_32, PUSH (uint32_t) SPI.transfer32((uint32_t) n0); NIP) \
  XV(spi, "SPI.transferBytes", SPI_TRANSFER_BYTES, SPI.transferBytes((const uint8_t *) n2, (uint8_t *) n1, (uint32_t) n0);
DROPn(3)) \
  XV(spi, "SPI.transferBits", SPI_TRANSFER_BITES, SPI.transferBits((uint32_t) n2, (uint32_t *) n1, (uint8_t) n0); DROPn(3)) \
  XV(spi, "SPI.write", SPI_WRITE, SPI.write((uint8_t) n0); DROP) \
  XV(spi, "SPI.write16", SPI_WRITE16, SPI.write16((uint16_t) n0); DROP) \
  XV(spi, "SPI.write32", SPI_WRITE32, SPI.write32((uint32_t) n0); DROP) \
  XV(spi, "SPI.writeBytes", SPI_WRITE_BYTES, SPI.writeBytes((const uint8_t *) n1, (uint32_t) n0); DROPn(2)) \
  XV(spi, "SPI.writePixels", SPI_WRITE_PIXELS, SPI.writePixels((const void *) n1, (uint32_t) n0); DROPn(2)) \
  XV(spi, "SPI.writePattern", SPI_WRITE_PATTERN, SPI.writePattern((const uint8_t *) n2, (uint8_t) n1, (uint32_t) n0);
DROPn(3))

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

El archivo completo también está disponible aquí:
https://github.com/MPETREMANN11/ESP32forth/blob/main/optional/spi.h

Cambios en el archivo ESP32forth.ino

archivo **spi.h** no se puede integrar en ESP32forth sin realizar algunos cambios en el archivo **ESP32forth.ino** . Estas son las pocas modificaciones que se deben realizar en este archivo. Estos cambios se realizaron en la versión 7.0.7.15, pero deberían aplicarse a otras versiones recientes o futuras.

Primera modificación

Código agregado en rojo:

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

```
  V(rtos) V(SPIFFS) V(serial) V(SD) V(SD_MMC) V(ESP) \
  V(ledc) V(Wire) V(WiFi) V(sockets) \
  OPTIONAL_CAMERA_VOCABULARY \
  OPTIONAL_BLUETOOTH_VOCABULARY \
  OPTIONAL_INTERRUPTS_VOCABULARIES \
  OPTIONAL_OLED_VOCABULARY \
  OPTIONAL_SPI_VOCABULARY \
  OPTIONAL_RMT_VOCABULARY \
  OPTIONAL_SPI_FLASH_VOCABULARY \
  USER_VOCABULARIES
```

Segunda modificación

Adición en rojo después de este código:

```
// Hook to pull in optional Oled support.
# if __has_include("oled.h")
#  include "oled.h"
# else
#  define OPTIONAL_OLED_VOCABULARY
#  define OPTIONAL_OLED_SUPPORT
# endif

// Hook to pull in optional SPI support.
# if __has_include("spi.h")
#  include "spi.h"
# else
#  define OPTIONAL_SPI_VOCABULARY
#  define OPTIONAL_SPI_SUPPORT
# endif
```

Tercera modificación

Adición en rojo:

```
#define EXTERNAL_OPTIONAL_MODULE_SUPPORT \
  OPTIONAL_ASSEMBLERS_SUPPORT \
  OPTIONAL_CAMERA_SUPPORT \
  OPTIONAL_INTERRUPTS_SUPPORT \
  OPTIONAL_OLED_SUPPORT \
  OPTIONAL_SPI_SUPPORT \
  OPTIONAL_RMT_SUPPORT \
  OPTIONAL_SERIAL_BLUETOOTH_SUPPORT \
  OPTIONAL_SPI_FLASH_SUPPORT
```

Cuarta modificación

Adición en rojo:

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

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

Si sigue estas instrucciones cuidadosamente, podrá compilar ESP32 con ARDUINO IDE y cargarlo en la placa ESP32. Una vez realizadas estas operaciones, inicie la terminal. Debe encontrar el mensaje de bienvenida de ESP32 en adelante. Tipo:

```
spi vlist
```

Debes encontrar las palabras definidas en este vocabulario **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
```

Ahora puede controlar extensiones a través del puerto SPI, como las pantallas LED MAX7219.

Comunicarse con el módulo de visualización MAX7219

En la comunicación SPI siempre hay un maestro *que* controla los periféricos (también llamados *esclavos*). Los datos se pueden enviar y recibir simultáneamente. Esto significa que el maestro puede enviar datos a un esclavo y un esclavo puede enviar datos al maestro al mismo tiempo.

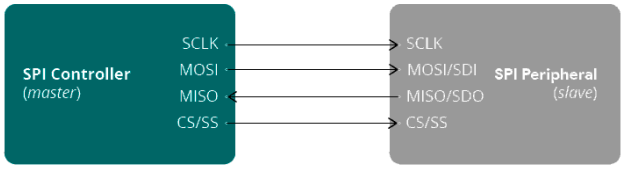


Figure 24: controlar un dispositivo SPI

Puedes tener varios esclavos. Un esclavo puede ser un sensor, una pantalla, una tarjeta microSD, etc. u otro microcontrolador. Esto significa que puedes tener tu ESP32 conectado a **múltiples dispositivos** .

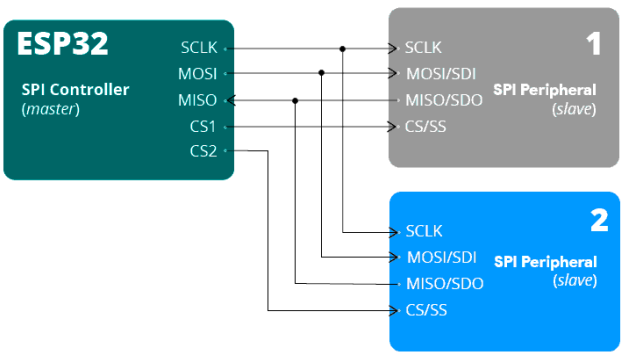


Figure 23: control de dos dispositivos SPI

Un esclavo se selecciona configurando el selector CS1 o CS2 en nivel bajo. Se necesitarán tantos selectores de CS como esclavos para gestionar.

Localización del puerto SPI en la placa ESP32

Hay dos puertos SPI en una placa ESP32: HSPI y VSPI. El puerto SPI que gestionaremos es aquel cuyos pines tienen el prefijo VSPI:

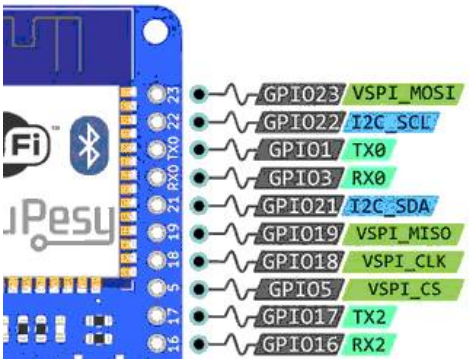


Figure 25: los dos puertos SPI en la placa ESP32

Con ESP32 en adelante, podemos definir las constantes que apuntan a estos pines VSPI:

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

Para comunicarnos con el módulo de visualización MAX7219, solo necesitaremos cablear los pines VSPI_MOSI, VSPI_SCLK y VSPI_CS.

Conectores SPI en el módulo de pantalla MAX7219

Aquí está el mapa del conector del puerto SPI en el módulo MAX7219:

Figure 26: connecteurs SPI sur le module MAX7219

Conexión entre el módulo MAX7219 y la tarjeta ESP32:

```
          MAX7219              ESP32
            DIN     <---->   VSPI_MOSI
```

```
CS      <---->   VSPI_CS
CLK     <---->   VSPI_SCLK
```

Los conectores VCC y GND están conectados a una fuente de alimentación externa:

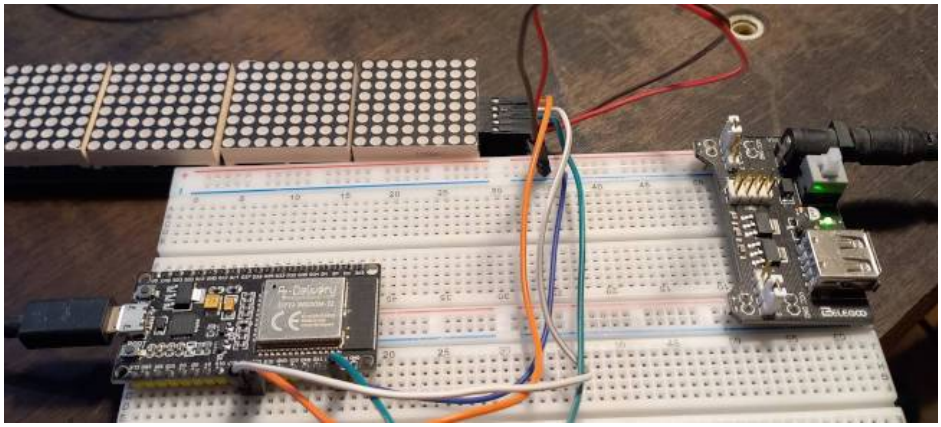


Figure 27: utilizando una fuente de alimentación externa

La parte GND de esta fuente de alimentación externa se comparte con el pin GND de la tarjeta ESP32.

Capa de software del puerto SPI

Todas las palabras para gestionar el puerto SPI ya están disponibles en el vocabulario `spi` .

Lo único que hay que definir es la inicialización del puerto SPI:

define SPI port frequency

```
4000000 constant SPI_FREQ

\ select SPI vocabulary
only FORTH  SPI also

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

Ahora estamos listos para usar nuestro módulo de visualización MAX7219.

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(bluetooth) 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

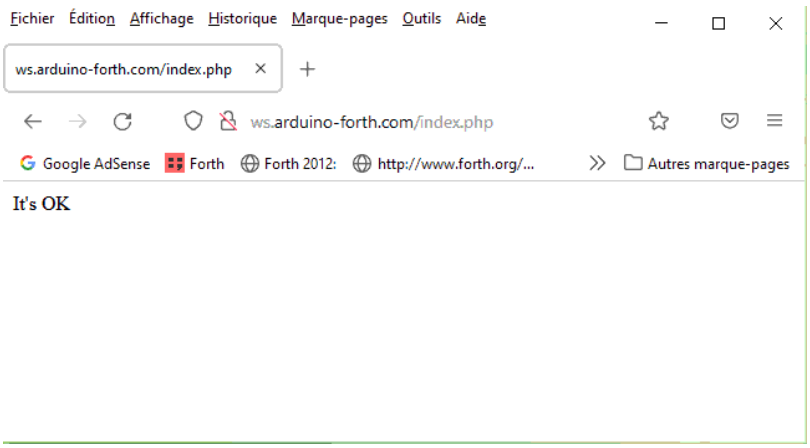
Para probar nuestro cliente HTTP, podemos hacerlo consultando cualquier servidor web. Pero para lo que veremos más adelante, es necesario tener un servidor web personal. En este servidor, creamos un subdominio:

- 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.

Recuperar la hora desde un servidor WEB

En el capítulo *Software de reloj en tiempo real* , vimos cómo administrar un reloj en tiempo real usando las propiedades del **timer**.

Sin embargo, la inicialización de este reloj de tiempo real debe realizarse manualmente. Ahora que tenemos una forma de comunicarnos con un servidor web, veremos cómo realizar esta inicialización a través de un servidor web.

Transmisión y recepción de hora desde un servidor web.

Para la parte del servidor, creamos un nuevo script **gettime.php** cuyo contenido es el siguiente:

```
<?php
echo date('H i s')." RTC.set-time";
```

Si ejecutamos este script http://ws.arduino-forth.com/gettime.php , en un navegador web, esto es lo que se muestra:

```
15 25 30 RTC.set-time
```

Preparamos el trabajo para que el intérprete ESP32Forth solo tenga esta línea para ejecutar. Aquí está el código FORTH para recuperar la hora:

```
WiFi
\ connection to local WiFi LAN
: myWiFiConnect
    z" mySSID"
    z" myWiFiCode"
    login
  ;

Forth

0 value currentTime

\ store current time
: RTC.set-time { hh mm ss -- }
    hh 3600 *
    mm 60 *
    ss + +  1000 *
    MS-TICKS - to currentTime
  ;
```

```
\ used for SS and MM part of time display
: :## ( n -- n' )
    #  6 base ! #  decimal  [char] : hold
  ;

\ display current time
: RTC.display-time ( -- )
    currentTime MS-TICKS + 1000 /
    <# :## :## 24 mod #S #> type
  ;

700 constant bufferSize
create httpBuffer
    bufferSize allot

0 buffer 700 erase

HTTP

: getTime
    cr
    z" http://ws.arduino-forth.com/gettime.php" HTTP.begin
    if
        HTTP.doGet
        if
            httpBuffer bufferSize HTTP.getPayload
            httpBuffer z>s evaluate
        then
    then
    HTTP.end
  ;

myWiFiConnect
getTime
RTC.display-time
```

En la palabra **`getTime`** , esta secuencia **`httpBuffer z>s evaluate`** recupera el contenido del búfer de transacciones web y evalúa su contenido. Esto es posible porque el servidor web transmitió una secuencia compatible con nuestro intérprete FORTH. Al ejecutar las últimas tres líneas de este código se muestra esto:

```
--> myWiFiConnect
192.168.1.23
MDNS started
 ok
```

```
--> getTime
 ok
--> RTC.display-time
15:33:09 ok
```
Esta inicialización se puede realizar solo una vez, generalmente al iniciar ESP32Forth. Esta técnica de consultar nuestro propio servidor web evita negociar con un servidor horario.

La mayoría de los servidores de tiempo entregan información en formatos que son difíciles de procesar con FORTH: csv, JSON, XML, etc.

Comprender la transmisión por GET a un servidor WEB

Transmisión de datos a un servidor mediante GET

Hay dos métodos para transmitir datos desde una página web a un servidor web:

POST que es el método utilizado generalmente desde formularios

GET cuál es el método que vamos a estudiar

Existen otros métodos, pero generalmente están reservados para transacciones de máquina a máquina a través de servicios web.

Parámetros en una URL

Empecemos explicando qué es una URL: http://mi-sitioweb.com/ (URL por ejemplo).

Analizamos una URL empezando por el final:

- **.com** es el TLD (dominio de nivel superior)
- **mi-sitio web** es el nombre de dominio
- **http://** es el protocolo de comunicación.

No vamos a hacer un curso exhaustivo sobre estos elementos. Lo único que hay que saber llega ahora.

Esta URL puede ir seguida del script o de la página HTML, por ejemplo: http://my-website.com/index.php

Podemos completar esta URL con un paso de parámetro:

```
http://my-website.com/index.php?temp=32.7
```

Aquí pasamos un parámetro **temp** cuyo valor es **32.7** .

El paso de parámetros mediante el método GET está marcado con el signo ?.

Pasando múltiples parámetros

Se pueden transmitir varios parámetros separándolos con el signo `&`:

```
http://my-website.com/index.php?log=myLog&pwd=myPassWd&temp=32.7
```

Aquí pasamos tres parámetros:

log con el valor myLog

pwd con el valor myPassWd

temp con valor 32,7

Para entender cómo el servidor recibirá estos datos, creamos un script **record.php** que provisionalmente simplemente contendrá esto:

```
<?php
var_dump($_GET);
```

y que mostrará esto si consultamos este script con nuestro navegador web favorito:

```
array(3) {
  ["log"]=>
  string(7) "myLogin"
  ["pwd"]=>
  string(10) "mypassword"
  ["temp"]=>
  string(4) "32.7"
}
```

Eso es prácticamente todo lo que necesitamos para obtener los datos y guardarlos en el servidor. Esto es lo que vamos a descubrir...

Gestión del paso de parámetros con ESP32forth

Para empezar, es necesario disponer de palabras para gestionar cadenas de caracteres. Encontrarás estas palabras en el *capítulo Visualización de números y cadenas de caracteres,* parte *Código de palabras para la gestión de variables de texto* .

Empezamos creando una cadena de caracteres:

```
256 string myUrl
s" http://ws.arduino-forth.com/record.php?
log=myLog&pwd=myPassWd&temp="
myUrl $!
```

Acabamos de definir una variable alfanumérica **`myUrl`** . Esta variable está casi completa. Lo único que falta es el valor del parámetro **`temp`**. Para agregar este valor, ejecutaremos **`append$`** :

```
s" 32.5" myUrl append$
myUrl type
```

```
\ display: http://ws.arduino-forth.com/record.php?
log=myLog&pwd=myPassWd&temp=32.5
```

Esta es la URL que usaremos en esta definición:

```
: sendData ( str -- )
    s" http://ws.arduino-forth.com/record.php?
log=myLog&pwd=myPassWd&temp="
    myUrl $!
    myUrl append$
    \  cr myUrl type
    myUrl s>z HTTP.begin
    if
        HTTP.doGet dup 200 =
        if  drop
            httpBuffer bufferSize HTTP.getPayload
            httpBuffer z>s type
        else
            cr ." CNX ERR: " .
        then
    then
    HTTP.end
  ;

myWiFiConnect
s" 32.65" sendData
```

La palabra **`sendData`** recupera el contenido de la cadena, aquí **`32.65`** , concatena este contenido en **`myUrl`** y luego inicia una transacción de cliente web con el servidor mencionado en **`myUrl`** .

Notarás que en la URL hay un parámetro de registro. Este parámetro puede ser diferente para cada tarjeta ESP32 que inicia una transacción al servidor web. Es posible que diez, veinte o incluso mil tarjetas ESP32 guarden sus datos en un único servidor web.

Transmisión de datos a un servidor WEB

Registro de datos en el lado del servidor web.

En el capítulo anterior *Comprensión de la transmisión por GET a un servidor WEB* , explicamos cómo ESP32Forth transmite información a un servidor web.

Ahora veamos cómo, del lado del servidor, guardaremos los datos. Aquí hay un primer script, en PHP, que realiza esta grabación:

```
<?php
// echo "<pre>"; var_dump($_GET);
$handle = fopen("datasRecords.csv","a");
$myDatas = array(
    'currentDateTime' => date("Y-m-d H:i:s"),
    'currentLogin'    => $_GET['log'],
    'currentTemp'     => $_GET['temp'],
);
fwrite($handle, implode(';', $myDatas)."
");
fclose($handle);
echo "DATAs recorded";
```

Este script es muy simple:

- Abrimos un archivo **dataRecords.csv** con `fopen` .
- preparamos los datos para guardar en una tabla `myDatas`
- guardamos estos datos con `fwrite`
- los datos se ponen en formato csv usando `implode`
- cerramos el archivo con `fclose`

archivo en formato **csv** es fácil de recuperar con una hoja de cálculo o leer con un simple editor de texto.

Protección de acceso

Si has seguido atentamente nuestras explicaciones, te habrás dado cuenta de que transmitimos dos parámetros `log` y `pwd` . Estos dos parámetros sirven primero como claves de acceso a nuestro script de registro de datos.

Es esta protección la que implementamos para evitar cualquier acceso al script por parte de un transmisor no autorizado. Aquí aceptamos dos transmisores:

```
<?php
// echo "<pre>"; var_dump($_GET);
$myAuths = array(
    'pooltemp'  => 'pool2022',
    'housetemp' => 'house2022',
);

/**
 * Test authorization access
 * @param array $auths
 * @return boolean
 */
function testAuths($auths){
    if(array_key_exists($_GET['log'], $auths) &&
$auths[$_GET['log']]==$_GET['pwd']) {
        return true;
    }
    return false;
}

// Recording datas in CSV file format
if (testAuths($myAuths)) {
    $handle = fopen("datasRecords.csv","a");
    $myDatas = array(
        'currentDateTime' => date("Y-m-d H:i:s"),
        'currentLogin'    => $_GET['log'],
        'currentTemp'     => $_GET['temp'],
    );
    fwrite($handle, implode(';', $myDatas)."
");
    fclose($handle);
    echo "DATAs recorded";
} else {
    echo "AUTH failed";
}
```

Este script sirve como ejemplo. Es deliberadamente simple. En una aplicación profesional, las claves y contraseñas se guardarían en la base de datos.

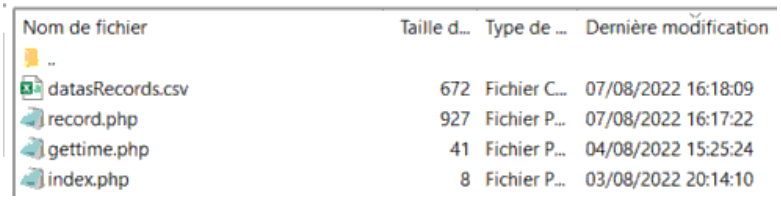
Aquí hay una transacción que se ejecutará exitosamente:

```
http://ws.arduino-forth.com/record.php?log=polltemp&pwd=pool2022&temp=27.5
```

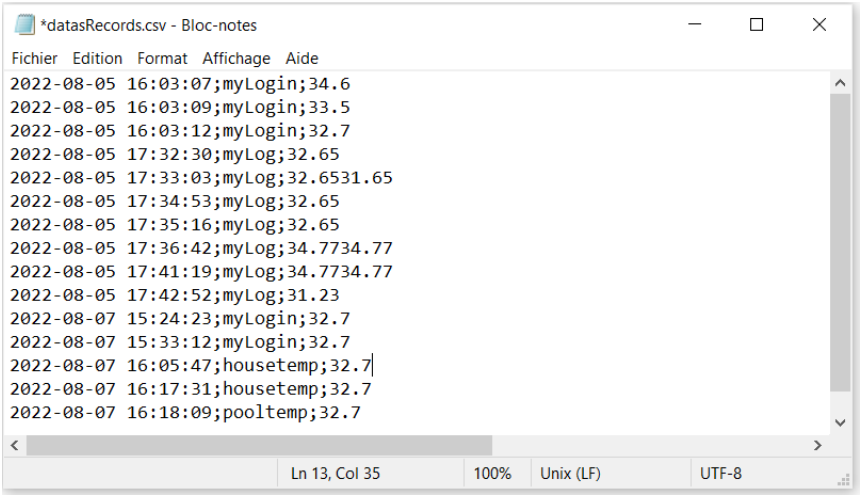
par `log pwd` cuyos valores son probados y aprobados por el script de registro de datos.

Ver datos grabados

Para acceder a los datos registrados utilizamos un cliente FTP (Filezilla):



Allí encontramos nuestro archivo **datasRecords.csv** . Simplemente descárgalo para ver su contenido con cualquier editor de texto:



Encontramos, en las últimas líneas, nuestras pruebas de transmisión con dos inicios de sesión diferentes. El script **record.php** puede procesar transacciones con cientos de tarjetas ESP32 diferentes, cada una con un inicio de sesión diferente.

Agregar datos para transmitir

Si manejas un sensor tipo DHT11 o DHT22 (sensor de temperatura y humedad), estarías tentado a registrar los valores de temperatura y humedad en una sola transacción. Para ello, nada podría ser más sencillo. Este es el aspecto de la transacción que permite esto:

```
http://ws.arduino-forth.com/record.php?log=polltemp&pwd=pool2022&temp=27.5&hygr=62.2
```

Pero para que funcione, debes actuar sobre el script PHP record.php:

```
<?php
// Recording datas in CSV file format
if (testAuths($myAuths)) {
    $handle = fopen("datasRecords.csv","a");
    $myDatas = array(
        'currentDateTime' => date("Y-m-d H:i:s"),
```

```
        'currentLogin'    => $_GET['log'],
        'currentTemp'     => $_GET['temp'],
        'currentHygr"     => $_GET['hygr'],
    );
    fwrite($handle, implode(';', $myDatas)."
");
    fclose($handle);
    echo "DATAs recorded";
} else {
    echo "AUTH failed";
}
```

Aquí, simplemente agregamos una fila a la tabla **`$myDatas`** .

En el FORTH lado, mejoraremos la gestión de URL:

```
256 string myUrl    \ declare string variable

: addTemp ( strAddrLen -- )
    s" &temp=" myUrl append$
    myUrl append$
  ;

: addHygr ( strAddrLen -- )
    s" &hygr=" myUrl append$
    myUrl append$
  ;

: sendData ( strHygr strTemp -- )
    s" http://ws.arduino-forth.com/record.php?
log=myLog&pwd=myPassWd" myUrl $!
    addTemp
    addHygr
    cr myUrl type
    myUrl s>z HTTP.begin
    if
        HTTP.doGet dup 200 =
        if  drop
            httpBuffer bufferSize HTTP.getPayload
            httpBuffer z>s type
        else
            cr ." CNX ERR: " .
        then
    then
    HTTP.end
  ;
```

```
\ for test:
myWiFiConnect
s" 64.2"   \ hygrometry
s" 31.23"  \ temperature
sendData
```

Agregamos dos palabras, **addTemp** y **addHygr**. Cada una de estas palabras concatena un parámetro y su valor a la URL que se utilizará para la transacción web entre su tarjeta ESP32 y el servidor web.

Solo hay dos limitaciones en cuanto a la cantidad de parámetros pasados por el método GET:

- la longitud de nuestra URL como se define en ADELANTE, aquí 256 caracteres. Si desea aumentar este límite, simplemente configure nuestra URL con una longitud inicial más larga: **512 string myUrl**
- la longitud máxima de las URL aceptadas por el protocolo HTTP. Esta longitud puede alcanzar los 8.000 caracteres según los estándares recientes.

Respecto a FORTH, tenemos otras limitaciones. En particular, si deseamos transmitir datos textuales. ciertos caracteres, por ejemplo "&", deberán codificarse. Tendrás que manejar esta codificación en ADELANTE.

Conclusión

PREGUNTA: ¿Para qué se puede utilizar todo esto?

Una tarjeta ESP32 cuesta menos de 10 €/$ cada una. Incluso más como 5€/$ si compras en cantidad. Si integra un sensor de temperatura y un relé, puede, por ejemplo, tomar lecturas de temperatura y transmitir comandos desde el servidor para activar/desactivar un relé. Gestionar la temperatura de varias habitaciones se vuelve muy fácil. Lo mismo ocurre con la gestión del riego inteligente en un invernadero.

También puede monitorear el acceso y activar luces o alarmas muy fácilmente. Tomemos el caso de un portal. Autorizas pasos entre determinadas horas y bloqueas esta misma cancela (ventosa magnética) desde la tarjeta ESP32.

Confiamos en su imaginación para encontrar soluciones prácticas que aprovechen esta transmisión de datos entre tarjetas ESP32 y un servidor web.

¿Y POR QUÉ UN SERVIDOR WEB?

Con un servidor web es fácil consultarlo desde cualquier lugar, con un navegador web instalado en tu PC, una tableta digital, un smartphone. Y un único servidor web puede integrar un número indefinido de scripts diferentes.

Síntesis de sonido con ESP32Forth

Para tus primeros experimentos de sonido, necesitas un altavoz que conectes a una salida GPIO. Pero como la impedancia de los altavoces es muy baja, será necesario pasar por un transistor. Aquí está el diagrama recomendado para un altavoz pequeño.

En este diagrama, se menciona el pin GPIO4. De hecho, este conjunto se puede utilizar en cualquier salida GPIO de la tarjeta ESP32. Las dos salidas que nos interesarán especialmente son GPIO25 y GPIO26 que están reservadas para salidas DAC (Conversión Digital a Analógica).

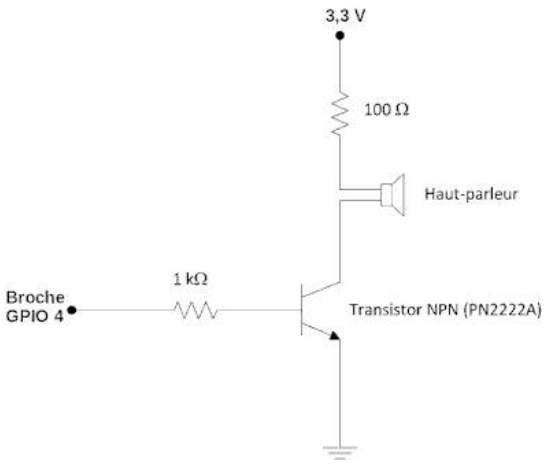


Figure 28: branchement d'un haut parleur

Síntesis de sonido sencilla

Usaremos generación de señal PWM, pero en las salidas DAC.

Nuestro altavoz está conectado a la salida GPIO25, a través del transistor PN2222A que sirve como adaptador de impedancia.

```
 0 constant CHANNEL0     \ define PWM channel 0
25 constant BUZZER       \ buzzer connected to GPI25

ledc                     \ select ledc vocabulary
: initTones ( -- )
    BUZZER CHANNEL0 ledcAttachPin
  ;
```

La palabra **`initTones`** conecta la salida GPIO25 al canal 0 de PWM. La generación de un sonido se realiza así:

```
CHANNEL0 440000 ledcWriteTone drop
```

donde **`freq`** es la frecuencia deseada, multiplicada por 1000. Así, para generar la nota LA (A en notación inglesa), cuya frecuencia es 440 Hz, necesitarás utilizar el valor 440*1000:

```
CHANNEL0 440000 ledcWriteTone drop
```

Definición de tabla de frecuencias del sonido.

Para encontrar las frecuencias sonoras de las notas musicales, acudimos a Wikipedia. Construimos una tabla de frecuencias, donde cada frecuencia se registrará en su forma utilizable por **`ledcWriteTone`**:

```
\ frequency notes
\ source: https://fr.wikipedia.org/wiki/Note_de_musique
\ frequency is multiplied by 1000
create NOTES
\ octave -1
 15350 ,    17330 ,   18360 ,   19450 ,   20600 ,   21830 ,
 23130 ,    24500 ,   25960 ,   27500 ,   29140 ,   30870 ,
\ octave 0
 32700 ,    34650 ,   36710 ,   38890 ,   41200 ,   43650 ,
 46250 ,   49000 ,    51910 ,   55000 ,   58270 ,   61740 ,
\ octave 1
 65410 ,    69300 ,   73420 ,   77780 ,   82410 ,   87310 ,
 92500 ,    98000 ,  103830 ,  110000 ,  116540 ,  123470 ,
\ octave 2
 130810 ,  138590 ,  146830 ,  155560 ,  164810 ,  174610 ,
 185000 ,  196000 ,  207650 ,  220000 ,  233080 ,  246940 ,
\ octave 3
 261630 ,  277180 ,  293660 ,  311130 ,  329630 ,  349230 ,
 369990 ,  392000 ,  415300 ,  440000 ,  466160 ,  493880 ,
\ octave 4
 523250 ,  554370 ,  587330 ,  622250 ,  659260 ,  698460 ,
 739990 ,  783990 ,  830610 ,  880000 ,  932330 ,  987770 ,
\ octave 5
1046500 , 1108730 , 1174660 , 1244510 , 1318510 , 1396910 ,
1479980 , 1567980 , 1661220 , 1760000 , 1864660 , 1975530 ,
\ octave 6
2093000 , 2217460 , 2349320 , 2489020 , 2637020 , 2793830 ,
2959960 , 3135960 , 3322440 , 3520000 , 3729310 , 3951070 ,
\ octave 7
4186010 , 4434920 , 4698640 , 4978030 , 5274040 , 5587650 ,
5919910 , 6271930 , 6644880 , 7040000 , 7458620 , 7902130 ,
\ octave 8
 8372020 ,  8869840 ,  9397280 ,  9956060 , 10548080 , 11175300 ,
11839820 , 12543860 , 13289760 , 14080000 , 14917240 , 15804260 ,
```

Hay doce notas por octava, de ahí la definición de 12 valores por octava. Aquí grabamos sólo 10 líneas, o 10 octavas. Porque después de 15 Khz los sonidos ya no serían audibles.

Para encontrar una nota, sólo necesitas saber su posición en una octava. Por ejemplo, nuestra nota La en la octava 3 será: ((octava+1)*12)+posición. A estando en la posición 10 de la octava 3, la dirección a determinar será NOTES+4*((OCTAVE+1*12)+position)

Recuperar la frecuencia de una nota musical

Primero creamos un conjunto de palabras.octava `que` nos permitirá seleccionar la octava deseada. Luego, definimos `get.note` que recupera la frecuencia de la nota deseada:

```
3 value OCTAVE
\ select octave in interval -1..8
: set.octave ( n[-1..8] )
    to OCTAVE
  ;

\ select note in interval 1..12
: get.note ( n[1..12] -- )
    1- OCTAVE 1+ 12 * +  cell *     \ calc. offset in NOTES array
    NOTES + @                       \ fetch frequency of selected
note
  ;

3 value OCTAVE
\ select octave in interval -1..8
: set.octave ( n[-1..8] )
    to OCTAVE
  ;

: OCT6 ( -- )    6 set.octave ;
: OCT5 ( -- )    5 set.octave ;
: OCT4 ( -- )    4 set.octave ;
: OCT3 ( -- )    3 set.octave ;
: OCT2 ( -- )    2 set.octave ;
: OCT1 ( -- )    1 set.octave ;
```

Más adelante veremos cómo gestionar las notas llamándolas desde su notación.

Administrar la duración de la nota

La duración de una nota es el intervalo de tiempo entre la activación de dos notas consecutivas.

Un retraso base está definido por la constante **`WHOLE-NOTE-DURATION`**.

Las duraciones se definen en un nuevo vocabulario **`music`**:

```
1600 constant WHOLE-NOTE-DURATION
WHOLE-NOTE-DURATION value duration

vocabulary music
music definitions
music also

\ set duration of a whole note
: o  ( -- )
    WHOLE-NOTE-DURATION to duration
  ;
```

```
\ set duration of a white note
: o|  ( -- )
    WHOLE-NOTE-DURATION 2/ to duration
  ;

\ set duration of a black note
: .|  ( -- )
    WHOLE-NOTE-DURATION 2/ 2/ to duration
  ;

\ set duration of a half black note
: .|'  ( -- )
    WHOLE-NOTE-DURATION 2/ 2/ 2/ to duration
  ;

\ set duration of a quarter black note
: .|"  ( -- )
    WHOLE-NOTE-DURATION 2/ 2/ 2/ 2/ to duration
  ;
```

Definimos palabras que simbolizan las duraciones deseadas: `o` para una nota completa, `\ o` para una blanca, `\.` para una nota negra, etc...

Soporte de una nota

El sostenido de una nota es la cantidad de tiempo que la nota es audible durante su tiempo de ejecución. Definimos un valor `de sustain` que expresa el porcentaje de emisión sustentada de la nota durante su duración total. Si este valor es 100, las notas se suceden sin ningún silencio entre las notas.

```
\ sustain of note, in interval [0..100]
90 value SUSTAIN

ledc
\ sustain note in interval [0..100]
: sustain.note ( -- )
    duration SUSTAIN 100 */ ms
    CHANNEL0 0 ledcWriteTone drop
    duration 100 SUSTAIN - 100 */ ms
  ;
```

La palabra `sustain.note` genera dos retrasos. El primer retraso corresponde a la duración del mantenimiento de la nota. El segundo retraso corresponde a un retraso de mantenimiento del silencio. La suma de estos dos retrasos siempre corresponde al retraso definido en duración.

Creando notas musicales

Llegamos a la parte más interesante, definiendo las notas por su nombre:

```
: create-note
    \ compile position in octave
    create      ( position -- )
        ,
    \ get note frequency in current octave
    does>
        @ 1- get.note
        CHANNEL0 swap ledcWriteTone drop
        sustain.note
  ;

\ notes in english notation
 1 create-note C
 2 create-note C#
 3 create-note D
 4 create-note D#
 5 create-note E
 6 create-note F
 7 create-note F#
 8 create-note G
 9 create-note G#
10 create-note A
11 create-note A#
12 create-note B

\ notes in french notation
 1 create-note DO
 2 create-note DO#
 3 create-note RE
 4 create-note RE#
 5 create-note MI
 6 create-note FA
 7 create-note FA#
 8 create-note SOL
 9 create-note SOL#
10 create-note LA
11 create-note LA#
12 create-note SI

: SIL ( -- )
    CHANNEL0 0 ledcWriteTone drop
    duration ms
```

```
  ;

forth definitions
```

Además de las doce notas, del **DO** al **SI** , definimos nuestro **SIL** que es un silencio.

Prueba de puntuación

Probamos todas las notas, escala a escala:

```
forth definitions
: music-scale ( -- )
    C C# D D# E F F# G G# A A# B
  ;

initTones
forth also music also
.|
80 to SUSTAIN
OCT1 music-scale
OCT2 music-scale
OCT3 music-scale
OCT4 music-scale
OCT5 music-scale
OCT6 music-scale
```

Si todo va bien, debemos desplegar todas las notas musicales, por semitonos, desde la octava 1 hasta la octava más alta, aquí la 6. No definimos una octava adicional. Factible. Pero los sonidos emitidos entran en una zona límite para ser audibles.

El vuelo del abejorro

Esta es una primera prueba de transposición de una partitura musical. Para ello, recopilamos una pieza musical especialmente difícil, EL **VUELO DEL BOURDON** de **Rimsky KORSAKOV** . Aquí está el primer compás de la primera línea:

Figure 29: première mesure - Le Vol du Bourdon - Rimski KORSAKOV

Así es como codificamos este primer compás, en notación francesa:

```
OCT5 MI RE# RE DO#     RE DO# DO  OCT4 SI
```

O en notación inglesa:

```
OCT5 E D# D C#     D C# C OCT4 B
```

Aquí está el código para la primera línea de esta partición:

```
: 1stLine ( -- )
    .|"  ( duration of a quarter black note )
    OCT5 MI RE# RE DO#     RE DO# DO  OCT4 SI
    OCT5 DO  OCT4 SI LA# LA     SOL# SOL FA# FA
    MI RE# RE DO#    RE DO# DO OCT3 SI
  ;
```

Mis disculpas si cometí algún error al traducir la partitura. A estas alturas es fácil probar esta línea musical:

```
: 2ndLine ( -- )
    .|"  ( duration of a quarter black note )
    OCT4 DO OCT3 SI FA# FA    SOL# SOL FA# FA
    MI RE# RE DO#    RE DO DO# OCT2 SI OCT3
    MI RE# RE DO#    RE DO DO OCT2 SI OCT3
    MI RE# RE DO#    DO FA FA RE#
  ;

: 3rdLine ( -- )
    .|"  ( duration of a quarter black note )
    MI RE# RE DO#   DO DO# RE RE#
    MI RE# RE DO#    DO FA FA RE#
    MI RE# RE DO#    DO DO# RE RE#
    MI RE# RE DO#    RE DO DO# OCT2 SI OCT3
  ;

: flightBumbleBee ( -- )
    initTones
    1stLine
    2ndLine
    3rdLine
  ;
flightBumbleBee
```

Te dejamos codificar las otras tres líneas de la partitura.

Programa en ensamblador XTENSA

Preámbulo

Para aquellos que no están familiarizados con el lenguaje ensamblador, es la capa de nivel más bajo en programación. En ensamblador nos dirigimos directamente al procesador.

También es un idioma difícil, poco legible. Pero por otro lado, el rendimiento es excepcional.

Programamos en ensamblador:

- cuando no existe otra solución para acceder a determinadas funcionalidades de un procesador;
- para hacer que ciertas partes del programa sean más rápidas. ¡El código generado por un ensamblador es el más rápido!
- para divertirse. La programación en ensamblador es un desafío intelectual;
- porque ningún lenguaje evolucionado puede hacerlo todo. A veces, puedes programar funciones en ensamblador que son demasiado complejas para escribirlas en otro idioma.

A modo de ejemplo, aquí tenéis el código de decodificación de Huffman realizado en el ensamblador XTENSA:

```
/* input in t0, value out in t1, length out in t2 */
    srl t1, t0, 6
    li t3, 3
    beq t3, t4, 2f
    li t2, 2
    andi t3, t0, 0x20
    beq t3, r0, 1f
    li t2, 3
    andi t3, t0, 0x10
    beq t3, r0, 1f
    li t2, 4
    andi t3, t0, 0x08
    beq t3, r0, 1f
    li t2, 5
    andi t3, t0, 0x04
    beq t3, r0, 1f
    li t2, 6
```

```
    andi t3, t0, 0x02
    beq t3, r0, 1f
    li t2, 7
    andi t3, t0, 0x01
    beq t3, r0, 1f
    li t2, 8
    b 2f
    li t1, 9
1: /* length = value */
    move t1, t2
2: /* done *
```

Desde la versión 7.0.7.4, ESP32Forth incluye un ensamblador XTENSA completo. Este ensamblador utiliza notación infija:

```
\ en ensamblador convencional:
\   andi t3, t0, 0x01

\ en ensamblador XTENSA con ESP32 en adelante:
    a3 a0 $01 ANDI,
```

ESP32forth es el **primer lenguaje de programación de alto nivel** para ESP32 que integra un ensamblador XTENSA.

Esta característica permite al programador definir sus macros de ensamblaje.

Cualquier palabra escrita en lenguaje ensamblador XTENSA desde ESP32forth se puede utilizar inmediatamente en cualquier definición en lenguaje FORTH.

Compile el ensamblador XTENSA

Desde la versión 7.0.7.15, ESP32forth ofrece el ensamblador XTENSA como opción. Para compilar esta opción:

- carpeta **optional** en la carpeta donde descomprimió el archivo ZIP de la cuarta versión de ESP32
- archivo **assemblers.h** a la carpeta raíz que contiene el archivo **ESP32forth.ino**
- ejecute ARDUINO IDE, compile **ESP32forth.ino** y cárguelo en la placa ESP32

Si todo ha ido bien accedes al ensamblador de XTENSA tecleando una vez:

```
xtensa-assembler
```

Para comprobar la correcta disponibilidad del conjunto de instrucciones XTENSA:

```
assembler xtensa vlist
```

Programación en ensamblador

Para entender claramente lo dicho anteriormente, aquí hay una definición propuesta a modo de ejemplo por Brad NELSON:

```
\ ejemplo propuesto por Brad NELSON
code my2*
    a1 32 ENTRY,
    a8 a2 0 L32I.N,
    a8 a8 1 SLLI,
    a8 a2 0 S32I.N,
    RETW.N,
end-code
```

Acabamos de definir la palabra **`my2*`** que tiene exactamente la misma acción que la palabra **`2*`**. El montaje del código es inmediato. Por lo tanto, podemos probar nuestra definición de **`my2*`** desde la terminal:

```
--> 3 my2*
 ok
6 --> 21 my2*
 ok
6 42 -->
```

Esta posibilidad de probar inmediatamente un código ensamblado permite probarlo in situ. Si tenemos que escribir código algo complejo será fácil cortarlo en fragmentos y probar cada parte de este código desde el intérprete ESP32forth.

El código ensamblador de XTENSA se coloca después de la palabra a definir. Es la secuencia de código **`my2*`** la que crea la palabra **`my2*`** .

Las siguientes líneas contienen el código ensamblador de XTENSA. La definición del ensamblado finaliza con la ejecución del **`end-code`**.

Resumen de instrucciones básicas.

Lista de instrucciones básicas incluidas en todas las versiones de la arquitectura Xtensa. El resto de esta sección proporciona una descripción general de las instrucciones básicas.

Carga / carga

```
    L8UI, L16SI, L16UI, L32I, L32R,
```

Tienda/almacenamiento

```
    S8I, S16I, S32I,
```

Orden de memoria

```
   MEMW, EXTW,
```

Saltos

```
    CALL0, CALLX0, RET, J, JX,
```

ramificación condicional

```
    BALL, BNALL, BANY, BNONE, BBC, BBCI, BBS, BBSI, BEQ, BEQI, BEQZ,
BNE,
    BNEI, BNEZ,BGE, BGEI, BGEU, BGEUI, BGEZ, BLT, BLTI, BLTU, BLTUI,
BLTZ,
```

Cambio

```
    MOVI, MOVEQZ, MOVGEZ,  MOVLTZ, MOVNEZ,
```

Aritmética

```
    ADDMI, ADD, ADDX2, ADDX4, ADDX8, SUB, SUBX2, SUBX4, SUBX8, NEG,
ABS,
```

Lógica binaria

```
    AND, OR, XOR,
```

Desplazamiento

```
    EXTUI, SRLI, SRAI, SLLI, SRC, SLL, SRL, SRA, SSL, SSR, SSAI,
SSA8B, SSA8L,
```

control del procesador

```
    RSR, WSR, XSR, RUR, WUR, ISYNC, RSYNC, ESYNC, DSYNC, NOP,
```

Un desensamblador extra

Un ensamblador es muy bueno. El código fácil de integrar con las definiciones ADELANTE es maravilloso. ¡Pero tener un desensamblador XTENSA es real!

Tomemos la definición de **`my2*`** previamente ensamblada. Es fácil de desmontar:

my2* cell+ @ 20 disasm

```
\ display:
\ 1074338656  --  a1 32 ENTRY,            -- 004136
\ 1074338659  --  a8 a2 0 L32I.N,                -- 0288
\ 1074338661  --  a8 a8 1 SLLI,            -- 1188F0
\ 1074338664  --  a8 a2 0 S32I.N,               -- 0289
\ 1074338666  --  RETW.N,                 -- F01D
\ 1074338668  -- ......
```

al código de nuestra palabra `my2*` mediante dirección indirecta, cuya dirección se coloca en el campo de parámetros.

Cada línea muestra:

- la dirección del código ensamblado
- el código desensamblado en esta dirección en 2 o 3 bytes
- el código hexadecimal correspondiente al código desensamblado

El desensamblador también puede actuar sobre todo el código ya compilado o ensamblado. Veamos el código de la palabra `2*` :

```
' 2* @ 20 disasm
\ display:
\ 1074606252  --  a12 a3 0 L32I.N,                -- 03C8
\ 1074606254  --  a5 a5 1 SLLI,            -- 1155F0
\ 1074606257  --  a15 a12 0 L32I.N,               -- 0CF8
\ 1074606259  --  a3 a3 4 ADDI.N,                -- 334B
\ 1074606261  --  1074597318 J,            -- F74346
```

El desmontaje indica que el código conduce a un salto incondicional `1074597318 J` ,. Es fácil continuar con el desmontaje hasta esta nueva dirección:

```
1074597318 20 disasm
\ display:
\ 1074597318  --  a15 JX,                   -- 000FA0
\ 1074597321  --  a10 64672 L32R,                -- FCA0A1
\ 1074597324  --  a5 a7 1 S32I,            -- 016752
\ 1074597327  --  1074633168 CALL8,               -- 08C025
\ 1074597330  --  a12 a3 0 L32I,           -- 0023C2
\ 1074597333  --  a2 a7 4 ADDI,            -- 04C722
\ 1074597336  ......
```

Primeros pasos en el ensamblador XTENSA

Preámbulo

El código ensamblador no es portátil en otro entorno, o a costa de enormes esfuerzos para comprender y adaptar el código ensamblado.

Una FORTH versión no está completa si no tiene ensamblador.

No se requiere programación en ensamblador. Pero en algunos casos, crear una definición en ensamblador puede ser mucho más fácil que una versión en lenguaje C o en lenguaje FORTH puro.

Pero, sobre todo, una definición escrita en ensamblador tendrá una velocidad de ejecución inigualable.

Veremos, mediante ejemplos muy sencillos y muy breves, cómo dominar la programación de las definiciones FORTH escritas en ensamblador Xtensa.

Invocando al ensamblador Xtensa

Al iniciar ESP32 en adelante, es imposible definir palabras en el ensamblador Xtensa sin invocar la palabra **`xtensa-assembler`**. Esta palabra cargará el contenido del vocabulario **`xtensa`** . Esta palabra solo debe invocarse una vez al iniciar ESP32 en adelante y antes de cualquier definición de una palabra en código xtensa:

```
forth
DEFINED? code invert [IF] xtensa-assembler [THEN]
```

Ahora, si escribimos **`order`**, ESP32 muestra:

```
    xtensa >> asm >> FORTH
```

Es este orden de vocabularios el que se debe respetar cuando se desea definir una nueva palabra en el ensamblado de Xtensa utilizando las palabras de definición **`code`** y **`end-code`** .

Xtensa y la pila FORTH

El procesador Xtensa tiene 16 registros, del a0 al a15. En realidad hay 64 registros, pero sólo podemos acceder a una ventana de 16 registros entre estos 64 registros, accesible en el intervalo 00..15.

El registro a2 contiene el puntero de pila ADELANTE.

Cada vez que se apila un valor, el puntero de la pila se incrementa en cuatro unidades:

```
SP@ .   \ affiche 1073632236
1
SP@ .   \ display 1073632240
2
SP@ .   \ display 1073632244
drop drop
SP@ .   \ 1073632236
```

Así es como podríamos reescribir esta palabra **SP@** en el ensamblador Xtensa:

```
\ obtener puntero de pila SP - equivalente a SP@
code mySP@
    a1 32       ENTRY,
    a8 a2       MOV.N,  \ copie contenu de a2 dans a8
    a2 a2 4     ADDI,   \ incremente a2
    a8 a2 0     S32I.N, \ copie a8 dans adresse pointée par a2+0
                RETW.N,
end-code
```

Probemos esta nueva palabra **mySP@** :

```
mySP@ .
\ muestra 1073632240
SP@ .
\ muestra 1073632240
```

Escribir una macro instrucción Xtensa

En nuestra definición de la palabra **mySP@** , la secuencia **a2 a2 4 ADDI** incrementa el puntero de la pila en cuatro unidades. Sin este incremento, es imposible devolver un valor a la parte superior de la FORTH pila. Con FORTH, escribiremos una macro que automatice esta operación.

Para empezar, ampliaremos el vocabulario **asm:**

```
asm definitions

: macro:
    :
  ;
```

Nuestra definición de **macro:** es redundante con **:** pero tiene la ventaja de hacer que el código FORTH sea un poco más legible cuando definimos una macroinstrucción que ampliará el vocabulario **xtensa** :

```
xtensa definitions

macro: sp++,
    a2 a2 4    ADDI,
  ;
```

macro instrucción `sp++` , podemos reescribir la definición de `mySP@`:

```
forth definitions
asm xtensa

   \ get Stack Pointer SP - equivalent for SP@
code mySP@
    a1 32       ENTRY,
    a8 a2       MOV.N,  \ copy content of a2 in a8
        sp++,
    a8 a2 0     S32I.N, \ copy a8 in address pointed by a2+0
                RETW.N,
end-code
```

Es perfectamente posible integrar una macro en otra. En el código `mySP@`, la línea de código `a8 a2 0 S32I.N` copia el contenido del registro a8 en la dirección señalada por a2. Aquí está esta nueva macro instrucción:

```
xtensa definitions

\ increment Stack Pointer and store content of ar in addr
\ pointed by Stack Pointer
macro: arPUSH, { ar -- }
    sp++,
    ar a2 0 S32I.N,
  ;
```

Esta macro instrucción utiliza una variable local `ar` . Podríamos haber prescindido de ella, pero la ventaja de esta variable es que el código de la macro es más legible.

Aquí está el código `mySP@` con esta macroinstrucción.

```
forth definitions
asm xtensa

\ get Stack Pointer SP - equivalent to SP@
code mySP@3
    a1 32       ENTRY,
    a8 a2       MOV.N,
    a8  arPUSH,
                RETW.N,
```

```
end-code
```

Completemos nuestra lista de instrucciones macro:

```
xtensa definitions

\ décrémente pointeur de pile
macro: sp--,    ( -- )
    a2 a2 -4    ADDI,
  ;

\ Store content of addr pointed by Stack Pointer in ar
\ and decrement Stack Pointer
macro: arPOP,   { ar -- }
    ar a2 0     L32I.N,
    sp--,
  ;
```

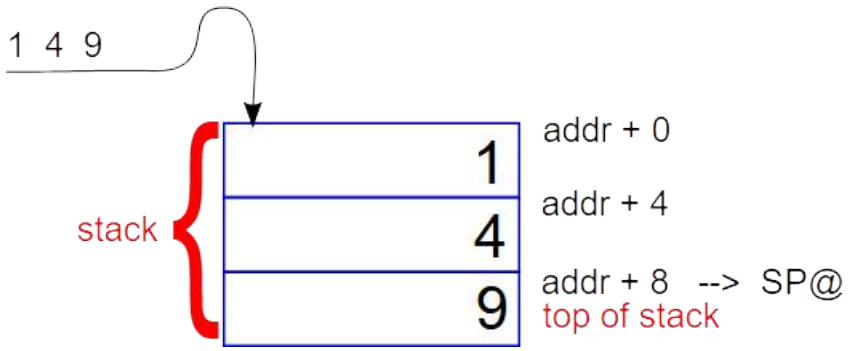
Con estas nuevas macros, reescribamos `swap` :

```
forth definitions
asm xtensa

code mySWAP
    a1 32       ENTRY,
    a9  arPOP,
    a8  arPOP,
    a9  arPUSH,
    a8  arPUSH,
                RETW.N,
end-code

17 24 mySWAP
```

Administrar la pila FORTH en el ensamblador Xtensa

Se puede acceder a la posición del puntero de pila FORTH mediante `SP@` . Apilar un entero de 32 bits (tamaño predeterminado para ESP32 en adelante) incrementa este puntero de pila en cuatro unidades.

Discutimos cómo administrar el incremento o decremento de este puntero de pila a través de las macroinstrucciones `sp++` y `sp-- .` Estas macroinstrucciones mueven el puntero de la pila cuatro unidades.



Aquí hemos apilado tres valores, `1, 4` y `9` . Cada vez que apila, el puntero de la pila se incrementa automáticamente. En el ensamblador Xtensa, el puntero de la pila se encuentra en el registro a2. Hemos visto que podemos manipular el contenido de este registro con las macroinstrucciones `sp++` y `sp--` ,. La manipulación de este registro tiene una acción directa sobre el puntero de la pila gestionado por ESP32forth.

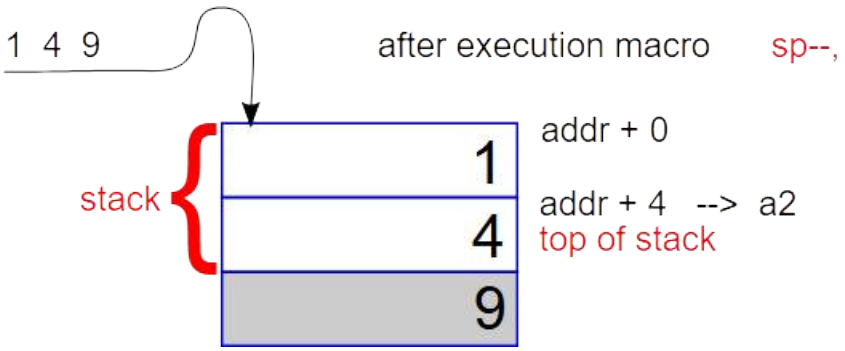
Así es como reescribimos la palabra `+` en ensamblador manipulando el puntero de la pila a través de nuestras macroinstrucciones `arPOP` y `arPUSH` :

```
code my+
    a1 32       ENTRY,
    a7  arPOP,
    a8  arPOP,
    a7 a8 a9    ADD,
    a9  arPUSH,
            RETW.N,
end-code
```

Hay otra forma de recuperar datos de la pila usando la instrucción `L32I.N.` Esta instrucción utiliza un índice inmediato:

```
code my+
    a1 32 ENTRY,
        sp--,
    a7 a2 0     L32I.N,
    a8 a2 1     L32I.N,
    a7 a8 a9    ADD,
    a9 a2 0     S32I.N,
    RETW.N,
end-code
```

Antes de recuperar los datos de la pila, disminuimos el puntero de la pila con nuestra macro instrucción `sp--` ,. De esta forma, el puntero retrocede 4 unidades.



Pero el hecho de que el puntero retroceda no significa que los datos previamente apilados desaparezcan. Veamos esta línea de código en detalle:

```
    a7 a2 0     L32I.N,
```

Esta instrucción carga el registro a7 con el contenido de la dirección señalada por (a2)+n*4. Aquí, n es 0. Esta instrucción pondrá el valor 4 en nuestro registro a7.

Veamos la siguiente línea:

```
    a8 a2 1     L32I.N,
```

El registro a8 se carga con el contenido señalado por (a2)+1*4. Esta instrucción pone el valor 9 en nuestro registro a8.

```
    a9 a2 0     S32I.N,
```

Aquí, el contenido del registro a9 se almacena en la dirección señalada por (a2)+1*0. De hecho, sobrescribimos el valor 4 con el resultado de sumar el contenido de los registros a7 y a8.

Veamos un último ejemplo en el que procesamos dos parámetros y enviamos dos de ellos a la pila de datos. En este ejemplo, reescribimos la palabra **/MOD** :

```
code my/MOD ( n1 n2 -- rem quot )
    a1 32       ENTRY,
    a7  arPOP,          \ diviseur dans a7
    a8  arPOP,          \ valeur à diviser dans a8
    a7 a8 a9    REMS,   \ a9 = a8 MOD a7
    a9  arPUSH,
    a7 a8 a9    QUOS,   \ a9 = a8 / a7
    a9  arPUSH,
            RETW.N,
end-code

5 2 my/MOD . .      \ display 2 1
```

```
-5 -2 my/MOD . .    \ display 2 -1
```

En la palabra **my/MOD** utilizamos los mismos datos n1 y n2 colocados respectivamente en los registros a8 y a7. Son entonces las instrucciones **REMS** y **QUOT** las que permiten calcular los resultados devueltos por **my/MOD** .

Eficiencia de palabras escritas en ensamblador XTENSA

En nuestro último ejemplo anterior, reescribimos la palabra **/MOD** . La pregunta que debemos hacernos es: "¿la palabra **my/MOD** **es** realmente más rápida de ejecutar que la palabra **/MOD** ?".

Para ello utilizaremos la palabra **medida:** cuyo código FORTH se explica en el capítulo *Medición del tiempo de ejecución de una FORTH palabra* .

```
: test1
    1000000 for
        5 2 /MOD
        drop drop
    next
  ;

: test2
    1000000 for
        5 2 my/MOD
        drop drop
    next
  ;

measure: test1  \ display: execution time: 0.856sec.
measure: test2  \ display: execution time: 0.600sec.
```

Las palabras **test1** y **test2** son similares, excepto que **test2** ejecuta **my/MOD** . En 1 millón de iteraciones, el ahorro de tiempo asciende a 0,144 segundos. No es mucho, pero la proporción todavía parece significativa.
Por el contrario, vemos que el lenguaje FORTH es muy rápido en tiempo de ejecución.

Lazos y conexiones en ensamblador XTENSA

La instrucción LOOP en el ensamblador XTENSA

El bucle LOOP en el ensamblador XTENSA funciona utilizando la instrucción **LOOP** para indicarle al procesador que repita un bloque de instrucciones hasta que un contador específico llegue a cero. El bucle se inicializa estableciendo el valor inicial del contador y luego ejecutando la instrucción **LOOP** con ese valor como argumento. En cada iteración del ciclo, el contador disminuye en 1 hasta llegar a cero, momento en el cual el ciclo se detiene. En ensamblador clásico:

```
    ; Initialization of the counter to 10
    MOVI a0, 10

    ; Beginning of the LOOP loop
loop:
    ; Instruction(s) to repeat
    ...
    ; Decrement the counter and test the stop condition
    LOOP a0, loop
```

Aquí, el bucle LOOP repite las instrucciones entre `loop:` y `LOOP a0,` realizando un bucle 10 veces, disminuyendo el contador a0 con cada iteración. Cuando el contador llega a cero, el ciclo se detiene.

Cuando el procesador XTENSA encuentra la instrucción **LOOP** , inicializa tres registros especiales:

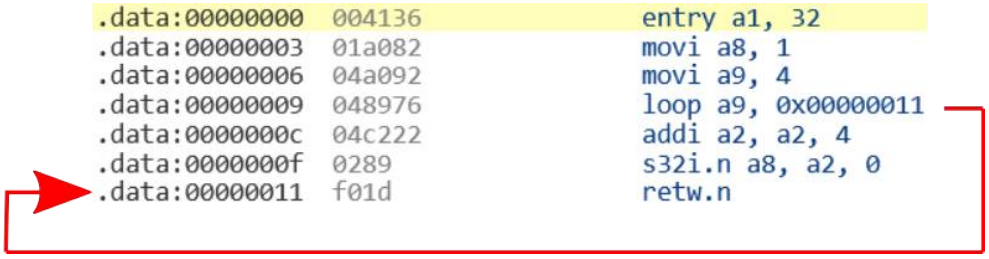
- **LCOUNT ← AR[s] – 1**
  El registro especial LCOUNT se inicializa con el contenido del registro como, aquí a0 en nuestro ejemplo, disminuido en una unidad. Cuando el contador alcanza el valor 0, la instrucción LOOP completa el ciclo;
- **LBEG ← PC + 3**
  El registro especial LBEG contiene la dirección inicial del bucle LOOP que se está ejecutando actualmente. Esta dirección está definida por la instrucción LOOP.
- **LEND ← PC + (024||imm8) + 4** El registro especial LEND contiene la dirección final del bucle LOOP que se está ejecutando actualmente. Esta dirección está definida por la instrucción LOOP.

En ensamblador XTENSA la instrucción LOOP admite dos parámetros:

```
  LOOP as, label
```

`Label` corresponde a un desplazamiento de 8 bits después de la instrucción `LOOP` . No puede repetir un código de más de 256 bytes de longitud.

Aquí hay algo de código XTENSA desensamblado usando un bucle LOOP:



Desmontarlo indica una dirección de sucursal. En realidad, el código ensamblado sólo contiene este desplazamiento indicado por la etiqueta en forma de valor positivo de 8 bits.

Gestionar un bucle en ensamblador XTENSA con ESP32forth

El lenguaje FORTH no puede resolver una referencia directa. A menos que busques a tientas, es difícil utilizar la instrucción `LOOP` sin encontrar un truco.

Definición de macroinstrucciones de gestión de bucles

instrucción `LOOP,` definiremos dos instrucciones macro, respectivamente `For,` y `Next,` de las cuales aquí está el código en lenguaje FORTH:

```
: For, { as n -- }
    as n MOVI,
    as 0 LOOP,
    chere 1- to LOOP_OFFSET
  ;

: Next, ( -- )
    chere LOOP_OFFSET - 2 -
    LOOP_OFFSET [ internals ] ca! [ asm xtensa ]
  ;
```

La instrucción macro `For` acepta los mismos parámetros que la instrucción `LOOP` :

```
  as n For,
```

- **as** igual que el registro que contiene el número de iteraciones del bucle;
- **n** es el número de iteraciones.

Usando las macros For, y Next,

Definimos una palabra **myLOOP** para probar la **instrucción LOOP,** a través de las instrucciones macro **For, Next,**:

```
code myLOOP ( n -- n' )
    a1 32           ENTRY,
    a8 1            MOVI,
    a9 4        For,            \ LOOP start here
        a8 a8 1     ADDI,
        a8      arPUSH,         \ push result on stack
    Next,
                    RETW.N,
end-code
```

El registro a8 se inicializa con el valor 1. El bucle **For, Next,** incrementa el contenido de a8 y apila su contenido. Esto es lo que ofrece ejecutar **MyLOOP** :

```
 ok
--> myLoop
 ok
2 3 4 5 -->
```

ATENCIÓN : si el número de iteraciones es cero, el número de iteraciones aumenta a 232.

Instrucciones de conexión en ensamblador XTENSA

El ensamblador XTENSA en el vocabulario **xtensa** tiene varios tipos de instrucciones de rama:

- conexiones que utilizan indicadores booleanos definidos en el registro especial **BR** : **BF, BT,**
- las conexiones que realizan pruebas en los registros: **BALL, BANY, BBC, BBS, BEQ, BGE, BLT, BNE, BNONE,**

Es esta segunda categoría de conexiones la que nos interesa.

Definición de macros de ramificación

El ensamblador ESP32forth xtensa no dispone de mecanismo de gestión de etiquetas como sí ocurre en un ensamblador clásico. Para ser eficaz, la gestión de etiquetas debe funcionar en varias etapas si es necesario resolver las bifurcaciones directas. Esto es incompatible con el funcionamiento del lenguaje FORTH que compila o ensambla en una sola pasada.

Superamos esta dificultad definiendo dos macroinstrucciones, `If` y `Then,` que gestionarán estas conexiones directas:

```
: If, ( -- BRANCH_OFFSET )
    chere 1-
  ;

: Then, { BRANCH_OFFSET -- }
    chere BRANCH_OFFSET - 2 -
    BRANCH_OFFSET [ internals ] ca! [ asm xtensa ]
  ;
```

La macroinstrucción debe ir precedida de otra macroinstrucción. Para nuestra primera prueba, definimos la macro `<,` que ensamblará una rama sin resolver:

```
: <,  ( as at -- )
    0 BGE,
  ;
```

Usando estas macros en nuestro primer ejemplo:

```
code my< ( n1 n2 -- fl )    \ fl=1 if n1 < n2
    a1 32           ENTRY,
    a8          arPOP,              \ a8 = n2
    a9          arPOP,              \ a9 = n1
    a7 0            MOVI,           \ a7 = 1
    a8 a9 <, If,
        a7 1        MOVI,           \ a7 = 0
    Then,
    a7          arPUSH,
                RETW.N,
end-code
```

Sintaxis de instrucciones macro de ramificación

En nuestro ejemplo, utilizamos la macro instrucción `<,` que está asociada a la instrucción de bifurcación BGE `,` y cuyo significado es: "Bifurcación si es mayor o igual". Normalmente, se traduciría por ">=". ¿Por qué se usó "<"?

Esto se debe a que nuestra macro instrucción `If,.... Then,` tiene una lógica opuesta a la del salto a realizar. El código incluido en `If, .... Then,` se ejecutará si la condición requerida no es válida. Aquí está la tabla que resume esta lógica invertida que explica la elección del nombre de estas macro instrucciones utilizadas antes de `If,.... Then` ,:

XTENSA branch instruction			Macro
BEQ	Branch if Equal	AR[s] = AR[t]	<>,
BGE	Branch if Greater Than or Equal	AR[s] ≥ AR[t]	<,
BLT	Branch if Less Than	AR[s] < AR[t]	>=,
BNE	Branch if Not Equal	AR[s] ≠ AR[t]	=,

ejemplo de ensamblaje `my<` . Esto es lo que da la ejecución de la palabra `my<` :

```
10 20 my< .     \ affiche: 1
20 20 my< .     \ affiche: 0
20 10 my< .     \ affiche: 0
-5 35 my< .     \ affiche: 1
-10 -3 my< .    \ affiche: 1
-3 -10 my< .    \ affiche: 0
```

Vemos que se respeta esta lógica invertida.

Una vez entendida esta lógica, podemos definir una nueva macroinstrucción `>=` ,:

```
: >=,  ( as at -- )
    0 BLT,
  ;
```

Y pruebe esta macroinstrucción:

```
code my>= ( n1 n2 -- fl )    \ fl=1 if n1 < n2
    a1 32           ENTRY,
    a8          arPOP,              \ a8 = n2
    a9          arPOP,              \ a9 = n1
    a7 0            MOVI,           \ a7 = 1
    a8 a9 >=, If,
        a7 1        MOVI,           \ a7 = 0
    Then,
    a7          arPUSH,
                    RETW.N,
end-code

10 20 my>= .        \ display: 0
20 20 my>= .        \ display: 1
20 10 my>= .        \ display: 1
-5 35 my>= .        \ display: 0
-10 -3 my>= .       \ display: 0
-3 -10 my>= .       \ display: 1
```

Definición y manipulación de registros.

En el documento técnico del ESP32 encontramos una cantidad muy grande de registros. Estos registros le permiten controlar todos los periféricos y puertos GPIO en la placa ESP32.

SAR ADC2 control registers			
SENS_SAR_READ_CTRL2_REG	SAR ADC2 data and sampling control	0x3FF48890	R/W
SENS_SAR_MEAS_START2_REG	SAR ADC2 conversion control and status	0x3FF48894	RO
ULP coprocessor configuration register			
SENS_ULP_CP_SLEEP_CYC0_REG	Sleep cycles for ULP coprocessor	0x3FF48818	R/W
Pad attenuation configuration registers			
SENS_SAR_ATTEN1_REG	2-bit attenuation for each pad	0x3FF48834	R/W
SENS_SAR_ATTEN2_REG	2-bit attenuation for each pad	0x3FF48838	R/W
DAC control registers			
SENS_SAR_DAC_CTRL1_REG	DAC control	0x3FF48898	R/W
SENS_SAR_DAC_CTRL2_REG	DAC output control	0x3FF4889C	R/W

Figure 30: extracto del manual de referencia técnica

Por lo general la manipulación de estos registros se realiza mediante la capa de aplicación que ofrece ESP32forth. Por tanto, no es necesario acceder directamente.

En algunos casos puede resultar interesante gestionar los registros directamente:

- para acceder a funciones no ofrecidas por ESP32forth
- para ejecutar el código FORTH más rápido

Definición de registros

Definir un registro es muy sencillo:

```
$3FF48898 constant SENS_SAR_DAC_CTRL1_REG   \ DAC control
```

La primera desventaja de definir un registro como una constante es que al leer el código fuente, no podremos distinguir el registro de otras constantes. Por tanto, definiremos una palabra de creación de registro así:

```
\ definir un registro, similar a una constante
: defREG:
    create ( addr1 --  )
        ,
    does>  ( -- regAddr )
        @
  ;

$3FF48898 defREG: SENS_SAR_DAC_CTRL1_REG   \ DAC control
```

De esta forma, al releer nuestro código, sabemos que la palabra creada es un registro. La otra ventaja es que podemos modificar **`defREG:`** para cambiar su comportamiento: añadiendo pruebas de control, inicializando parámetros, etc...

Acceso a contenidos de registro

Ejemplo, valores de bits en el registro **SENS_SAR_DAC_CTRL1_REG** :

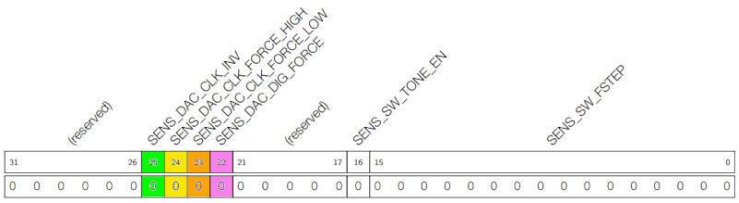


Figure 31: bits en el registro ENS_SAR_DAC_CTRL1_REG

Este registro contiene bits o bloques de bits que tienen funciones definidas.

Primero crearemos una palabra que nos permitirá visualizar el contenido de un registro:

```
\ mostrar contenido de registro
: .reg ( reg -- )
    base @ >r
    binary
    @ <#
    4 for
        aft
            8 for
                aft  #  then
            next
            bl hold
        then
    next
    #>
    cr space ." 33222222 22221111 11111100 00000000"
    cr space ." 10987654 32109876 54321098 76543210"
    cr type
    r> base !
  ;
```

Veamos qué aporta el contenido de nuestro registro **SENS_SAR_DAC_CTRL1_REG**:

```
SENS_SAR_DAC_CTRL1_REG .reg
\ display:
 33222222 22221111 11111100 00000000
 10987654 32109876 54321098 76543210
 00000000 00000000 00000000 00000000 ok
```

Las dos primeras líneas permiten leer verticalmente el rango de un bit de este registro, aquí en rojo, 25, cuyo contenido es 0. Para leer este bit se procede de la siguiente manera:

```
SENS_SAR_DAC_CTRL1_REG @
1 25 lshift and
```

Para modificar este bit y establecerlo en 1:

```
1 25 lshift
SENS_SAR_DAC_CTRL1_REG @
or
SENS_SAR_DAC_CTRL1_REG !
```

Comprobemos con `.reg` :

```
SENS_SAR_DAC_CTRL1_REG .reg
\ display:
 33222222 22221111 11111100 00000000
 10987654 32109876 54321098 76543210
 0000001 00000000 00000000 00000000 ok
```

Si es sólo una vez, ayuda. Veamos cómo hacerlo de manera más eficiente...

Manejo de bits de registro

Reanudemos la modificación del bit 25 de nuestro registro `SENS_SAR_DAC_CTRL1_REG` . A continuación se explica cómo configurar el bit b25 en 1:

```
SENS_SAR_DAC_CTRL1_REG .reg     \ display:
\  33222222 22221111 11111100 00000000
\  10987654 32109876 54321098 76543210
\  00000000 10000000 00000000 00000000 ok

registers
1 25 $02000000 SENS_SAR_DAC_CTRL1_REG m!
SENS_SAR_DAC_CTRL1_REG .reg     \ display:
\  33222222 22221111 11111100 00000000
\  10987654 32109876 54321098 76543210
\  0000001 00000000 00000000 00000000 ok
```

Usamos la `palabra m!` (val shift mask addr --) que acepta cuatro parámetros:

- **val** que es el valor a modificar, aquí 1
- **shift** que corresponde al desplazamiento a aplicar a este valor, aquí 25
- **mask** que corresponde a la máscara lógica de la parte del registro a modificar, aquí $02000000
- **addr** que es la dirección del registro, aquí SENS_SAR_DAC_CTRL1_REG

Definición de máscaras

Se utiliza una máscara para indicar qué bits son modificables. En el ejemplo anterior, modificamos el bit b25. En la documentación de Espressif, el bit b25 está marcado con la etiqueta `SENS_DAC_CLK_INV` . La solución más sencilla sería crear una constante como esta:

```
11 25 lshift constant SENS_DAC_CLK_INV
```

Pero esto no ajusta el valor de compensación, que debe ser el mismo que el valor de la máscara binaria.

Veamos una forma más elegante de definir máscaras:

```
: defMASK:
    create ( mask0 position -- )
        dup ,
        lshift ,
    does> ( -- position mask1 )
        dup @
        swap cell + @
  ;

1 25 defMASK: mSENS_DAC_CLK_INV
```

De paso, tenga en cuenta que el nombre de la máscara tiene el prefijo ' `m` ' (para máscara). Esto no es en absoluto obligatorio. Pero cuando hayas compilado muchos registros y máscaras, el prefijo ' `m` ' te permitirá orientarte entre registros y máscaras:

```
--> words
mSENS_SW_FSTEP mSENS_SW_TONE_EN mSENS_DAC_DIG_FORCE mSENS_DAC_CLK_FORCE_LOW
mSENS_DAC_CLK_FORCE_HIGH mSENS_DAC_CLK_INV defMask: SENS_SAR_DAC_CTRL2_REG
SENS_SAR_DAC_CTRL1_REG GPIO_ENABLE_W1TC_REG GPIO_ENABLE_W1TS_REG GPIO_ENABLE_REG
GPIO_OUT_W1TC_REG GPIO_OUT_W1TS_REG GPIO_OUT_REG DR_REG_GPIO_BASE PIN_DAC2
PIN_DAC1 CONFIG_IDF_TARGET_ESP32S3 CONFIG_IDF_TARGET_ESP32S2 .reg AdcREG:
mtst mset mclr --DAdirect SENS_DAC_CLK_INV defMASK: input$ c+$! mid$ left$
right$ 0$! $! maxlen$ string $= FORTH camera-server camera telnetd bterm
......
```

La palabra definida con **`defMASK:`** coloca el desplazamiento de la máscara en la pila, aquí 25 para **`mSENS_DAC_CLK_INV`** y el valor de la máscara binaria a aplicar.

Reanudemos la modificación del bit b25 con esta definición de máscara:

```
1 mSENS_DAC_CLK_INV SENS_SAR_DAC_CTRL1_REG m!
SENS_SAR_DAC_CTRL1_REG .reg   \ display:
\  33222222 22221111 11111100 00000000
\  10987654 32109876 54321098 76543210
\  0000001 0 00000000 00000000 00000000

0 mSENS_DAC_CLK_INV SENS_SAR_DAC_CTRL1_REG m!
SENS_SAR_DAC_CTRL1_REG .reg   \ display:
\  33222222 22221111 11111100 00000000
\  10987654 32109876 54321098 76543210
\  0000000 0 00000000 00000000 00000000
```

Cambiar del lenguaje C al lenguaje FORTH

Con ESP32Forth, existen dos soluciones para agregar primitivas al diccionario:

- reescriba el código fuente de ESP32Forth agregando las primitivas deseadas;
- reescribe estas palabras en lenguaje C en ADELANTE

La primera solución, en lenguaje C, está escrita aquí:

```
#ifndef ENABLE_DAC_SUPPORT
# define OPTIONAL_DAC_SUPPORT
# else
# include <driver/dac.h>
# include <driver/dac_common.h>
# include <soc/rtc_io_reg.h>
# include <soc/rtc_cntl_reg.h>
# include <soc/sens_reg.h>
# include <soc/rtc.h>
# define OPTIONAL_DAC_SUPPORT \
Y(dac_output_enable, n0 = dac_output_enable( (dac_channel_t) n0 ) ) \
Y(dac_output_disable, n0 = dac_output_disable( (dac_channel_t) n0 ) ) \
Y(dac_output_voltage, n0 = dac_output_voltage((dac_channel_t ) n1, (gpio_num_t ) n0); NIP ) \
Y(dac_cw_generator_enable, PUSH dac_cw_generator_enable () ) \
Y(dac_cw_generator_disable, PUSH dac_cw_generator_disable () ) \
Y(dac_i2s_enable, PUSH dac_i2s_enable() ) \
Y(dac_i2s_disable, PUSH dac_i2s_disable() ) \
Y(rtc_freq_div_set, REG_SET_FIELD(RTC_CNTL_CLK_CONF_REG, RTC_CNTL_CK8M_DIV_SEL, n0 ); DROP ) \
Y(dac_freq_step_set, SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL1_REG, SENS_SW_FSTEP, n0, SENS_SW_FSTEP_S);
DROP ) \
Y(dac1_scale_set, SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_SCALE1, 0, SENS_DAC_SCALE1_S); ) \
Y(dac2_scale_set, SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_SCALE2, 0, SENS_DAC_SCALE2_S); ) \
Y(dac1_offset_set, SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_DC1, n0, SENS_DAC_DC1_S); DROP ) \
Y(dac2_offset_set, SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_DC2, n0, SENS_DAC_DC2_S); DROP ) \
Y(dac1_invert_set, SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_INV1, n0, SENS_DAC_INV1_S); DROP ) \
Y(dac2_invert_set, SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_INV2, n0, SENS_DAC_INV2_S); DROP ) \
Y(dac1_cosine_enable, SET_PERI_REG_MASK(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_CW_EN1_M); ) \
Y(dac2_cosine_enable, SET_PERI_REG_MASK(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_CW_EN2_M); ) \
Y(dacWrite, dacWrite(n1, n0); DROPn(2))
#endif
```

La segunda solución es mirar el código fuente de un archivo escrito en lenguaje C e intentar comprender cómo se manipulan los registros en este lenguaje.

Extracto del archivo **dac-cosine.c** :

```
/*
 * Enable cosine waveform generator on a DAC channel
 */
void dac_cosine_enable(dac_channel_t channel)
{
    // Enable tone generator common to both channels
    SET_PERI_REG_MASK(SENS_SAR_DAC_CTRL1_REG, SENS_SW_TONE_EN);
    switch(channel) {
        case DAC_CHANNEL_1:
            // Enable / connect tone tone generator on / to this channel
            SET_PERI_REG_MASK(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_CW_EN1_M);
            // Invert MSB, otherwise part of waveform will have inverted
            SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_INV1, 2, SENS_DAC_INV1_S);
            break;
        case DAC_CHANNEL_2:
```

```
            SET_PERI_REG_MASK(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_CW_EN2_M);
            SET_PERI_REG_BITS(SENS_SAR_DAC_CTRL2_REG, SENS_DAC_INV2, 2, SENS_DAC_INV2_S);
            break;
        default :
           printf("Channel %d\n", channel);
    }
}
```

Una de las funciones de C que aparece con frecuencia es **SET_PERI_REG_MASK** . Esta función establece en 1 los bits designados por una máscara en un registro. Ejemplo:

```
             SET_PERI_REG_MASK(SENS_SAR_DAC_CTRL2_REG,
SENS_DAC_CW_EN1_M);
```

La función C que establece en 0 los bits designados por una máscara en un registro es **CLEAR_PERI_REG_MASK** .

Nos interesará saber cómo reescribiremos **dac_cosine_enable(canal dac_channel_t)** en el lenguaje ADELANTE. Vemos que se menciona el registro SENS_SAR_DAC_CTRL2_REG. Definiremos este registro:

```
$3FF4889c defREG: SENS_SAR_DAC_CTRL2_REG   \ DAC output control
```

registro **SENS_SAR_DAC_CTRL2_REG**, los dos bits que nos interesan son b24 y b25. Definamos las máscaras correspondientes:

```
1 24 defMASK: mSENS_DAC_CW_EN1   \ selects CW generator as source
for PDAC1
1 25 defMASK: mSENS_DAC_CW_EN2   \ selects CW generator as source
for PDAC2
```

La programación 'bare-metal' que actúa directamente sobre los registros ESP32 no requiere definir en FORTH todos los registros y máscaras de registro como lo hace el lenguaje C. Limítese a los registros y máscaras esenciales para su aplicación.

Se recomienda utilizar los nombres de registros y máscaras de registros tal como aparecen en la documentación de Espressif, o en su defecto los nombres de registros utilizados en los códigos fuente del lenguaje C.

Administrar algunos dispositivos es muy complejo. La documentación de Espressif es escasa en ejemplos sobre el uso directo de registros.

El generador de números aleatorios

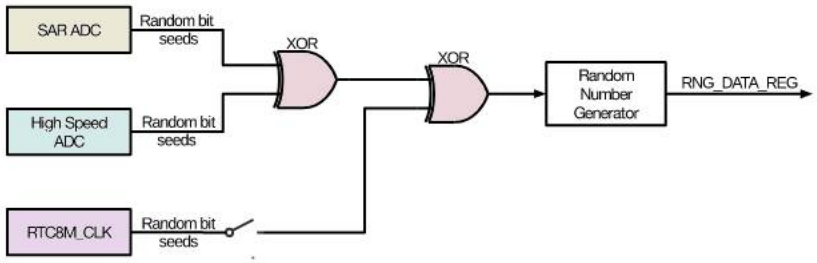
Característica

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:

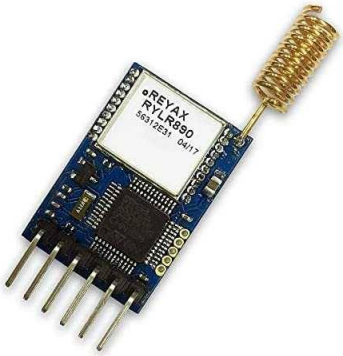
```
forth definitions
asm xtensa
$3FF75144 constant RNG_DATA_REG

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
```

El sistema de transmisión LoRa

LoRa es una tecnología de comunicaciones que utiliza una red de área amplia de bajo consumo. LoRa le permite conectar dispositivos y puertas de enlace de forma inalámbrica.

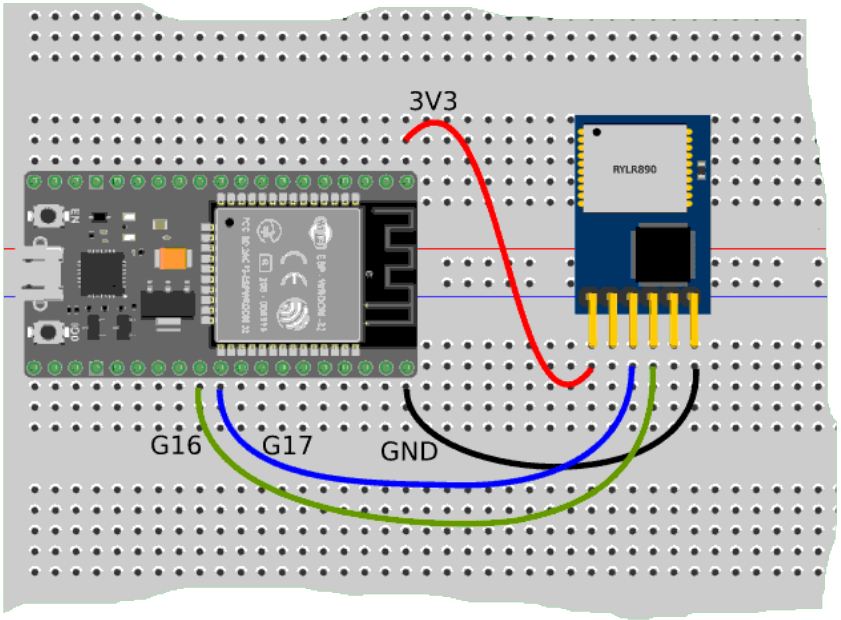
Este estándar aún no requiere ninguna suscripción. Ofrece comunicación **entre pares** .



LoRa, WiFi y Bluetooth son complementarios y no se superponen. En comparación con Wi-Fi y Bluetooth, que proporcionan un alcance muy corto, LoRa se beneficia de un ancho de banda muy estrecho. Los dispositivos conectados apenas utilizan los gateways o hubs el 1% del tiempo. Lo que reduce significativamente el ancho de banda. El tráfico es lento y unidireccional entre los sensores y la puerta de enlace. ¡LoRa es la mejor manera de comunicarte a lo largo de varios kilómetros, con muy poca potencia y de una forma muy sencilla!

Cableado del transmisor REYAX LR890 LoRa

El transmisor se conecta a la tarjeta ESP32 de esta manera:



ATENCIÓN: verifique la posición de los pines G16 y G17 en su tarjeta ESP32, que pueden ser diferentes según su versión de tarjeta ESP32.

El transmisor LoRa para ESP32

El módulo REYAX LR890 cuesta unos 15€. Pesa 7 gramos.

Su consumo, en transmisión, es de 43 mA (3,3V). En recepción es de 16,5 mA y puede bajar a 0,5 mA en modo SLEEP.

Para garantizar la transmisión punto a punto, se requieren dos módulos LoRa. Cada módulo es un transmisor y un receptor.

La tarjeta ESP32 se comunica a través de su puerto serie con el módulo LoRa. Todas las transmisiones entre la placa ESP32 y el transmisor LoRa se procesan mediante comandos AT. Ejemplo:

```
EN+ENVIAR=50.5,HOLA
```

Esta cadena es transmitida por la tarjeta ESP32 al módulo de transmisión LoRa:

- el módulo LoRa cambia al modo de transmisión y transmite esta cadena de caracteres
- Inmediatamente después de la transmisión, el módulo LoRa vuelve al modo de recepción.
- el módulo LoRa remoto recibe la cadena de caracteres.
- el módulo LoRa remoto puede reconocer esta recepción haciendo **+OK**

Un módulo LoRa puede comunicarse con una puerta de enlace LoRaWan. Generalmente es una caja conectada a un enrutador mediante una conexión Ethernet. Por tanto, es posible tener una aplicación web que se comunique con uno o más módulos LoRa.

Seguridad de transmisión LoRa

Un único módulo LoRa puede comunicarse con varios módulos LoRa remotos.

Estos módulos LoRa deben diferenciarse por su **NETWORKID** . El transmisor y el receptor deben tener el mismo **NETWORKID** .

Luego, cada módulo recibe una **ADDRESS** , por defecto 0. Esta dirección está entre 0..65535.

La transmisión se puede cifrar mediante una **clave AES** de 32 caracteres . Los módulos transmisor y receptor LoRa deben tener la misma **clave AES** . Si un módulo recibe un mensaje cifrado con una **clave AES** desconocida , ignorará el mensaje.

Y finalmente a cada módulo se le asigna una frecuencia de transmisión. Los módulos transmisor y receptor deben funcionar en la misma frecuencia.

Ejemplo de selección de frecuencia de 868,5 MHz.

```
\ select frequency 865.5 Mhz for LoRa transmission
32 string AT_BAND
s" AT+BAND=868500000" AT_BAND $!  \ set frequency at 868.5 Mhz
$0a AT_BAND c+$!
$0d AT_BAND c+$!     \ add CR LF code at end of command
AT_BAND Serial2.write drop
```

En la misma frecuencia podemos gestionar una flota de 65.535 módulos LoRa, teniendo cada módulo su dirección. Si transmitimos con la dirección 0, abordaremos todos los módulos LoRa.

Si añadimos la clave de cifrado AES, ¡habrá cientos de miles de módulos LoRa que podrán coexistir en un radio de unos pocos kilómetros!

El alcance de los módulos se puede aumentar cambiando la potencia de transmisión. También podemos actuar sobre la antena receptora. Con una antena direccional se puede alcanzar **de 20 a 30 kilómetros** de alcance...

Revisión del transmisor REYAX RYLR890 LoRa

Entorno de prueba requerido

Para probar nuestro transmisor REYAX RYLR890 LoRa, debe:

- utilice palabras de gestión de cadenas…. @todo: referencia en el archivo
- use una versión ESP32Forth con acceso al puerto UART2
- Conecte el transmisor REYAX RYLR890 LoRa de la siguiente manera:

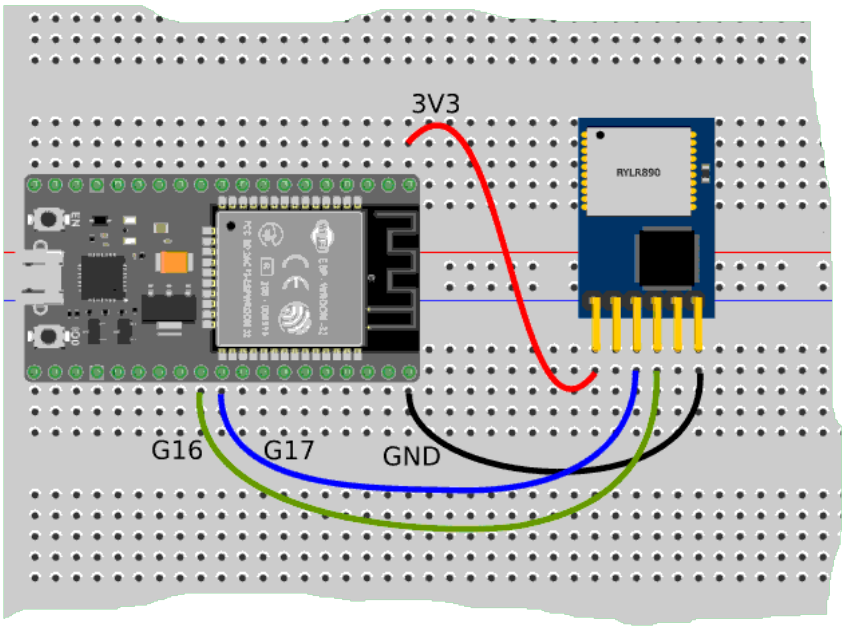


Figure 32: conectar el transmisor LoRa

Preparar la comunicación con el transmisor LoRa.

Todos los programadores que gestionan el puerto UART2 definen un área de memoria que servirá como buffer. Por nuestra parte crearemos directamente una variable alfanumérica:

```
128 string LoRaTX   \ buffer ESP32 -> LoRa transmitter
```

Aquí solo realizaremos pruebas de envío de comandos al transmisor REYAX RYLR890 LoRa y veremos cómo recuperar lo que devuelve este mismo transmisor. Por tanto necesitamos otra variable alfanumérica para la recepción:

```
128 string LoRaRX   \ buffer LoRa transmitter -> ESP32
```

Comencemos inicializando la transmisión en serie al transmisor LoRa. Aquí la velocidad es de 115200 baudios. Esta es la velocidad de transmisión predeterminada del transmisor LoRa:

```
Serial \ Seleccionar vocabulario de serie

\ inicializar Serie2
: Serial2.init ( -- )
    #SERIAL2_RATE Serial2.begin
  ;
```

Para nuestro comando de prueba al transmisor LoRa, seleccionamos la frecuencia de trabajo del transmisor, aquí 868,5 Mhz:

```
\ Configurar frecuencia LoRa
: .band8685 ( -- )
    s" AT+BAND=868500000" LoRaTX $!
    $0d LoRaTX c+$!
    $0a LoRaTX c+$!     \ add CR LF code at end of command
    LoRaTX Serial2.write drop
  ;
```

Finalmente, definimos una palabra que nos permitirá recuperar la respuesta del transmisor LoRa:

```
\ entrada del transmisor LoRa
: LoRaInput ( -- n )
    Serial2.available dup if
        LoRaRX maxlen$ nip
        Serial2.readBytes
        LoRaRX drop cell - !
    then
  ;
```

La palabra **`LoRaInput`** prueba si se ha recibido una transmisión de enlace serie desde el puerto serie UART2:

- si no hay recepción, devuelve 0
- si tiene caracteres, los almacena en la cadena alfanumérica LoRaRX y actualiza el tamaño de esa cadena.

Ejemplo de transmisión y recepción:

```
Serial2.init
.band8685
LoRaInput
```

Esto es lo que ofrece un volcado de memoria **`LoRaRX`** :

```
LoRaRX dump
```

```
--addr---  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  ------
chars-----
3FFF-87A0  05 00 00 00 2B 4F 4B 0D 0A 31 0D 0A 2B 4F 4B 0D  ....
+OK..1..+OK.
```

Si ejecutamos **LoRaRX** , recuperamos la dirección y sobre todo la longitud de todos los caracteres recibidos, incluidos los caracteres CR+LF ($0d $0a). Para gestionar solo caracteres estrictamente incluidos en el intervalo [0..0A..Za..z], debes reducir el tamaño de la cadena en dos unidades:

```
: LoRaType ( -- )
    LoRaRX dup 0 > if
        2 - type
    else
        2drop
    then
  ;
LoRaType  \ display: +OK
```

Aquí, al ejecutar **LoRaType** se muestra +OK, que es la respuesta a nuestro comando de prueba **AT+BAND=868500000** .

Configuración del transmisor REYAX RYLR890 LoRa

Antes de definir comandos para nuestro transmisor REYAX RYLR896 LoRa, definimos la palabra **crlf** :

```
: crlf ( -- )       \ misma acción que cr, pero adaptada para LoRa
    $0d emit
    $0a emit
  ;
```

El propósito de esta palabra crlf es completar la transmisión en el puerto UART2 desde la placa ESP32 al transmisor LoRa. La definición de esta palabra usa emitir. No te sorprendas. Más adelante veremos cómo aprovechar la ejecución vectorizada de palabras en el lenguaje FORTH para realizar la acción deseada en la emisión. Esta solución sorprenderá a los programadores principiantes de lenguajes FORTH. También mostrará cómo FORTH es mucho más flexible que muchos otros lenguajes de programación.

Parámetros esenciales

Aquí está la lista de parámetros esenciales para configurar su módulo LoRa.

La secuencia de uso del comando **AT** :

- Utilice **AT+ADDRESS** para configurar la DIRECCIÓN. Se considera que la DIRECCIÓN es la identificación del transmisor o receptor especificado.
- Utilice **AT+NETWORKID** para configurar el ID de red de Lora. Esta es una función de grupo. Sólo configurando el mismo NETWORKID los módulos pueden comunicarse entre sí. Si la DIRECCIÓN del destinatario especificado pertenece a un grupo diferente, no podrán comunicarse entre sí. El valor recomendado: 1 ~ 15
- Utilice **AT+BAND** para ajustar la frecuencia central de la banda inalámbrica. El transmisor y el receptor deben utilizar la misma frecuencia para comunicarse entre sí.
- Utilice **AT+PARAMETER** para ajustar la configuración inalámbrica de RF. El transmisor y el receptor deben configurar los mismos parámetros para comunicarse entre sí. Los parámetros se definirán de la siguiente manera:
    - **<Spreading Factor>** : Cuanto mayor sea el SF, mejor será la sensibilidad. Pero el tiempo de transmisión será más largo.
    - **<Bandwidth>** : Cuanto menor sea el ancho de banda, mejor será la sensibilidad. Pero el tiempo de transmisión será más largo.

- **<Coding Rate>** : La velocidad de codificación será más rápida si la configura en 1.
- **<Programmed Preamble>** : Código de preámbulo. Si el código del preámbulo es más grande, habrá menos posibilidades de perder datos. El código de preámbulo generalmente se puede configurar por encima de 10 si está bajo la autorización del tiempo de transmisión.
  * Comunicación hasta 3 km: configuración recomendada "AT+PARAMETER=10.7,1.7"
  * Más de 3 km: configuración recomendada "AT+PARAMETER=12.4,1.7"

- Utilice **AT+SEND** para enviar datos a la DIRECCIÓN especificada. Debido al programa utilizado por el módulo, la parte de la carga útil aumentará en más de 8 bytes en comparación con la longitud real de los datos.

Es necesario pasar `crlf` al final de todos los comandos **AT** .

Debes esperar a que el módulo responda `+OK` para poder ejecutar el siguiente comando **AT** .

ADDRESS Define la dirección del módulo.

Cada módulo de transmisión LoRa debe tener una dirección personal.

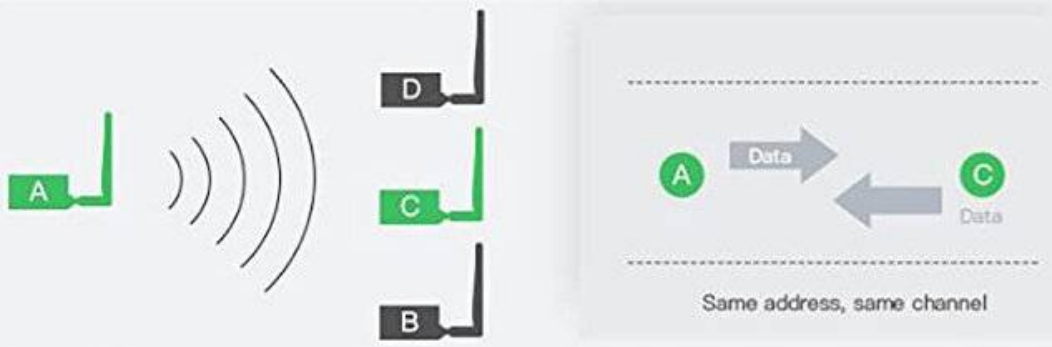
sintaxis	respuesta
AT+ADDRESS=<address>	+OK
AT+ADDRESS=?	+ADDRESS=22

```
\ Configure la DIRECCIÓN del transmisor LoRa:
\ s" " valor en el intervalo [0..65535][?] (predeterminado 0)
: ATaddress ( addr len -- )
    ." AT+ADDRESS="
    type  crlf
  ;
```

<Address>=0~65535(predeterminado 0)

Ejemplo: establezca la dirección del módulo en `22` . La configuración se almacenará en LoRa.

```
s" 22" Ataddress
```



AT Test Disponibilidad de LoRa

sintaxis	respuesta
AT	+OK

```
\ Disponibilidad de prueba LoRa
: AT_ ( -- )
    ." AT"
    type  crlf
  ;
```

BAND Configuración de la frecuencia RF

sintaxis	respuesta
AT+BAND=<parameter>	+OK
AT+BAND=?	+BAND=868500000

```
\ Configure la BANDA del transmisor LoRa:
\ s" " el valor es la frecuencia de RF, unidad Hz
: ATband ( addr len -- )
    ." AT+BAND="
    type  crlf
  ;
```

El parámetro es la frecuencia de RF, la unidad es Hz: 915000000Hz (predeterminado: RYLY89x)

Ejemplo: Seleccione la frecuencia a 868500000Hz:

```
s" 868500000" ATband
```

CPIN Establece la contraseña de red AES128

sintaxis	respuesta
AT+CPIN=<password>	+OK

sintaxis	respuesta
AT+CPIN=?	+CPIN=FABC0002EEDCA…..

Contraseña: Contraseña AES de 32 caracteres desde 00000000000000000000000000000001 a FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF.

Se acepta el intercambio si ambos módulos tienen la misma contraseña. Después del restablecimiento, se elimina la contraseña anterior.

```
\ Establecer la contraseña AES32:
\ El valor s"<parámetro>" es una contraseña AES de 32 caracteres
\ de  00000000000000000000000000000001 a
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
: ATcpin ( addr len -- )
    ." AT+CPIN="
    type  crlf
  ;
```

Ejemplo: Seleccione esta contraseña: FABC0002EEDCAA90FABC0002EEDCAA90

```
s" FABC0002EEDCAA90FABC0002EEDCAA90" ATcpin
```

CRFOP Selecciona la potencia de RF de salida.

sintaxis	respuesta
AT+CRFOP=<power>	+OK
AT+CRFOP=?	+CRFOP=10

Potencia: entre 0..15, 15dBm (predeterminado)

```
\ Establezca la potencia de salida CRFOP RF:
\  s" " el valor es la potencia de salida de RF entre 0..15
: ATcrfop ( addr len -- )
    ." AT+CRFOP="
    type  crlf
  ;
```

Ejemplo, seleccione la potencia de salida a 10dBm:

```
s" 10" ATcrfop
```

FACTORY Establece todas las configuraciones actuales a los valores predeterminados

Establece todas las configuraciones actuales según los valores predeterminados del fabricante.

sintaxis	respuesta
AT+FACTORY	+FACTORY

```
\ Resetear el transmisor LoRa con parámetros de FÁBRICA
: ATfactory ( -- )
    ." AT+FACTORY"
    crlf
  ;
```

IPR Establece la velocidad en baudios de UART

sintaxis	respuesta
AT+IPR=<parameter>	+OK
AT+IPR=?	+IPR=38400

Parámetro de baudios UART:

- 300
- 1200
- 4800
- 9600
- 19200
- 28800
- 38400
- 57600
- 115200 (predeterminado)

La configuración se almacenará en la EEPROM.

MODE Selecciona el modo de trabajo.

sintaxis	respuesta
AT+MODE=<parameter>	+OK
AT+MODE=?	+MODE=1

Configuración:

- 0: Modo de transmisión y recepción (predeterminado).
- 1: Modo de suspensión.

```
\ Establecer MODO de trabajo:
\  s" " el valor es [0,1]
\ 0 (predeterminado) Modo de transmisión y recepción
\ 1 Modo de suspensión
: ATmode ( addr len -- )
    ." AT+MODE"
    type  crlf
```

```
  ;
```

NETWORKID Selecciona el ID de red

sintaxis	respuesta
AT+NETWORKID=<Network ID>	+OK
AT+NETWORKID=?	+NETWORKID=6

```
\ Establecer ID DE RED:
\ El valor de s" " es [0..16] (valor predeterminado de bahía 0)
: ATnetworkid ( addr len -- )
    ." AT+NETWORKID"
    type  crlf
  ;
```

ID de red: 0~16 (0 por defecto)

Ejemplo: seleccione ID de red en 6

La configuración se almacenará en la EEPROM.

El "0" es el identificador público de LoRa. No se recomienda utilizar 0 para configurar el ID DE RED.

```
s" 6" Atnetworkid
```

PARAMETER definición de parámetros de RF

sintaxis	respuesta
AT+PARAMETER=<Spreading Factor>, <Bandwidth>,<Coding Rate>, <Programmed Preamble>	+OK
AT+PARAMETER=?	+PARAMETER=7,3,4,5

Parámetros:

- Factor de dispersión
  - 7~12, (predeterminado 12)
- Ancho de banda/ancho de banda (0~9):
  - 0: 7,8 KHz (no recomendado, por encima de las especificaciones)
  - 1: 10,4 KHz (no recomendado, por encima de las especificaciones)
  - 2: 15,6 kHz
  - 3: 20,8 kHz
  - 4: 31,25 kHz
  - 5: 41,7 KHz
  - 6: 62,5 KHz

    - 7: 125 KHz (predeterminado).
    - 8: 250 kHz
    - 9: 500 kHz
- Tasa de codificación
    - 1~4, (predeterminado 1)
- Preámbulo programado
    - 4~7 (predeterminado 4)

Factor de dispersión

factor de dispersión	Tasa de bits / flujo
7	5469 bps
8	3125 bps
9	1758 bps
10	977 bps
11	537 bps
12	293 bps

Tasa de codificación

La modulación LoRa también agrega corrección de errores directa (FEC) en cada transmisión de datos. Esta implementación se realiza codificando datos de 4 bits con redundancias de 5 bits, 6 bits, 7 bits o incluso 8 bits. El uso de esta redundancia permitirá que la señal LoRa cubra las interferencias. El valor de la tasa de codificación debe ajustarse según las condiciones del canal utilizado para la transmisión de datos. Si hay demasiada interferencia en el canal, se recomienda aumentar el valor de la tasa de codificación.

Sin embargo, aumentar el valor CR también aumentará la duración de la transmisión.

Ejemplo: configuración de parámetros como se muestra a continuación:
<Factor de dispersión> 7, <Ancho de banda> 20,8 KHz, <Velocidad de codificación> 4, <Preámbulo programado>5,

```
s" 7,3,4,5" Atparameter
```

REINICIAR software

sintaxis	respuesta
EN+REINICIAR	+OK

```
\ RESETABLECER el transmisor LoRa
: ATreset ( -- )
```

```
    ." AT+RESET"
    crlf
  ;
```

SEND envío de datos a la dirección designada

sintaxis	respuesta
AT+SEND=<Address>,<Payload Length>,<Data>	+OK
AT+SEND=?	+SEND=50,5,HELLO

<Dirección>0~65535, cuando <dirección> es 0, enviará datos a todas las direcciones (de 0 a 65535).

<Longitud de la carga útil>Máximo 240 bytes

<Datos>Formato ASCII

Cuarto código para ESP32Forth:

```
\ convertir un número a una cadena decimal
: .n ( n ---)
    base @ >r decimal
    <# #s #> type
    r> base !
  ;
\ ENVIAR Enviar datos a la dirección de la cita
: ATsend { addr len address -- }
    ." AT+SEND="
    address .n  [char] , emit
    len     .n  [char] , emit
    addr len type crlf
  ;
```

Ejemplo: envía la cadena **HOLA** a la dirección 50:

```
s" HELLO" 50 ATsend     \ display: AT+SEND=50;5;HELLO
```

VER para solicitar la versión del firmware

```
\ VER para consultar la versión del firmware
: ATver ( -- )
    ." AT+VER"
    crlf
  ;
```

Códigos de resultado de error

- +ERR=1 no hay “enter” o $0D $0A al final del comando AT
- +ERR=2 el encabezado del comando AT no es una cadena "AT"
- +ERR=3 no hay ningún símbolo “=" en el comando AT

- +ERR=4 comando desconocido
- +ERR=10 TX llega a tiempo
- +ERR=11 RX superado
- +ERR=12 error CRC
- +ERR=13 datos TX de más de 240 bytes
- +ERR=15 Error desconocido

Vectorización de emisiones de personajes.

Si has seguido el desarrollo de nuestras palabras para configurar el transmisor REYAX RYLR890 LoRa hasta este punto, seguro que algo te ha sorprendido:

```
\ Configure la DIRECCIÓN del transmisor LoRa:
\ s" <address>" valor en el intervalo [0..65535][?] (predeterminado
0)
: ATaddress ( addr len -- )
    ." AT+ADDRESS="
    type  crlf
  ;
```

Porque, a menos que me equivoque, esta secuencia `." AT+ADDRESS="` envía la cadena de caracteres a nuestro terminal, y no al transmisor a través del puerto serie UART2, ¡por lo tanto al transmisor LoRa!

Entendemos tu sorpresa. Y veremos cómo desviar el flujo de caracteres al transmisor LoRa sin cambiar nada en la definición de nuestra palabra `ATaddress`.

Comprender la vectorización en FORTH

El lenguaje FORTH tiene ciertas ventajas que son completamente inexistentes en muchos otros lenguajes de programación. Entre estos activos, mencionemos la palabra `defer`. Esta palabra permite crear una palabra cuya acción sea diferida:

```
defer myWords
```

`¡¡Defer` crea una palabra `de myWords` que no hace NADA!!!

¡Sí!

Ahora nos toca a nosotros darle acción. Veamos esta definición:

```
: (myWords) ( -- )
    cr ." I display my words: "
  ;
```

Para que myWords se ejecute `(myWords)`, obtenemos el código de acción de `(myWords)` y lo asignamos a `myWords`:

```
' (misPalabras) son misPalabras
```

A partir de ahora, si escribimos `misPalabras` se ejecutará la acción definida en `(misPalabras) .`

DE ACUERDO. Pero aquí, ¿es necesario realizar tal sobrecarga de código si simplemente podemos ejecutar `(myWords)` ?

Y tienes toda la razón al hacer esta pregunta. Pero puedes cambiar la acción de `myWords` :

```
' vlist is myWords
```

Ahora, si escribimos `myWords`, se ejecuta la palabra `vlist .`

Veremos cómo utilizar este mecanismo para modificar el comportamiento de ESP32Forth…

Vectorización en ESP32Forth

Comencemos con un poco de ingeniería inversa. Profundizando en el código ESP32Forth, encontramos esto para la palabra `."` :

```
: ."
    postpone s" state @ if postpone type else type then ; immediate
```

Aquí la palabra que nos interesa es `tipo` cuya definición es:

```
defer type
```

Ahhhh..... ¿Estás empezando a entender?

Qué acción toma una ejecución `type`? Encontramos esto en el código fuente de ESP32Forth:

```
: serial-type ( a n -- ) Serial.write drop ;
: default-type serial-type ;

' default-type is type
```

Si nos interesa la palabra `emit`, encontramos esta definición en el código fuente de ESP32Forth:

```
: emit ( n -- )
    >r rp@ 1 type rdrop ;
```

Aquí nuevamente encontramos `type`.

Por tanto, es sobre este tipo de palabra que actuaremos para desviar la transmisión de caracteres hacia el puerto serie UART2.

Vectorizar el tipo al puerto serie UART2

Es mirando la definición de `serial-type` que definimos nuestra versión para transmitir al puerto serie UART2:

```
: serial2-type ( a n -- )
    Serial2.write drop ;
```

¿Hasta dónde ha llegado? ¿No es demasiado difícil?

Ahora, para desviar el flujo de transmisión de caracteres de ESP32Forth al puerto serie UART2, simplemente ejecute la secuencia `' serial2-type is type`.

Pero si hace eso, tendrá algunas dificultades para volver al comportamiento normal de ESP32Forth a menos que restaure `type` a su acción inicial con la secuencia `' default-type is type`.

Resumamos estas secuencias en estas dos palabras:

```
: typeToLoRa ( -- )
    ['] serial2-type is type
  ;

: typeToTerm ( -- )
    ['] default-type is type
  ;
```

Y ahora, para ejecutar nuestra palabra `ATaddress` haciéndola transmitir los caracteres al puerto serie UART2, simplemente escriba:

```
typeToLoRa
s" 45" ATaddress   \ send AT+ADDRESS=45  to UART2
typeToTerm
```

Y ahí espero tu comentario: "¿pero cuál es el punto de pasar por la vectorización?"

En nuestro caso, la vectorización ofrece muchas ventajas:

- escribir código simple con palabras ya conocidas del diccionario FORTH de ESP32Forth;
- ofrece la posibilidad de probar todas las palabras de configuración del transmisor LoRa hacia el terminal
- posibilidad de desviar el flujo a otro dispositivo, por ejemplo I2S o UART1, sin tener que reescribir estas definiciones de parámetros...

Independientemente de nuestra gestión de los parámetros del transmisor LoRa, entendemos que basta con explotar este mismo mecanismo de vectorización de los caracteres recibidos del puerto serie UART2 para tomar fácilmente el control de ESP32Forth desde este puerto serie.

¡Esto es en realidad lo que hace ESP32Forth si activas el puerto WiFi o Bluetooth! Los invito a explorar el código fuente de ESP32Forth. Mire la definición `del servidor` :

```
: server ( port -- )
```

```
   server
   ['] serve-key is key
   ['] serve-type is type
   webserver-task start-task
;
```

Reescribir un listado completo

Los parámetros mínimos para comunicar entre tarjetas ESP32+LoRa son: frecuencia y dirección:

```
\ *** defining LoRa Setup words *****************************

create $crlf
    $0d c,  $0a c,

: crlf ( -- )       \ same action as cr, but adapted for LoRa
    $crlf 2 type
  ;

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

\ Set the BAND of LoRa transmitter:
\  s" " value is RF frequency, unit Hz
: ATband ( addr len -- )
    s" AT+BAND=" type
    type  crlf ;
```

Los transmisores LoRa, sacados de su embalaje original, teóricamente se comunican a 115200 baudios con la tarjeta ESP32:

```
\ 115200 speed communication for LoRa REYAX
115200 value #SERIAL2_RATE

\ definition of OUTput and INput buffers
128 string LoRaRX   \ buffer LoRa transmitter -> ESP32

Serial \ Select Serial vocabulary

\ initialise Serial2
: Serial2.init ( -- )
    #SERIAL2_RATE Serial2.begin
  ;
```

También recuperamos la palabra **LoRaInput** que lee los mensajes devueltos por el transmisor LoRa en el puerto UART2. La palabra **rx.** Se ha añadido para facilitar el manejo:

```
\ entrada del transmisor LoRa
: LoRaInput ( -- n )
    Serial2.available if
        LoRaRX maxlen$ nip
        Serial2.readBytes
        LoRaRX drop cell - !
    else
        0 LoRaRX drop cell - !
    then
  ;

: rx.
    LoRaINPUT
    loRaRX type
  ;
```

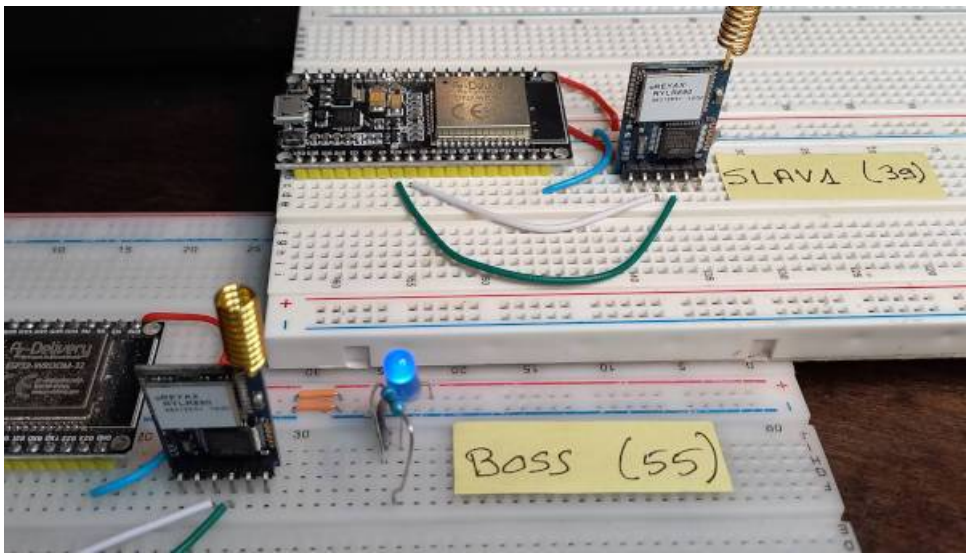
Aquí, las palabras **typeToLoRa** y **typeToTerm** se utilizan para transferir la visualización de texto desde el terminal al puerto UART2:

```
\ *** definición de palabras diferidas
*******************************

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
  ;
```

Aquí tenemos las CUARTAS palabras necesarias y suficientes para configurar nuestros tres transmisores REYAX RYLR890 LoRa.

Configuración de transmisores LoRa

En esta foto tienes etiquetas BOSS y SLAV1. Se trata de simples post-its adhesivos pegados a las placas de prueba.



Crearemos tres constantes asociadas a estas etiquetas:

```
55 constant LoRaBOSS
39 constant LoRaSLAV1
40 constant LoRaSLAV2
```

Para comunicarse entre sí, nuestros transmisores LoRa deben configurarse para utilizar la misma frecuencia. La frecuencia elegida es 868,5 MHz, o 868500000 Hz:

```
: emptyRX ( -- )
    LoRaINPUT
  ;

: SETband ( -- )
    emptyRX
    typeToLoRa
    s" 868500000" ATband
    typeToTerm
  ;
```

Comencemos a configurar nuestro primer transmisor LoRa:

```
serial2.init
SETband
rx.   \ display +OK
```

Si todo va bien, realizando **`rx.`** muestra **`+Aceptar`** .

Este es el mensaje devuelto por el transmisor LoRa. Es posible recibir un mensaje de error, como +ERR=1. Repita el comando de configuración.

La frecuencia se indica en 9 dígitos, sin separadores ni espacios. La unidad es Hz [12].

12Para FRANCIA, la banda de frecuencia libre va de 863 MHz a 868,6 MHz. Fuente: ARCEP El "portal de bandas libres"

El módulo REYAX RYLR896 LoRa puede operar frecuencias de 862 MHz a 1020 MHz.

ATENCIÓN : ¡la antena debe estar sintonizada a la frecuencia utilizada! La antena instalada en el módulo REYAX LoRa está sintonizada para frecuencias de alrededor de 868 MHz. El uso de una antena mal sintonizada reducirá considerablemente la eficiencia de transmisión del módulo LoRa.

Los módulos LoRa transmiten en banda estrecha. Elija cualquier frecuencia de las frecuencias autorizadas en su país.

Determinar la dirección de los transmisores LoRa

Para estar operativo, todos los transmisores de una red deben estar en la misma frecuencia. Cuando desee transmitir un mensaje a un transmisor en particular, deberá indicar la dirección del transmisor destinatario. Por ejemplo, si **BOSS** quiere enviar un mensaje a **SLAV1** , transmitiremos el mensaje al transmisor que tiene la dirección 39.

ATENCIÓN : ¡no se pueden tener dos transmisores con la misma dirección en la misma frecuencia!

Aquí se define la palabra que permite configurar la dirección 55 para el transmisor **BOSS** :

```
: SETaddress ( n -- )
    emptyRX
    typeToLoRa
    str ATband
    typeToTerm
  ;

LoRaBOSS SETaddress
rx.  \ display +OK
```

Podemos tomar cualquier valor para cada transmisor, en el intervalo [1..65535]. La dirección 0 está reservada para transmisiones a todos los transmisores LoRa que escuchan en la misma frecuencia.

Ahora que nuestro transmisor **BOSS** está configurado, podemos desconectarlo de la PC y enchufar el que tiene la etiqueta **SLAV1** . Compilamos el script fuente y comenzamos a configurar **SLAV1** :

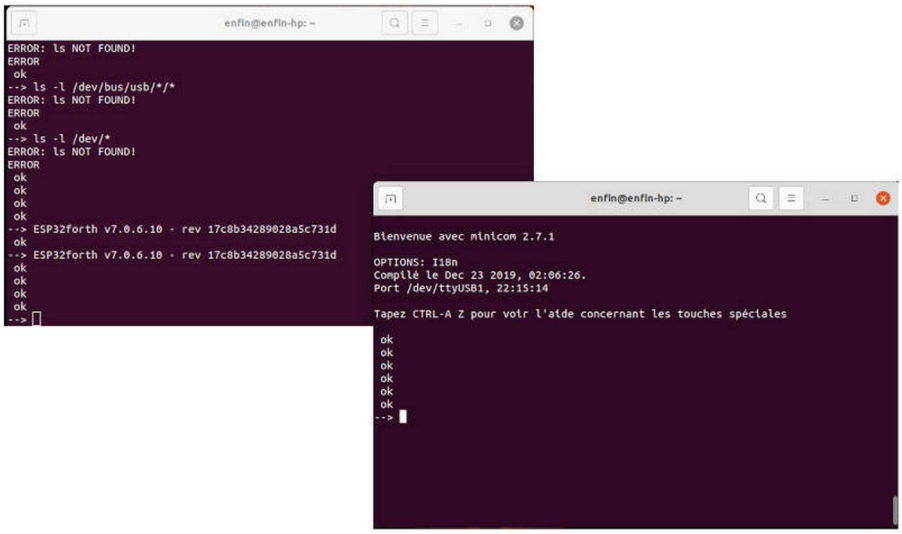
```
serial2.init
SETband
rx.  \ display +OK

LoRaSLAV1 SETaddress
rx.  \ display +OK
```

Comunicación entre dos transmisores REYAX RYLR890 LoRa

Para inicializar nuestros transmisores REYAX RYLR890 LoRa, debe:

- tener dos transmisores REYAX RYLR890 LoRa con la tarjeta ESP32
- conecte cada placa ESP32 a un puerto USB disponible en su PC

Aquí, bajo Linux, hemos abierto dos terminales Minicom:



Comandos de Linux para abrir estas dos terminales:

- desde el teclado, inicie la terminal de comandos usando CTRL-ALT-T
- en la ventana de comandos, al escribir sudo minicom se inicia el terminal minicom conectado a /dev/ttyUSB0
- En el mensaje, escriba la contraseña del administrador de Linux. El primer terminal da acceso a tu primera tarjeta ESP32
- nuevamente, desde el teclado, inicie la terminal de comandos usando CTRL-ALT-T
- en esta nueva ventana de comando, escriba sudo minicom -D /dev/ttyUSB1 que inicia otro terminal minicom conectado a /dev/ttyUSB1
- En el mensaje, escriba la contraseña del administrador de Linux. Este otro terminal da acceso a la segunda tarjeta ESP32

Transmisión de BOSS a SLAV2

La lista de nuestro código FORTH es sólo ligeramente diferente a la del capítulo anterior. Simplemente eliminamos las palabras **ATaddress** y **ATband** . Estas palabras ya no son necesarias para configurar nuestros transmisores LoRa **BOSS** , **SLAV1** y **SLAV2** .

Una vez configurado, un transmisor LoRa mantiene esta configuración, incluso cuando está apagado. Encender una tarjeta ESP32 y su transmisor LoRa no cambia la configuración del transmisor LoRa.

Las palabras **ATaddress** y **ATband** se reemplazan por **Atsend** :

```
\ SEND Enviar datos a la dirección de la cita
: ATsend { addr len address -- }
    ." AT+SEND="
    address n.  [char] , emit
    len     n.  [char] , emit
    addr len type crlf
  ;
```

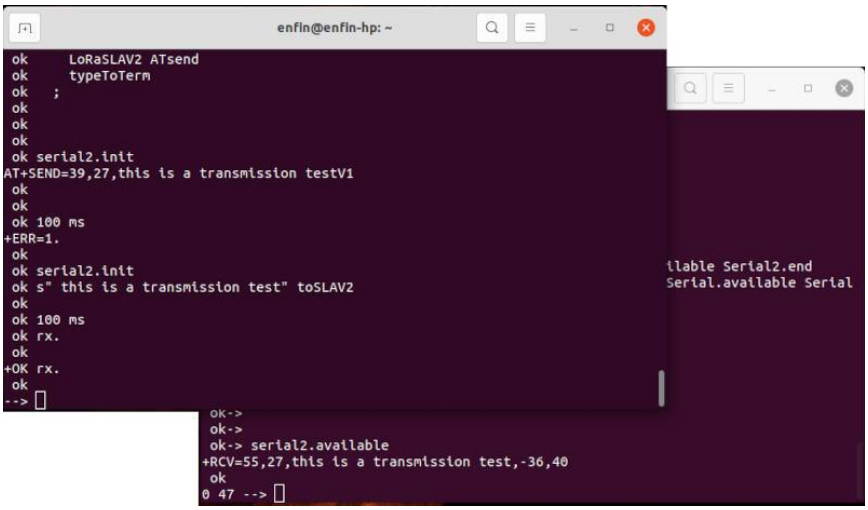
Ahora integramos esta palabra en **toSLAV2** :

```
: toSLAV2 ( addr len -- )
    emptyRX
    typeToLoRa
    LoRaSLAV2 ATsend
    typeToTerm
  ;
```

Compilamos el mismo programa en cada ESP32 (**BOSS** y **SLAV2**). Es muy fácil. Simplemente copie del listado y péguelo en la terminal. Cada tarjeta ESP32 compilará su lista en unos diez segundos.

Aquí, las dos ventanas de nuestro terminal minicom. La ventana izquierda le permite controlar **BOSS** .

La ventana derecha controla **SLAV2** :



Probemos la transmisión de **BOSS** a **SLAV2** . Para hacer esto, coloque el puntero del mouse en la ventana izquierda y escriba directamente:

```
serial2.init
s" this is a transmission test" toSLAV2
rx.  \ display: +OK
```

¿Estamos seguros de que **SLAV2** recibió el mensaje? Nada más fácil.

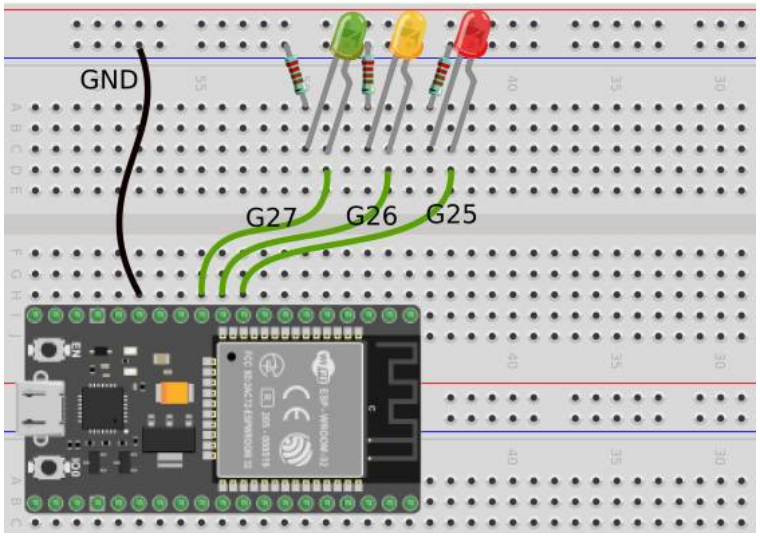
Colocamos el cursor del ratón en la ventana derecha y simplemente escribimos:

```
serial2.init
rx.  \ display: +RCV=55,27,this is a transmission test,-36,40
El resultado de esta transmisión por parte de SLAV2 se almacena en
la variable alfanumérica LoRaRX .
```

Interfaz de una transmisión LoRa con ESP32Forth

Para demostrar la increíble flexibilidad del lenguaje FORTH, y más particularmente de la versión ESP32Forth en ESP32, usaremos el programa utilizado en el capítulo *Gestión de un semáforo con ESP32* .

El problema con las definiciones de este capítulo es que el control de los LED utiliza los terminales GPIO del enlace serie. Por tanto, movemos la conexión del LED así:



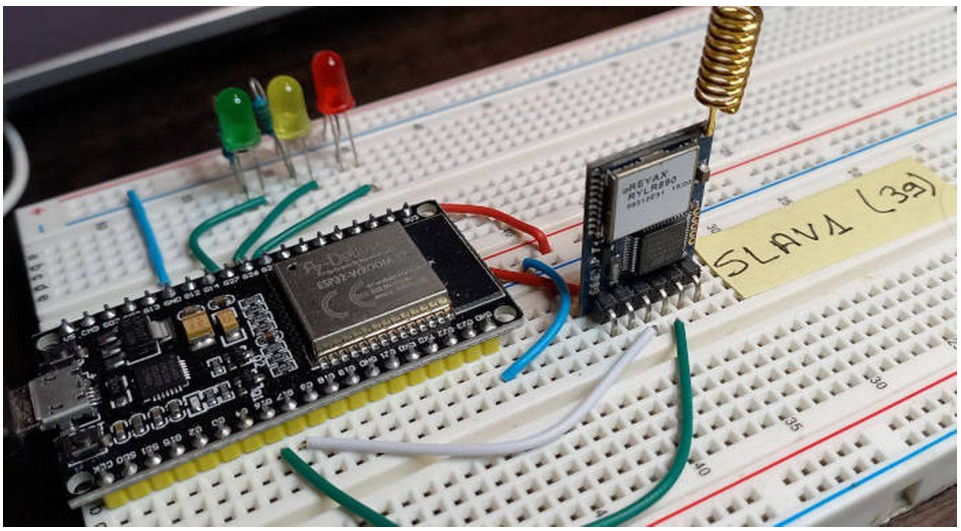
Aquí tenéis la única adaptación de código que se hace para adaptarse a la nueva conexión de los LED:

```
\ nuevo código
27 constant ledGREEN          \ green  LED on GPIO2
26 constant ledYELLOW         \ yellow LED on GPIO21
25 constant ledRED            \ red    LED on GPIO17
```

Aquí están las secuencias de códigos en el lenguaje FORTH para encender o apagar selectivamente cada LED. Estas secuencias se pueden ejecutar desde la ventana del terminal conectado a la tarjeta ESP32:

```
LEDinit
ledGREEN high       \ set GREEN led on
ledRED   high       \ set RED led on
ledGREEN low        \ set GREEN led off
```

Y son estas secuencias, y sólo éstas, las que serán transmitidas y recibidas por los transmisores LoRa. Aquí está el montaje de los LED y el transmisor LoRa en nuestra placa de prueba llamada **SLAV1** :



El programa paralelo del transmisor LoRa llamado BOSS

Completaremos el ajetreado programa de comunicaciones. Partimos de lo descrito en el capítulo anterior.

El listado incluye los componentes esenciales que permiten la transmisión LoRa desde la tarjeta ESP32 marcada como BOSS.

Simplemente añadimos algunas definiciones simples para ejecutar remotamente el encendido y apagado de los LED que se encuentran en la tarjeta marcada **SLAV1** :

```
\ 55 constant LoRaBOSS
39 constant LoRaSLAV1
\ 40 constant LoRaSLAV2

: toSLAV1 ( addr len -- )
    emptyRX
    typeToLoRa
    LoRaSLAV1 ATsend
    typeToTerm
  ;

: REDhigh ( -- )
    s" LEDred high"     toSLAV1
  ;

: REDlow ( -- )
    s" LEDred low"      toSLAV1
  ;
```

```
: YELLOWhigh ( -- )
    s" ledYELLOW high"  toSLAV1
  ;

: YELLOWlow ( -- )
    s" ledYELLOW low"   toSLAV1
  ;

: GREENhigh ( -- )
    s" ledGREEN high"   toSLAV1
  ;

: GREENlow ( -- )
    s" ledGREEN low"    toSLAV1
  ;
```

Creamos una definición por comando, con el objetivo de hacerlo lo más sencillo posible. Eres libre de crear una forma más interactiva. Éste no es el propósito de este capítulo. Por el momento, una vez conectada la tarjeta **BOSS** y compilado el código, si queremos transmitir un comando a **SLAV1** , simplemente tecleamos en la terminal:

```
serial2.init
REDhigh
```

mensaje `LEDred high` a **SLAV1** . La última fase será ejecutar este comando como si estuviera escrito desde un terminal conectado a **SLAV1** .

Recepción y ejecución de comandos FORTH por SLAV1

Resumamos: el transmisor BOSS envía un mensaje, por ejemplo `LEDrojo alto` al transmisor SLAV1. El transmisor SLAV1 recibe el mensaje y ejecuta `RXdecode` para almacenar este comando FORTH en la variable alfanumérica `RCVdata` .

```
      x . . . . "LEDred high" . . . . . x
      |                                  |
+-----+-+                          +-----+-+
| BOSS  |                          | SLAV1 |
+-------+                          +-------+
                                       | RXdecode
                                       +-----> RXdata: LEDred high
```

Ejecutando un comando recibido por LoRa

En nuestro diagrama, tenemos dos placas ESP32, cada una con un transmisor LoRa:

- **BOSS** (dirección 55) que transmite comandos ADELANTE
- **SLAV1** (dirección 39) que recibe estos comandos FORTH

La única diferencia, con los comandos FORTH ingresados directamente en el teclado de la PC y transmitidos por el programa terminal conectado a SLAV1, se refiere a los comandos transmitidos por LoRa y almacenados en `RXdata` .

Cómo ejecutar estos comandos almacenados en `Rxdata` ? La respuesta es sorprendentemente simple:

```
RCVdata evaluate
```

¡¡¡No necesitamos absolutamente nada más PARA INTERCONECTAR la transmisión LoRa con ningún programa integrado en una placa ESP32!!!

Aquí hay una definición segura de esta interfaz:

```
: RXinterface ( -- )
    RCVdata ?dup if
        evaluate
    else
        2drop
    then
  ;
```

Aquí hay algunas manipulaciones en FORTH para probar esta interfaz:

```
LEDinit                 \ initialize GPIOs
s" LEDred high" RCVdata $!
RCVdata RXinterface     \ turn RED led on
s" LEDred low" RCVdata $!
RCVdata RXinterface     \ turn RED led off
```

Cumplimos nuestra promesa: actuar desde LoRa en cualquier programa sin cambiar una sola línea de código.

En cualquier tarjeta ESP32 podrás compilar y probar fácilmente todas las funcionalidades de tus programas con el terminal.

Para interconectar estos programas, bastará con añadir la capa de transmisión LoRa y su código de interfaz.

El transmisor remoto sólo tendrá que enviar comandos ADELANTE para actuar sobre sus programas.

¡Solo el lenguaje FORTH permite una transmisión tan simple -> interfaz de aplicación!

Ahora veamos el último punto: leer periódicamente el buffer de recepción del transmisor LoRa....

Bucle de gestión de transmisión LoRa

El transmisor LoRa está conectado al puerto serie UART2. Cuando se recibe una transmisión, la palabra `Serial2.available` indica la cantidad de bytes en espera en el búfer serial UART2. Si no hay transmisión, el valor reportado por `Serial2.available` será cero. Aquí está el código para probar la presencia de caracteres recibidos por UART2:

```
Serial
\ final loop
: LoRaLoop ( -- )
    begin
        Serial2.available   \ not 0 if chars availalble
        if
            100 ms          \ ensures that entire transmission is
received
            LoRaRX maxlen$ nip
            Serial2.readBytes
            LoRaRX drop cell - !
            RXdecode        \ analyse content of LoRa message
            RXinterface     \ interpret content or RCVdata
        then
        pause               \ skip to next task
    again
  ;
```

El código `LoRaLoop` utiliza un bucle infinito. No se recomienda ejecutar esta palabra tal como está. Si hace esto, ya no tendrá control del intérprete FORTH de ESP32Forth.

Para usar `LoRaLoop` sin bloquear el intérprete FORTH, definiremos una nueva tarea `my-loop` como esta:

```
' LoRaLoop 100 100 task my-loop
my-loop start-task
```

A partir de este momento, cualquier transmisión realizada desde la tarjeta marcada como **BOSS** será interpretada en esta tarjeta marcada como **SLAV1** .

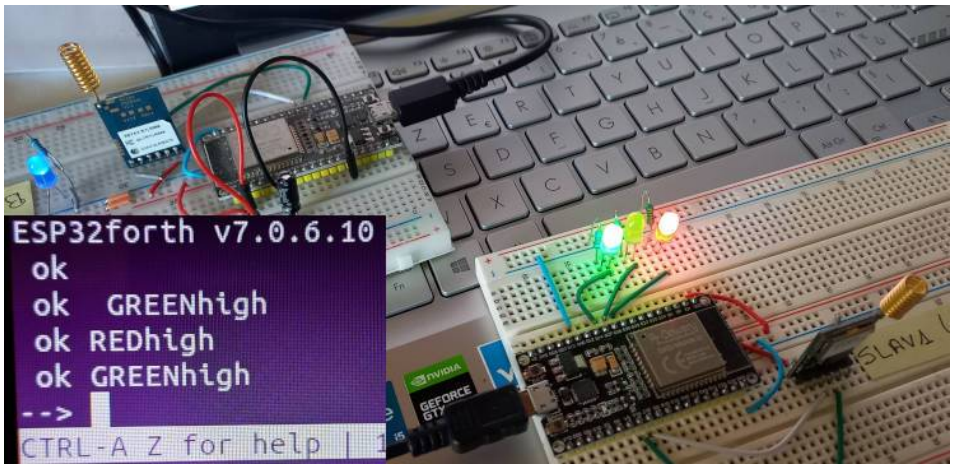
Para que toda nuestra programación quede persistente en nuestra tarjeta **SLAV1** , finalizamos la inicialización general:

```
\ Comunicación de 115200 velocidades para LoRa REYAX
115200 value #SERIAL2_RATE
```

```
Serial
: mainInit ( -- )
    cr ." Starting SLAV1 LoRa" cr
    LEDinit
    #SERIAL2_RATE Serial2.begin     \ initialise Serial2
    my-loop start-task
  ;
startup: mainInit
```

A partir de este momento, una vez compilado el programa en la tarjeta ESP32 marcada como **SLAV1** , al reiniciar la tarjeta se ejecutará la palabra mainInit, lo cual se confirma con el mensaje Starting **SLAV1** LoRa que normalmente debería mostrarse.

Aquí hay una foto de las acciones realizadas desde la tarjeta BOSS, insertada en la parte inferior izquierda:



En la tarjeta marcada **SLAV1** , los LED reaccionan con una latencia de uno a dos segundos. Este retraso es normal. Resulta del protocolo LoRa, que es ciertamente lento, pero extremadamente robusto. En la foto de arriba, las pruebas se realizaron a una distancia de un metro. Para la foto se juntaron las tarjetas **BOSS** y **SLAV1 .**

Si deja **SLAV1** conectado al terminal, aún tendrá el control del intérprete FORTH. ¡Esto también es normal! La palabra **LoRaLoop** se ejecuta en multitarea.

Desde sus orígenes, el lenguaje FORTH ha sido multitarea. Ya estaba en versiones de MS-DOS cuando MS-DOS no era multitarea.

Con ESP32Forth continuamos con las funcionalidades de FORTH, incluyendo las posibilidades de activar tareas concurrentes. En nuestro caso específico, la tarea de monitor le da control sobre el intérprete mientras administra los LED desde la tarea **LoRaLoop** .

- # Una interfaz WEB sencilla para ESP32Forth

Autor: Václav POSELT

Reinicié el uso de Forth después de años de no programar con FlashFORTH en Atmega 328 y Arduino. Después de crear mi primera construcción, fue necesario construir un panel de control para la electrónica, los botones y la pantalla también. Pensé que era hora de IoT y control inalámbrico, así que mejor ahorrar el trabajo de construcción y controlar todo de forma inalámbrica. Para esto cambié a ESP32 con WiFi y BT, encontré docenas de ejemplos de programas de interfaz web en Arduino C con JavaScript, pero nada en ESP32Forth en ESP32. Para mí, que soy principiante, esto fue un problema.

Aquí está el resultado de mis esfuerzos: un ejemplo simple de una interfaz web, en un servidor web que se ejecuta en ESP32Forth. El código está basado en el ejemplo de Peter Forth (peter-webpage-dht11-graphic-example.txt). El código completo se encuentra en el archivo adjunto **example_web.fs** .

El servidor web se ejecuta en la placa ESP32 con una conexión WiFi habilitada y responde a las solicitudes de los clientes (navegador en PC, móvil, etc.).

Por tanto, la interfaz web básica es sencilla:

```
: runpage  begin handleClient if serve-page 100 ms then 500 ms again
;
```

donde `handleClient` detecta si hay alguna solicitud del cliente, resuelve la solicitud y proporciona contenido HTML al cliente con la página de servicio de Word. El tiempo de espera `de ms` mejora la estabilidad de la conexión wifi en mi red doméstica.

```
: serve-page  ( --)  \ simple parsing and action of client respond
    path s" /" str= if
        htmlpagesend exit  \ exit leaves from serve-page
    then
    path s" /26/on" str= if
        cr ." ACTION for  /26/on " cr   \ here put action word
        0 to GPIO26 htmlpagesend exit
    then
    path s" /26/off" str=  if
        cr ." ACTION for /26/off " cr
        1 to GPIO26 htmlpagesend exit
    then
    path s" /27/on" str= if
        cr ." ACTION for /27/on " cr
        0 to GPIO27 htmlpagesend exit
    then
    path s" /27/off" str= if
```

```
        cr ." ACTION for /27/off " cr
        1 to GPIO27 htmlpagesend exit
    then
    path respond       \ actions for html forms
    htmlpagesend exit  \ resend html page
  ;
```

La palabra **serve-page** toma el texto de la solicitud del cliente de la ruta de la palabra en el formulario addr len y lo compara con las posibles respuestas del cliente, cada coincidencia activa la acción relevante y actualiza el contenido de la página HTML con la palabra **htmlpagesend** . Las palabras de acción se pueden colocar en lugar de sustitutos como está : **." ACTION for /26/on "**.

```
: htmlpagesend  \ send whole html page
    s" text/html" ok-response
    htmlpage    \ create html page in webintstream buffer
    webcontent send    \ and send it to client
  ;
```

La palabra **htmlpagesend** devuelve el primer código de estado y el tipo de datos HTML al cliente (navegador). Luego, el código de la página HTML se crea dinámicamente como texto y finalmente se envía al cliente para su visualización en el navegador.

En resumen, es todo un proceso.

Para uso práctico, me centré en tres tipos de información generada por la interfaz web ESP32:

- datos textuales pasivos simples, como los resultados de ciertas mediciones, por ejemplo de una estación meteorológica
- Botones para encendido/apagado para controlar algo mediante el circuito ESP32.
- Formularios HTML para ajustar ciertos parámetros en un programa que se ejecuta en ESP32.

Para ello creé este ejemplo simple de una página web generada en ESP32 con la dirección IP 92.168.1.6.

Figure 33: página web generada por ESP32forth

Así, el estado del texto GPIO 26 es: información textual y el valor 0 o 1 es el CUARTO valor para GPIO26 incluido en la página web al generar la página HTML.

Los botones GPIO26 y GPIO27 pueden cambiar el valor apropiado para controlar algo, por ejemplo, un relé conectado a los GPIO ESP32.

Otros formularios HTML pueden controlar ajustes más avanzados de la configuración del programa.

El último *Haga clic en mí para mostrar* ... solo genera información de fecha/hora real desde el navegador del cliente sin ninguna conexión programática al código ESP32forth.

script **example_web.fs** en el archivo **ESP32forth-book.zip** disponible aquí: https://github.com/MPETREMANN11/ESP32forth/blob/main/__documentation/ESP32forth-book.zip

Entonces sólo en breve:

Las líneas 8 a 29 crean la palabra de ayuda `mvbar` , utilizada como `mvbar` para cualquier texto de varias líneas | para crear una cadena temporal como addr len sobre más líneas de texto.

El búfer para el texto de la página HTML se crea en la línea 31 mediante `stream webintstream` y usa `word >stream` para agregar partes de texto.

La palabra `htmlpage` en las líneas 46 a 135 crea dinámicamente texto html después de cada activación. Las líneas 67 a 71 crean texto con los valores reales de FORTH GPIO26, GPIO27. Si se utiliza para mostrar continuamente ciertos valores medidos, es necesario colocar código para la actualización automática de la página HTML en el HTML generado.

Las líneas 72 a 88 crean botones de color rojo o verde basados en valores GPIO con información del cliente /26/on o /26/off para detección en la palabra de la página de servicio.

Las líneas 91 a 127 generan datos de formulario HTML para ajustar ciertos valores como datos, hora, texto o rango. Las líneas 128-131 solo generan información de fecha/hora real con código JavaScript.

Al final del código se habilita el servidor con wifi y se inicia la interfaz web como tarea en segundo plano.

Presento este código como base para experimentos. Estoy seguro de que se puede mejorar, los comentarios son bienvenidos.

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 7.0.7.17

FORTH

-	-rot	,	;	:	:noname	!
?	?do	?dup	.	."	.s	'
(local)	[	[']	[char]	[ELSE]	[IF]	[THEN]
]	{	{	}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	bg	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	fg	file-exists?	
FILE-POSITION		FILE-SIZE	fill	FIND	fliteral	FLN
FLOOR	flush	FLUSH-FILE	FMAX	FMIN	FNEGATE	FNIP
for	forget	FORTH	forth-builtins		FOVER	FP!
FP@	fp0	free	freq	FROT	FSIN	FSINCOS
FSQRT	FSWAP	fvariable	handler	here	hex	HIGH
hld	hold	httpd	I	if	IMMEDIATE	include
included	included?	INPUT	internals	invert	is	J
K	key	key?	L!	latestxt	leave	LED
ledc	list	literal	load	login	loop	LOW
ls	LSHIFT	max	MDNS.begin	min	mod	ms
MS-TICKS	mv	n.	needs	negate	nest-depth	next
nip	nl	NON-BLOCK	normal	octal	OF	ok
only	open-blocks	OPEN-DIR	OPEN-FILE	OR	order	OUTPUT
OVER	pad	page	PARSE	pause	PI	pin
pinMode	postpone	precision	previous	prompt	PSRAM?	pulseIn
quit	r"	R@	R/O	R/W	R>	r|
r~	rdrop	read-dir	READ-FILE	recurse	refill	registers
remaining	remember	RENAME-FILE	repeat	REPOSITION-FILE		required
reset	resize	RESIZE-FILE	restore	revive	RISC-V?	rm
rot	RP!	RP@	rp0	RSHIFT	rtos	s"
S>F	s>z	save	save-buffers		scr	SD
SD_MMC	sealed	see	Serial	set-precision		set-title
sf,	SF!	SF@	SFLOAT	SFLOAT+	SFLOATS	sign
SL@	sockets	SP!	SP@	sp0	space	spaces
SPIFFS	start-task	startswith?	startup:	state	str	str=
streams	structures	SW@	SWAP	task	tasks	telnetd
terminate	then	throw	thru	tib	to	tone
touch	transfer	transfer	type	u.	U/MOD	UL@
UNLOOP	until	update	use	used	UW@	value
VARIABLE	visual	vlist	vocabulary	W!	W/O	web-interface
webui	while	WiFi	Wire	words	WRITE-FILE	XOR
Xtensa?	z"	z>s				

asm

xtensa disasm disasm1 matchit address istep sextend m. m@ for-ops op >operands >mask >pattern >length >xt op-snap opcodes coden, names operand l o bits bit skip advance advance-operand reset reset-operand for-operands operands >printop >inop >next >opmask& bit! mask pattern length demask enmask >>1

odd? high-bit **end-code** code, code4, code3, code2, code1, callot chere reserve code-at code-start

bluetooth

SerialBT.new SerialBT.delete SerialBT.begin SerialBT.end SerialBT.available SerialBT.readBytes SerialBT.write SerialBT.flush SerialBT.hasClient SerialBT.enableSSP SerialBT.setPin SerialBT.unpairDevice SerialBT.connect SerialBT.connectAddr SerialBT.disconnect SerialBT.connected SerialBT.isReady bluetooth-builtins

editor

a r d e wipe p n l

ESP

getHeapSize getFreeHeap getMaxAllocHeap getChipModel getChipCores getFlashChipSize getCpuFreqMHz getSketchSize deepSleep getEfuseMac esp_log_level_set ESP-builtins

httpd

notfound-response bad-response ok-response response send path method **hasHeader** handleClient read-headers completed? body content-length header crnl= eat skipover skipto in@<> end< **goal#** **goal** strcase= upper server client-cr client-emit client-read client-type **client-len** client httpd-port clientfd **sockfd** body-read body-1st-read body-chunk body-chunk-size **chunk-filled** chunk **chunk-size** **max-connections**

insides

run normal-mode raw-mode step ground handle-key quit-edit save load backspace delete handle-esc insert update crtype cremit ndown down nup up caret length capacity text start-size fileh filename# filename max-path

internals

assembler-source xtensa-assembler-source MALLOC SYSFREE REALLOC heap_caps_malloc heap_caps_free heap_caps_realloc heap_caps_get_total_size heap_caps_get_free_size heap_caps_get_minimum_free_size heap_caps_get_largest_free_block RAW-YIELD RAW-TERMINATE READDIR CALLCODE CALL0 CALL1 CALL2 CALL3 CALL4 CALL5 CALL6
CALL7 CALL8 CALL9 CALL10 CALL11 CALL12 CALL13 CALL14 CALL15 DOFLIT S>FLOAT? fill32 'heap 'context 'latestxt 'notfound 'heap-start 'heap-size 'stack-cells 'boot 'boot-size **'tib** 'argc 'argv 'runner 'throw-handler NOP BRANCH 0BRANCH DONEXT DOLIT DOSET DOCOL DOCON DOVAR DOCREATE DODOES ALITERAL LONG-SIZE S>NUMBER? 'SYS YIELD EVALUATE1 'builtins internals-builtins autoexec
arduino-remember-filename
arduino-default-use esp32-stats serial-key? serial-key serial-type yield-task yield-step e' @line grow-blocks use?! common-default-use block-data **block-dirty** clobber clobber-line include+ path-join **included-files** raw-included include-file sourcedirname sourcefilename! sourcefilename **sourcefilename#** **sourcefilename&**

starts../ starts./ dirname ends/ default-remember-filename remember-filename
restore-name save-name forth-wordlist setup-saving-base 'cold park-forth
park-heap saving-base crtype cremit cases (+to) (to) --? }? ?room scope-create
do-local scope-clear scope-exit local-op scope-depth local+! local! local@
<>locals locals-here locals-area locals-gap locals-capacity ?ins. ins.
vins. onlines line-pos line-width size-all size-vocabulary vocs. voc. voclist
voclist-from see-all >vocnext see-vocabulary nonvoc? see-xt ?see-flags
see-loop see-one indent+! icr see. indent mem= ARGS_MARK -TAB +TAB NONAMED
BUILTIN_FORK SMUDGE IMMEDIATE_MARK relinquish dump-line ca@ cell-shift
cell-base cell-mask MALLOC_CAP_RTCRAM MALLOC_CAP_RETENTION MALLOC_CAP_IRAM_8BIT
MALLOC_CAP_DEFAULT MALLOC_CAP_INTERNAL MALLOC_CAP_SPIRAM MALLOC_CAP_DMA
MALLOC_CAP_8BIT MALLOC_CAP_32BIT MALLOC_CAP_EXEC #f+s internalized BUILTIN_MARK
zplace $place free. boot-prompt raw-ok [SKIP]' [SKIP] ?stack sp-limit input-limit
tib-setup raw.s $@ digit parse-quote leaving, leaving)leaving leaving(
value-bind evaluate&fill evaluate-buffer arrow ?arrow. ?echo input-buffer
immediate? eat-till-cr wascr *emit *key notfound last-vocabulary voc-stack-end
xt-transfer xt-hide xt-find& scope

interrupts

pinchange #GPIO_INTR_HIGH_LEVEL #GPIO_INTR_LOW_LEVEL #GPIO_INTR_ANYEDGE
#GPIO_INTR_NEGEDGE #GPIO_INTR_POSEDGE #GPIO_INTR_DISABLE ESP_INTR_FLAG_INTRDISABLED
ESP_INTR_FLAG_IRAM ESP_INTR_FLAG_EDGE ESP_INTR_FLAG_SHARED ESP_INTR_FLAG_NMI
ESP_INTR_FLAG_LEVELn ESP_INTR_FLAG_DEFAULT gpio_config gpio_reset_pin gpio_set_intr_type
gpio_intr_enable gpio_intr_disable gpio_set_level gpio_get_level gpio_set_direction
gpio_set_pull_mode gpio_wakeup_enable gpio_wakeup_disable gpio_pullup_en
gpio_pulldown_en gpio_pulldown_dis gpio_hold_en gpio_hold_dis
gpio_deep_sleep_hold_en gpio_deep_sleep_hold_dis gpio_install_isr_service
gpio_isr_handler_add gpio_isr_handler_remove
gpio_set_drive_capability gpio_get_drive_capability esp_intr_alloc esp_intr_free
interrupts-builtins

ledc

ledcSetup ledcAttachPin ledcDetachPin ledcRead ledcReadFreq ledcWrite ledcWriteTone
ledcWriteNote ledc-builtins

oled

OledInit SSD1306_SWITCHCAPVCC SSD1306_EXTERNALVCC WHITE BLACK
OledReset HEIGHT WIDTH OledAddr OledNew OledDelete OledBegin OledHOME OledCLS
OledTextc OledPrintln OledNumln OledNum OledDisplay OledPrint OledInvert
OledTextsize OledSetCursor OledPixel OledDrawL OledFastHLine OledFastVLine
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

rmt

rmt_set_clk_div rmt_get_clk_div rmt_set_rx_idle_thresh rmt_get_rx_idle_thresh rmt_set_mem_block_num rmt_get_mem_block_num rmt_set_tx_carrier rmt_set_mem_pd rmt_get_mem_pd rmt_tx_start rmt_tx_stop rmt_rx_start rmt_rx_stop rmt_tx_memory_reset rmt_rx_memory_reset rmt_set_memory_owner rmt_get_memory_owner rmt_set_tx_loop_mode rmt_get_tx_loop_mode rmt_set_rx_filter rmt_set_source_clk rmt_get_source_clk rmt_set_idle_level rmt_get_idle_level rmt_get_status rmt_set_rx_intr_en rmt_set_err_intr_en rmt_set_tx_intr_en rmt_set_tx_thr_intr_en rmt_set_gpio rmt_config rmt_isr_register rmt_isr_deregister rmt_fill_tx_items rmt_driver_install rmt_driver_uinstall rmt_get_channel_status rmt_get_counter_clock rmt_write_items rmt_wait_tx_done rmt_get_ringbuf_handle rmt_translator_init rmt_translator_set_context rmt_translator_get_context rmt_write_sample rmt-builtins

rtos

vTaskDelete xTaskCreatePinnedToCore xPortGetCoreID rtos-builtins

SD

SD.begin SD.beginFull SD.beginDefaults SD.end SD.cardType SD.totalBytes SD.usedBytes SD-builtins

SD_MMC

SD_MMC.begin SD_MMC.beginFull SD_MMC.beginDefaults SD_MMC.end SD_MMC.cardType SD_MMC.totalBytes SD_MMC.usedBytes SD_MMC-builtins

Serial

Serial.begin Serial.end Serial.available Serial.readBytes Serial.write Serial.flush Serial.setDebugOutput Serial2.begin Serial2.end Serial2.available Serial2.readBytes Serial2.write Serial2.flush Serial2.setDebugOutput serial-builtins

sockets

ip. ip# ->h_addr ->addr! ->addr@ ->port! ->port@ sockaddr l, s, bs, SO_REUSEADDR SOL_SOCKET sizeof(sockaddr_in) AF_INET SOCK_RAW SOCK_DGRAM SOCK_STREAM socket setsockopt bind listen connect sockaccept select poll send sendto sendmsg recv recvfrom recvmsg 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**

timers

interval onalarm int-enable! alarm-enable! divider! autoreload! increase!
enable! alarm! alarm@ timer! timer@ tmp t>nx timer_isr_callback_add timer_init_null
timer_get_counter_value timer_set_counter_value timer_start timer_pause
timer_set_counter_mode timer_set_auto_reload timer_set_divider
timer_set_alarm_value
timer_get_alarm_value timer_set_alarm timer_group_intr_enable
timer_group_intr_disable
timer_enable_intr timer_disable_intr timers-builtins

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 MUL.AD.HH, MUL.AD.LH, MUL.AD.HL,
MUL.AD.LL, MUL.AD MUL.AA.HH, MUL.AA.LH, MUL.AA.HL, MUL.AA.LL, MUL.AA MOVT,
MOVSP, MOVT.S, MOVF.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 MOVF, MOVGEZ, MOVLTZ, MOVNEZ,
MOVEQZ, MAXU, MINU, MAX, MIN, CONDOP MOV, LSXU, LSX, L32E, LICW, LICT,
LDCT, JX, IITLB, IDTLB, LSIU, LSI, LDINC, LDDEC, L32R, EXTUI, S32E, S32RI,
S32C1I, ADDMI, ADDI, L32AI, L16SI, S32I, S16I, S8I, L32I, L16UI, L8UI,
LDSTORE MOVI, IIU, IHU, IPFL, DIWBI, DIWB, DIU, DHU, DPFL, CACHING2 III,
IHI, IPF, DII, DHI, DHWBI, DHWB, DPFWO, DPFRO, 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 cofs cofs. >cofs offset18 offset12 offset8 ofs18 ofs12
ofs8 ofs18. ofs12. ofs8. >ofs sr imm16 imm8 imm4 im numeric register reg.
nop a15 a14 a13 a12 a11 a10 a9 a8 a7 a6 a5 a4 a3 a2 a1 a0
```

Anexo A – Resumen de registros

.....

GPIO registers

Name	Description	Address	Access
GPIO_OUT_REG	GPIO 0-31 output register	$3FF44004	R/W
GPIO_OUT_W1TS_REG	GPIO 0-31 output register_W1TS	$3FF44008	WO
GPIO_OUT_W1TC_REG	GPIO 0-31 output register_W1TC	$3FF4400C	WO
GPIO_OUT1_REG GPIO	GPIO 32-39 output register	$3FF44010	R/W
GPIO_OUT1_W1TS_REG	GPIO 32-39 output bit set register	$3FF44014	WO
GPIO_OUT1_W1TC_REG	GPIO 32-39 output bit clear register	$3FF44018	WO
GPIO_ENABLE_REG	GPIO 0-31 output enable register	$3FF44020	R/W
GPIO_ENABLE_W1TS_REG	GPIO 0-31 output enable register_W1TS	$3FF44024	WO
GPIO_ENABLE_W1TC_REG	GPIO 0-31 output enable register_W1TC	$3FF44028	WO
GPIO_ENABLE1_REG	GPIO 32-39 output enable register	$3FF4402C	R/W
GPIO_ENABLE1_W1TS_REG	GPIO 32-39 output enable bit set register	$3FF44030	WO
GPIO_ENABLE1_W1TC_REG	GPIO 32-39 output enable bit clear register	$3FF44034	WO
GPIO_STRAP_REG	Bootstrap pin value register	$3FF44038	RO
GPIO_IN_REG	GPIO 0-31 input register	$3FF4403C	RO
GPIO_IN1_REG	GPIO 32-39 input register	$3FF44040	RO
GPIO_STATUS_REG	GPIO 0-31 interrupt status register	$3FF44044	R/W
GPIO_STATUS_W1TS_REG	GPIO 0-31 interrupt status register_W1TS	$3FF44048	WO
GPIO_STATUS_W1TC_REG	GPIO 0-31 interrupt status register_W1TC	$3FF4404C	WO
GPIO_STATUS1_REG	GPIO 32-39 interrupt status register1	$3FF44050	R/W
GPIO_STATUS1_W1TS_REG	GPIO 32-39 interrupt status bit set register	$3FF44054	WO
GPIO_STATUS1_W1TC_REG	GPIO 32-39 interrupt status bit clear register	$3FF44058	WO
GPIO_ACPU_INT_REG	GPIO 0-31 APP_CPU interrupt status	$3FF44060	RO
GPIO_ACPU_NMI_INT_REG	GPIO 0-31 APP_CPU non-maskable interrupt status	$3FF44064	RO
GPIO_PCPU_INT_REG	GPIO 0-31 PRO_CPU interrupt status	$3FF44068	RO
GPIO_PCPU_NMI_INT_REG	GPIO 0-31 PRO_CPU non-maskable interrupt status	$3FF4406C	RO
GPIO_ACPU_INT1_REG	GPIO 32-39 APP_CPU interrupt status	$3FF44074	RO
GPIO_ACPU_NMI_INT1_REG	GPIO 32-39 APP_CPU non-maskable interrupt status	$3FF44078	RO
GPIO_PCPU_INT1_REG	GPIO 32-39 PRO_CPU interrupt status	$3FF4407C	RO
GPIO_PCPU_NMI_INT1_REG	GPIO 32-39 PRO_CPU non-maskable interrupt status	$3FF44080	RO
GPIO_PIN0_REG	Configuration for GPIO pin 0	$3FF44088	R/W
GPIO_PIN1_REG	Configuration for GPIO pin 1	$3FF4408C	R/W
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

Name	Description	Address	Access
GPIO_FUNC254_IN_SEL_CFG_REG	Peripheral function 254 input selection register	$3FF44528	R/W
GPIO_FUNC255_IN_SEL_CFG_REG	Peripheral function 255 input selection register	$3FF4452C	R/W
GPIO_FUNC0_OUT_SEL_CFG_REG	Peripheral output selection for GPIO 0	$3FF44530	R/W
GPIO_FUNC1_OUT_SEL_CFG_REG	Peripheral output selection for GPIO 1	$3FF44534	R/W
GPIO_FUNC38_OUT_SEL_CFG_REG	Peripheral output selection for GPIO 38	$3FF445C8	R/W
GPIO_FUNC39_OUT_SEL_CFG_REG	Peripheral output selection for GPIO 39	$3FF445CC	R/W
IO_MUX_PIN_CTRL	Clock output configuration register	$3FF49000	R/W
IO_MUX_GPIO36_REG	Configuration register for pad GPIO36	$3FF49004	R/W
IO_MUX_GPIO37_REG	Configuration register for pad GPIO37	$3FF49008	R/W
IO_MUX_GPIO38_REG	Configuration register for pad GPIO38	$3FF4900C	R/W
IO_MUX_GPIO39_REG	Configuration register for pad GPIO39	$3FF49010	R/W
IO_MUX_GPIO34_REG	Configuration register for pad GPIO34	$3FF49014	R/W
IO_MUX_GPIO35_REG	Configuration register for pad GPIO35	$3FF49018	R/W
IO_MUX_GPIO32_REG	Configuration register for pad GPIO32	$3FF4901C	R/W
IO_MUX_GPIO33_REG	Configuration register for pad GPIO33	$3FF49020	R/W
IO_MUX_GPIO25_REG	Configuration register for pad GPIO25	$3FF49024	R/W
IO_MUX_GPIO26_REG	Configuration register for pad GPIO26	$3FF49028	R/W
IO_MUX_GPIO27_REG	Configuration register for pad GPIO27	$3FF4902C	R/W
IO_MUX_MTMS_REG	Configuration register for pad MTMS	$3FF49030	R/W
IO_MUX_MTDI_REG	Configuration register for pad MTDI	$3FF49034	R/W
IO_MUX_MTCK_REG	Configuration register for pad MTCK	$3FF49038	R/W
IO_MUX_MTDO_REG	Configuration register for pad MTDO	$3FF4903C	R/W
IO_MUX_GPIO2_REG	Configuration register for pad GPIO2	$3FF49040	R/W
IO_MUX_GPIO0_REG	Configuration register for pad GPIO0	$3FF49044	R/W
IO_MUX_GPIO4_REG	Configuration register for pad GPIO4	$3FF49048	R/W
IO_MUX_GPIO16_REG	Configuration register for pad GPIO16	$3FF4904C	R/W
IO_MUX_GPIO17_REG	Configuration register for pad GPIO17	$3FF49050	R/W
IO_MUX_SD_DATA2_REG	Configuration register for pad SD_DATA2	$3FF49054	R/W
IO_MUX_SD_DATA3_REG	Configuration register for pad SD_DATA3	$3FF49058	R/W
IO_MUX_SD_CMD_REG	Configuration register for pad SD_CMD	$3FF4905C	R/W
IO_MUX_SD_CLK_REG	Configuration register for pad SD_CLK	$3FF49060	R/W
IO_MUX_SD_DATA0_REG	Configuration register for pad SD_DATA0	$3FF49064	R/W
IO_MUX_SD_DATA1_REG	Configuration register for pad SD_DATA1	$3FF49068	R/W
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

Name	Description	Address	Access
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
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 ESp32forth
  https://github.com/Esp32forth-org
-

índice léxico