

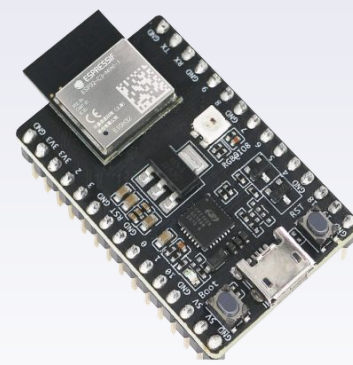
Unidad 2.0

Debugging

► Computadoras vs microcontroladores



Una **computadora de propósitos generales** contiene los 3 bloques funcionales (**CPU, MEM, E/S**). Generalmente tiene buses de expansión para E/S (USB, PCI, ETC), puertos para memoria secundaria (sata, M.2, ETC), y zócalos para **expandir la memoria principal** (SDRAM). Suele poseer un **S.O.** que permite cambiar el programa (aplicación) según la necesidad.



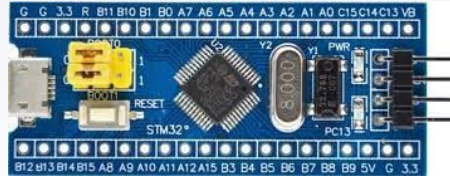
Un **microcontrolador** también contiene los 3 bloques funcionales pero integrados en un **único chip**. Suele tener periféricos de E/S que permiten conectar dispositivos (GPIO, UART, SPI, I2C, ETC). **La memoria principal generalmente NO se puede expandir** y la memoria secundaria (FLASH) tampoco. Suelen programarse **Bare-Metal** (sin S.O.) o con un RTOS específico. No suele cambiar la aplicación o el programa (salvo updates).

Microcontroladores populares



Arduino

- ▶ Basado en ATmega^{*}
- ▶ 8 Bits AVR (original)
- ▶ Biblioteca comunitaria muy extensa
- ▶ No soporta Debugger (excepto cuando se programa en C)
- ▶ Corre en casi cualquier lado



ST (ej: stm32f103 bluepill)

- ▶ Basado en Cortex ARM
- ▶ 32 bits ARM
- ▶ ST HAL
- ▶ Programador y debugger ST LinkV2 utilizando protocolo SWD
- ▶ Popular en China (pirateado)



ESP32

- ▶ MIPS(sX) o RISC-V (c3)
- ▶ 32 bits MIPS o RISC-V
- ▶ ESP-IDF
- ▶ Debugger JTAG, en el c3/s3 está incorporado en el chip.
- ▶ Tiene radio Wifi y BT
- ▶ OTA (over the air) updates

Comparativa



Arduino Uno Ch340 C/cable Usb Compatible

\$ 4.340

Llega mañana

2KB RAM
32KB Flash
u\$8,68
USA: ~u\$8.



Blue Pill Stm32f103c8t6 Modulo Stm32 Para Desarrollo Arduino

\$ 4.751

Llega mañana

20KB RAM
128KB Flash
u\$9,52
USA: u\$3.



MAS VENDIDO

Programador Usb St-link V2 Stm32 Usb Con Cable

\$ 2.858

Llega mañana

u\$5,71
USA: u\$7.



Programador
\$ 6.580



Nodemcu Esp32 Wroom 32d Wifi + Bluetooth V4 Arduino Devkit C

\$ 3.850

Llega mañana **FULL**

512KB RAM
4MB Flash
u\$7,7 + 13,16
USA: ~u\$6 + 8



Placa Desarrollo Espressif Esp32 C3 Dev Kit 02 Wroom

\$ 14.339

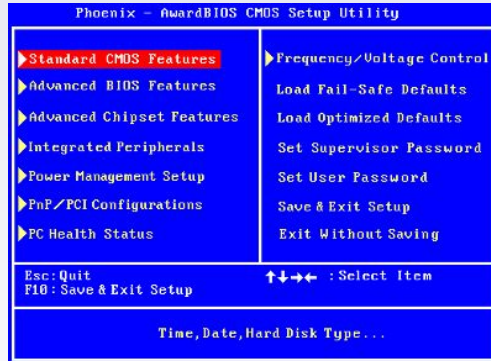
Envío gratis

400KB RAM
4MB Flash
u\$28,67
USA: ~u\$6.

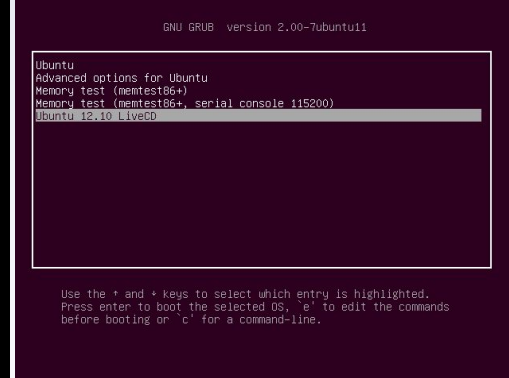
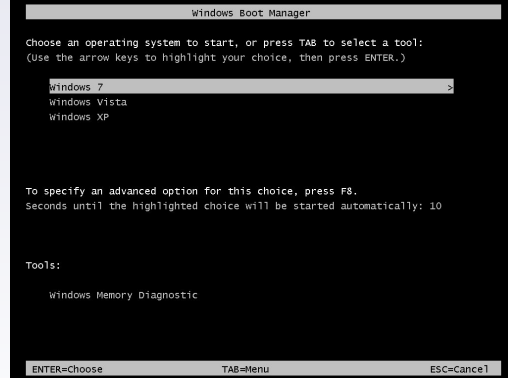




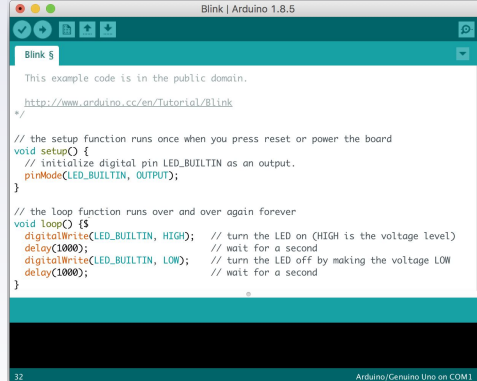
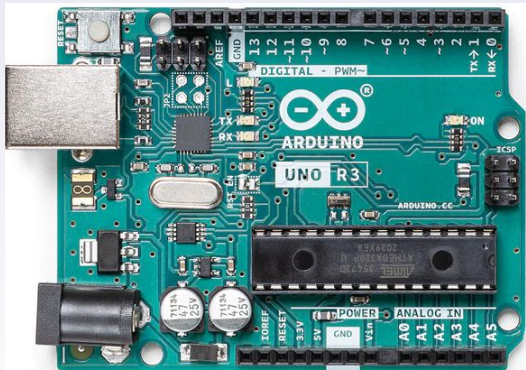
Encendido



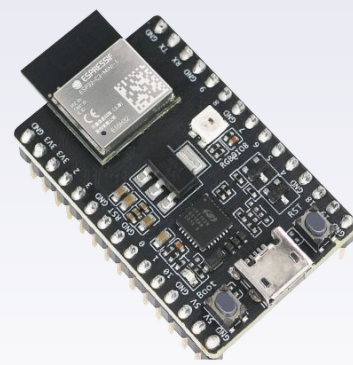
Cuando se enciende una computadora (x86) el CS:IP se carga con FFFF:0000 (0xFFFF0). Existe una **memoria ROM** (hoy en día flash) que contiene un **programa inicial (BIOS)**. Este BIOS es configurable (mediante un programa con GUI) y almacena datos (como fecha/hora, dispositivo principal de boot, overclocking, password, etc) en una memoria NVRAM (respaldada con una batería) que suele poder limpiarse poniendo en corto un jumper. El BIOS se encarga de leer el **sector 0 (512 bytes)** del dispositivo de booteo configurado, y si el mismo termina con la firma **0x55AA** asume que es un programa de booteo (bootloader) y lo carga en RAM y ejecuta. En máquinas **UEFI** la secuencia es similar, pero en vez de buscar el sector 0 busca una **partición de tipo FAT32 con flags esp y boot en 1 (GPT)**, y dentro de esa partición busca el archivo **EFI/BOOT/BOOTX64.EFI** (o similar) y lo ejecuta.



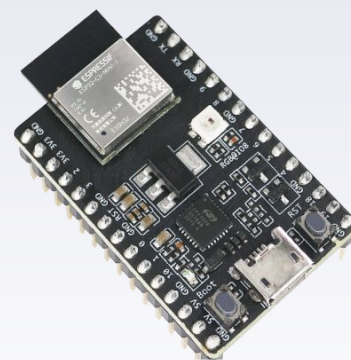
El programa de booteo (**bootloader**) suele ser de tamaño muy reducido, y su función es **encontrar al sistema operativo** en el disco donde esté instalado y **cargarlo en memoria** para que funcione. En el caso de Windows se usa el NTLDR (**NT Loader**) o BOOTMGR (actualmente). En el caso de Linux el más común es GRUB (**GR**and **U**nified **B**ootloader) sobre una PC x86, pero si corre desde un pendrive o ISO suele usarse isolinux o syslinux como bootloader. Dado que Linux puede correr en múltiples dispositivos existe U-Boot que permite cargar el kernel de Linux en memoria en diversas arquitecturas (x86, RISC-V, MIPS, ARM, etc). Una vez que el kernel que cargó en memoria principal, comenzará con su secuencia de inicialización (Init, Systemd, etc) detectando dispositivos, cargando sus respectivos drivers e inicializando los mismos. Luego entrega el control al usuario.



Cada fabricante de microcontrolador define la secuencia de encendido. En el caso de arduino, su micro ATmega328 particiona la flash en **memoria de aplicación y memoria de bootloader**. Luego puede mediante unos fusibles indicar si durante el inicio comienza a ejecutar la dirección 0000, o la dirección de inicio del bootloader. Arduino escribe un programa **bootloader que toma control del puerto serie** (UART). El IDE de arduino utiliza el puerto serie (COM en windows) para comunicarse con el bootloader. La línea DTR se usa para resetear el ATmega desde la PC y forzar al bootloader. Durante un pequeño tiempo el bootloader espera comandos de la PC. Si no recibe ninguno, ejecuta el programa en la flash de aplicación. Si recibe el comando de escritura, recibe de la PC el nuevo programa, lo escribe en la flash de aplicación y luego lo ejecuta. Si el ATmega no tiene el bootloader, debe escribirse el mismo usando los pines de ICSP. **Arduino NO tiene debugger, ATmega si.** 9



El ESP32**C3** se enciende y lee el nivel de tensión en ciertos pines. Dependiendo de esto puede arrancar en **modo Download** (donde recibe el programa a grabar en flash usando el puerto serie/JTAG USB), o puede arrancar en **modo SPI Flash**, donde ejecuta un bootloader en ROM que se encarga de cargar en RAM un segundo bootloader almacenado en Flash en la dirección 0x1000. Este segundo bootloader se encarga de **leer una tabla de particiones en Flash y elegir que aplicación correr**. La idea es poder hacer actualizaciones OTA (Over The Air), por ende en Flash puede haber más de una versión de la aplicación, y el bootloader elige cual de ellas correr. En el caso de actualizar el firmware, se almacena la nueva versión en espacio libre en la flash (borrando versiones viejas previamente) y en el reset se ejecuta la versión nueva. Si esto falla, el bootloader inicial puede volver a la versión original. Todo esto está documentado por Espressif en el manual de OTA.



En un booteo normal, el segundo bootloader va a terminar luego de cargar la aplicación en RAM y llama a la función **app_main()**. Esta secuencia está definida en la **ESP-IDF**.

Con el fin de darle más valor agregado a sus productos los fabricantes de microcontroladores suelen incluir bibliotecas con código para interactuar con los distintos periféricos que componen un microcontrolador. En el caso de ST con su línea ARM esto se conoce como HAL (Hardware Abstraction Layer), en el caso de Arduino son un conjunto de bibliotecas desarrolladas por la comunidad y en el caso de Espressif se brinda ESP-IDF.

► GCC y GDB

```

1  #include <stdio.h>
2
3  int funcion(int var1,int var2){
4      int aux=0;
5      while (var2>0){
6          aux = aux + var1;
7          var2--;
8      }
9      return aux;
10 }
11
12 int main(){
13     int a = 10;
14     int b;
15     int c=0;
16     printf("Ingrese un valor:");
17     scanf("%d",&b);
18     c=funcion(a,b);
19     printf("El valor es: %d\n",c);
20     return 0;
21 }
22
23

```

Compilamos con:

```
gcc -g -o programa main.c
```

-g indica que tiene que incluir los "debug symbols" , -o indica el nombre que toma el ejecutable.

Verificamos con: **file programa**

```

programa: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked
, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=d5c685f1ba1b7e28453d2b825e33
0e8149f203d9, for GNU/Linux 3.2.0, with debug_info, not stripped

```

Usamos gdbtui (GDB con Terminal User Interface): **gdbtui programa**

```

main.c
3  int funcion(int var1,int var2){
4      int aux=0;
5      while (var2>0){
6          aux = aux + var1;
7          var2--;
8      }
9      return aux;
10 }
11
12 int main(){
13     int a = 10;
14     int b;
15     int c=0;
16     printf("Ingrese un valor:");
17     scanf("%d",&b);
18     c=funcion(a,b);
19     printf("El valor es: %d\n",c);
20     return 0;
21 }
22 }
23

```

exec No process in: L?? PC: ??
(gdb)

Comandos GDB

- ▶ **b** xx (inserta un breakpoint)
- ▶ **r** (run)
- ▶ **ctrl+X 2** (cambia particiones)
- ▶ **layout src** (muestra source)
- ▶ **layout asm** (muestra asm)
- ▶ **layout regs** (muestra registros)
- ▶ **si** (step instruction)
- ▶ **refresh** (redibuja la pantalla)
- ▶ **s** (step)
- ▶ **n** (next)
- ▶ **c** (continue)
- ▶ **print** xxxx (imprime valor)
- ▶ **watch** xxxx (reporta cambios)
- ▶ **info** breakpoints (lista de bp)
- ▶ **Info** watch (lista de watches)
- ▶ **disable** N (apaga breakpoint)
- ▶ **d** N (delete breakpoint)
- ▶ **condition N var == X** (br var==X)
- ▶ **finish** (busca el return)
- ▶ **bt** (imprime back trace)
- ▶ **x** \$pc (imprime *pc = mem)
- ▶ **q** (termina gdb)

Breakpoint HW SW

Para insertar un **breakpoint o watchpoint** (variable) se utiliza un **registro donde se almacena una dirección** contra la cual se valida una condición. Estas condiciones dependen de la arquitectura pero en su mayoría todas soportan detectar:

- **Ejecución de instrucción en la dirección**
- **Lectura o Escritura de datos en la dirección**

La cantidad de estos registros suele ser limitada. En x86 existen solo 6. En algunas arquitecturas solo 2. En RISC-V depende de la microarquitectura (se recomienda como mínimo 4 hasta 4096). **Estos son conocidos como breakpoints HW.** El ESP32c3 soporta 8. El SiFive FE310 soporta solo 2.

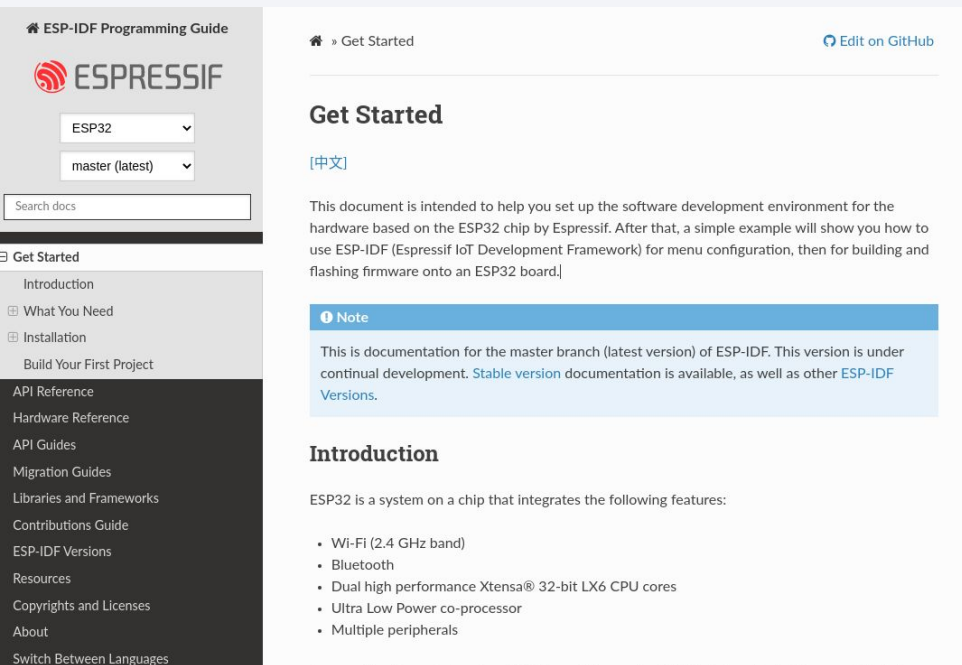
Dado que un programa ejecuta desde memoria principal (al menos en x86), los debuggers generan **breakpoints SW**. Esto se logra **reemplazando instrucciones del programa por instrucciones de tipo break**. Estos breaks **detienen la ejecución y el debugger puede entonces introducir la instrucción original**. De esta forma el debugger se convierte en una especie de intérprete.

Si el programa ejecuta directamente desde Flash (con QSPI Flash esto es posible), entonces estos **breakpoints SW son imposibles**.

<https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/index.html>

<https://github.com/espressif/idf-eclipse-plugin/blob/master/docs/Espressif-IDE.md>

ESP-IDF IDE



ESP-IDF Programming Guide

ESPRESSIF

ESP32

master (latest)

Search docs

Get Started

Introduction

What You Need

Installation

Build Your First Project

API Reference

Hardware Reference

API Guides

Migration Guides

Libraries and Frameworks

Contributions Guide

ESP-IDF Versions

Resources

Copyrights and Licenses

About

Switch Between Languages

» Get Started

Edit on GitHub

Get Started

[中文]

This document is intended to help you set up the software development environment for the hardware based on the ESP32 chip by Espressif. After that, a simple example will show you how to use ESP-IDF (Espressif IoT Development Framework) for menu configuration, then for building and flashing firmware onto an ESP32 board.

Note

This is documentation for the master branch (latest version) of ESP-IDF. This version is under continual development. [Stable version](#) documentation is available, as well as other [ESP-IDF Versions](#).

Introduction

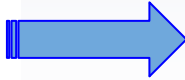
ESP32 is a system on a chip that integrates the following features:

- Wi-Fi (2.4 GHz band)
- Bluetooth
- Dual high performance Xtensa® 32-bit LX6 CPU cores
- Ultra Low Power co-processor
- Multiple peripherals

- **Consola (command line)**
 - 100% automatizable
 - Muy poco amigable
 - Multiplataforma
 - Guiado por variables de entorno
- **ESP IDE (eclipse)**
 - Muy Amigable
 - Automatización por cmd line
 - Multiplataforma
- VSCode (plugin)
 - Muy Amigable
 - Automatización por cmd line
 - Multiplataforma

Alto nivel

```
1 int main(){
2   int a=8;
3   int b=10;
4   int c=0;
5
6   while (a>0){
7     c=c+b;
8     a--;
9   }
10
11   return c;
12 }
```



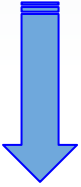
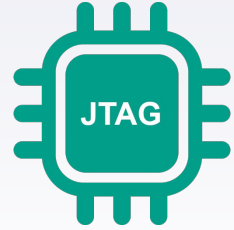
Bajo nivel

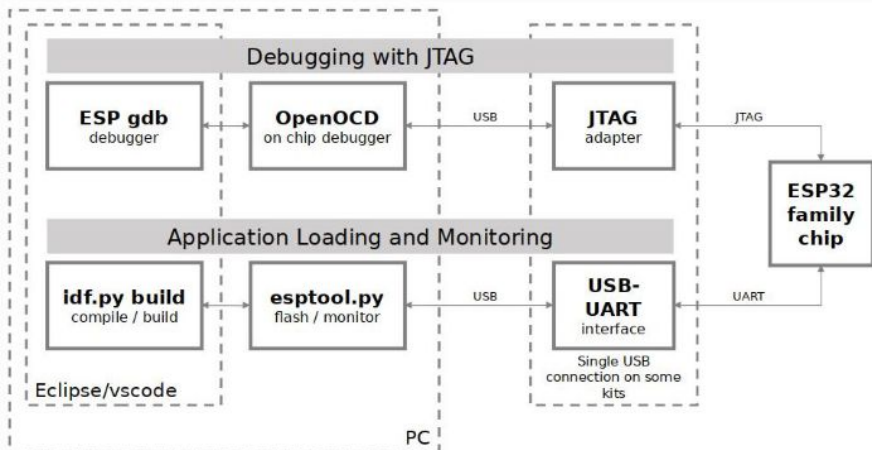
```
addi x2 x2 -32
sw x8 28 x2
addi x8 x2 32
addi x15 x0 8
sw x15 -20 x8
addi x15 x0 10
sw x15 -28 x8
sw x0 -24 x8
jal x0 32
lw x14 -24 x8
lw x15 -28 x8
add x15 x14 x15
sw x15 -24 x8
lw x15 -20 x8
addi x15 x15 -1
sw x15 -20 x8
lw x15 -20 x8
blt x0 x15 -32
lw x15 -24 x8
addi x10 x15 0
lw x8 28 x2
addi x2 x2 32
jalr x0 x1 0
```



Máquina

10074:	fe010113
10078:	00812e23
1007c:	02010413
10080:	00800793
10084:	fef42623
10088:	00a00793
1008c:	fef42223
10090:	fe042423
10094:	0200006f
10098:	fe842703
1009c:	fe442783
100a0:	00f707b3
100a4:	fef42423
100a8:	fec42783
100ac:	fff78793
100b0:	fef42623
100b4:	fec42783
100b8:	fef040e3
100bc:	fe842783
100c0:	00078513
100c4:	01c12403
100c8:	02010113
100cc:	00008067





ADC1_CH4
ADC2_CH0

MTMS
MTDI
MTCK
MTDO

FSPIHD
FSPIWP
FSPICLK
FSPID

GPIO4
GPIO5
GPIO6
GPIO7

GPIO8
GPIO9
5V
5V
GND

GPIO10
GPIO20
GPIO21
GPIO18
GPIO19
GND
GND

FSPICS0
U0RXD
U0TXD

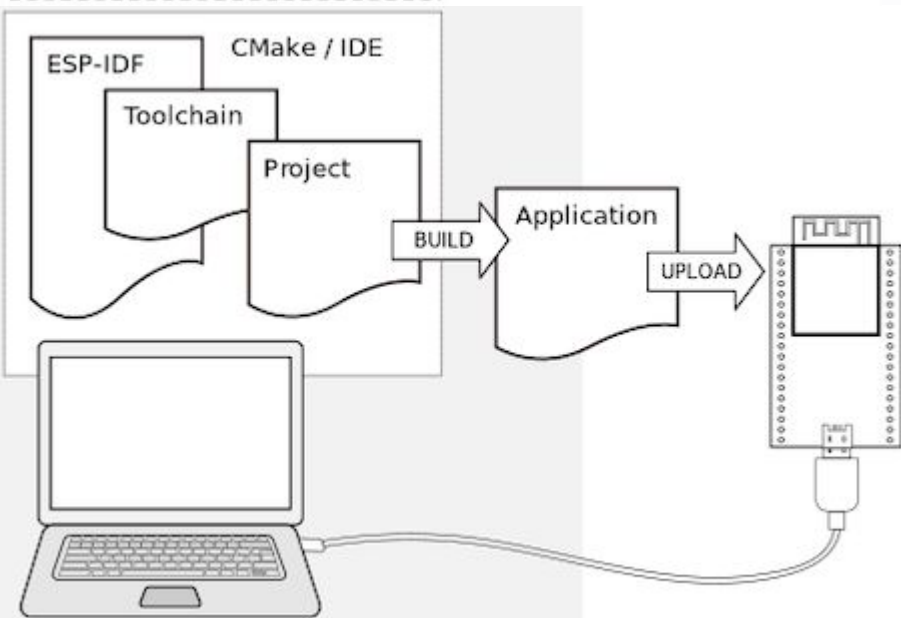
GPIO0
GPIO1
GPIO2
GPIO3
GND
GND
GND
GND
GND
GND
GND
GND

XTAL_32K_P
XTAL_32K_N
FSPIQ
ADC1_CH0
ADC1_CH1
ADC1_CH2
ADC1_CH3

ESPRESSIF
ESP32-C3-WROOM-02

ESP32-C3-DevKitM-1 V1.6
www.espressif.com

VBUS
GND



D-
D+

VBUS
GND

VBUS
GND

VBUS
GND

VBUS
GND

VBUS
GND

VBUS
GND

VBUS
GND

VBUS
GND

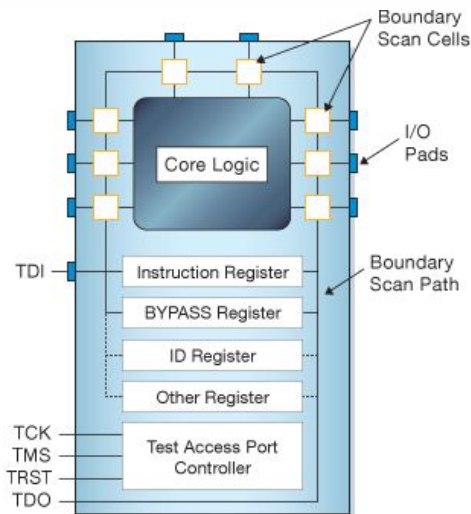
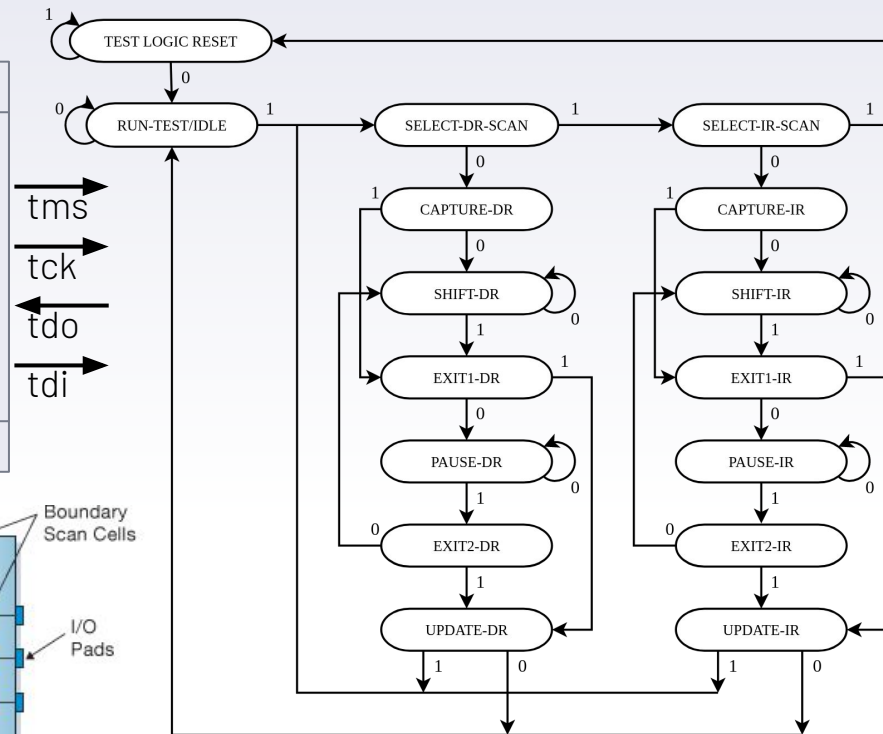
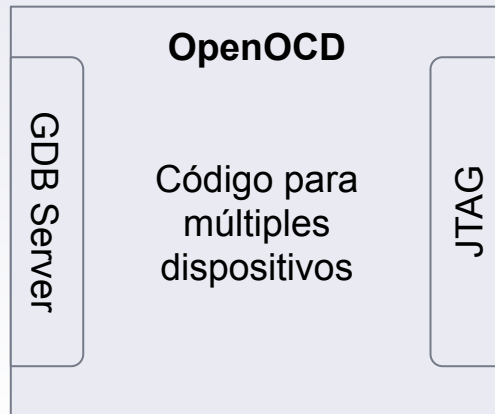
VBUS
GND

VBUS
GND

VBUS
GND

VBUS
GND

VBUS
GND



OpenOCD

Pasos para instalar ESP-IDF (Linux / Mac)

<https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/linux-macos-setup.html>

1. Instalar dependencias
2. **./install.sh --enable-gdbgui all**
3. Una vez instalado el toolchain completo esta en ~/esp , agregar los paths y variables de entorno a la terminal con:
../export.sh

Una vez instalado...

1. Crear Proyecto: **idf.py create-project NOMBRE**
2. Definir chip: **idf.py set-target esp32c3**
3. Configurar componentes: **idf.py menuconfig (conviene apagar el watchdog)**
4. Compilar: **idf.py build**
5. Escribir Flash: **idf.py flash**
6. Monitorear: **idf.py monitor**

```
(Top) → Component config → ESP System Settings
Espressif IoT Development Framework Configuration
CPU frequency (160 MHz) --->
Panic handler behaviour (Print registers and reboot) --->
(0) Panic reboot delay (Seconds)
[*] Enable RTC fast memory for dynamic allocations
[ ] Generate and use eh_frame for backtracing
Memory protection --->
(32) System event queue size
(2304) Event loop task stack size
(3584) Main task stack size
Main task core affinity (CPU0) --->
(2048) Minimal allowed size for shared stack
Channel for console output (Default: UART0) --->
Channel for console secondary output (USB_SERIAL_JTAG PORT) --->
[ ] Interrupt watchdog
[ ] Enable Task Watchdog Timer
[ ] Place panic handler code in IRAM
[ ] OpenOCD debug stubs
[*] Make exception and panic handlers JTAG/OCD aware
Interrupt level to use for Interrupt Watchdog and other system checks
Brownout Detector --->
```

Debugger...

1. En una terminal correr OpenOCD: **idf.py openocd**
2. Correr el debugger:
 - a. **idf.py gdb**
 - b. **idf.py gdb-tui**
 - c. **idf.py gdbgui**

Pasos para instalar ESP-IDF (Windows)

<https://docs.espressif.com/projects/esp-idf/en/latest/esp32/get-started/windows-setup.html>

Ver Videos de como instalar , ejecutar y usar. Version Command-line y VSCode

Una vez instalado...

1. Crear Proyecto: **idf.py create-project NOMBRE**
2. Definir chip: **idf.py set-target esp32c3**
3. Configurar componentes: **idf.py menuconfig (conviene apagar el watchdog)**
4. Compilar: **idf.py build**
5. Escribir Flash: **idf.py flash**
6. Monitorear: **idf.py monitor**

Debugger...

1. En una terminal correr OpenOCD: **idf.py openocd**
2. Correr el debugger:
 - a. **idf.py gdb**
 - b. **idf.py gdb-tui**
 - c. **idf.py gdbgui**

Hola Mundo LED

main.c

```
1 #include <stdio.h>
2 #include "driver/gpio.h"
3 #include "freertos/FreeRTOS.h"
4 #include "freertos/timers.h"
5
6 void app_main(void)
7 {
8     gpio_reset_pin(10);
9     gpio_set_direction(10, GPIO_MODE_OUTPUT);
10    while(1){
11        gpio_set_level(10,0);
12        vTaskDelay(1000/portTICK_PERIOD_MS);
13        gpio_set_level(10,1);
14        vTaskDelay(1000/portTICK_PERIOD_MS);
15    }
16 }
17
```

void vTaskDelay(const TickType_t xTicksToDelay)

Delay a task for a given number of ticks.

Delay a task for a given number of ticks. The actual time that the task remains blocked depends on the tick rate. The constant portTICK_PERIOD_MS can be used to calculate real time from the tick rate - with the resolution of one tick period.

esp_err_t gpio_reset_pin(gpio_num_t gpio_num)

Reset an gpio to default state (select gpio function, enable pullup and disable input and output).

esp_err_t gpio_set_direction(gpio_num_t gpio_num, gpio_mode_t mode)

GPIO set direction.

Configure GPIO direction,such as output_only,input_only,output_and_input

esp_err_t gpio_set_level(gpio_num_t gpio_num, uint32_t level)

GPIO set output level.

Note

This function is allowed to be executed when Cache is disabled within ISR context, by enabling `CONFIG_GPIO_CTRL_FUNC_IN_IRAM`

- Parameters:
- **gpio_num** – GPIO number. If you want to set the output level of e.g. GPIO16, gpio_num should be GPIO_NUM_16 (16);
 - **level** – Output level. 0: low ; 1: high

- Returns:
- ESP_OK Success
 - ESP_ERR_INVALID_ARG GPIO number error

Espressif IDE (configuración y proyecto inicial)

ESP-IDF Configuration

Download or use ESP-IDF

Click on "Finish" to configure IDF_PATH with /home/edgardog/esp/esp-idf

☒ Use an existing ESP-IDF directory from file system

Choose existing ESP-IDF directory: /home/edgardog/esp/esp-idf

Browse...

Download ESP-IDF

Please choose ESP-IDF version to download: v5.0.2

Choose a directory to download ESP-IDF to:

Browse...

For more information about ESP-IDF versions, see <https://docs.espressif.com/projects/esp-idf/en/latest/versions.html>

Note: The newly configured ESP-IDF will set to IDF_PATH in the CDT Build environment (Preferences > C/C++ > Build > Environment)

New IDF Project

Templates

Select one of the available templates to generate a fully-functioning IDF project.

Project name: HolaMundoIDE

☒ Create a project using one of the templates

Available IDF Templates:

type filter text

sample_project

blink

hello_world

storage

mesh

common_components

< Back

Next >

Cancel

Finish

Install Tools

ESP-IDF Tools installation dialog

Provide ESP-IDF directory, git and python executable paths to install the tools

ESP-IDF Directory: /home/edgardog/esp/esp-idf

Browse...

Git Executable Location: /usr/bin/git

Browse...

Python Executable Location: /usr/bin/python3

Browse...

?

Cancel

Install Tools

File Edit Source Refactor Navigate Search Project Run Espressif Window Help

Run

HolaMundoIDE

on esp32c3

Project Explorer

HolaMundoIDE

build

main

main.c

CMakeLists.txt

CMakeLists.txt

README.md

main.c

```
1 #include <stdio.h>
2
3 void app_main(void)
4 {
5
6 }
7
```

Build complete (0 errors, 0 warnings):

Espressif IDE (menuconfig más amigable)

Project Explorer ×

HolaMundoIDE

Binaries

Archives

build

main

main.c

CMakeLists.txt

CMakeLists.txt

README.md

sdkconfig

main.c

SDK Configuration ×

type filter text

Ethernet

Event Loop Library

GDB Stub

ESP HTTP client

HTTP Server

ESP HTTPS OTA

ESP HTTPS server

Hardware Settings

LCD and Touch Panel

ESP NETIF Adapter

Partition API Configuration

PHY

Power Management

ESP PSRAM

ESP Ringbuf

ESP System Settings

IPC (Inter-Processor Call)

✓ Enable memory protection

✓ Lock memory protection settings

System event queue size

32

Event loop task stack size

2304

Main task stack size

3584

Main task core affinity

CPU0

Minimal allowed size for shared stack

2048

Channel for console output

Default: UART0

Channel for console secondary output

USB_SERIAL_JTAG PORT

☐ Interrupt watchdog

☐ Enable Task Watchdog Timer

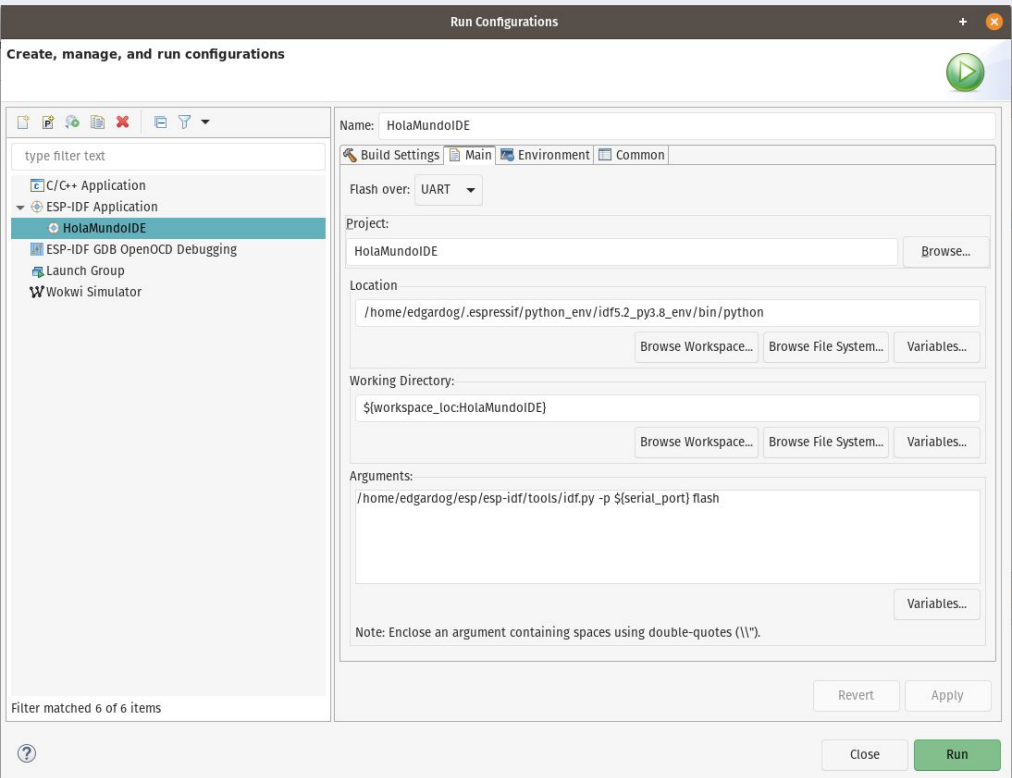
☐ Place panic handler code in IRAM

☐ OpenOCD debug stubs

☒ Make exception and panic handlers JTAG/OCD aware

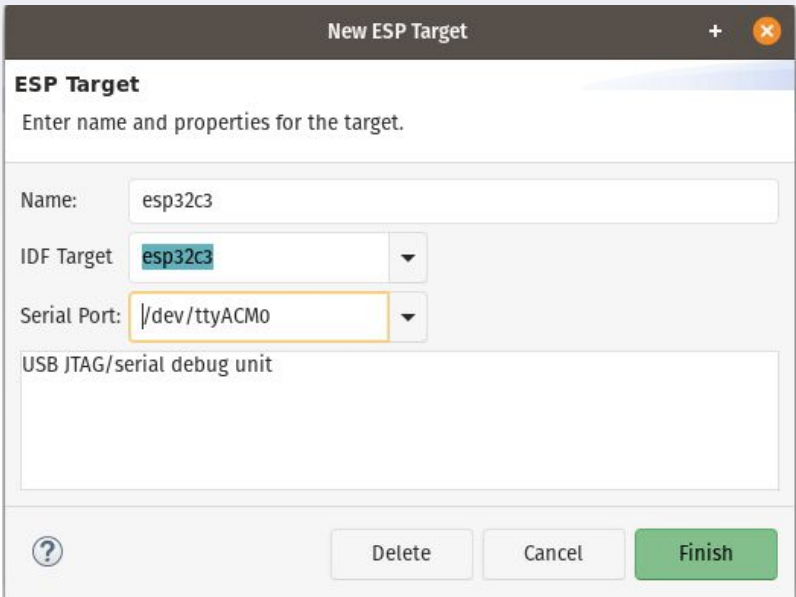
24

Espressif IDE (Run Configuration)



```
Wrote 164512 bytes (87447 compressed) at 0x00010000 in 1.5 seconds (effective 907.2 kbit/s)...
Hash of data verified.
Compressed 3072 bytes to 103...
Writing at 0x00008000... (100 %)
Wrote 3072 bytes (103 compressed) at 0x00008000 in 0.0 seconds (effective 614.8 kbit/s)...
Hash of data verified.
```

```
Leaving...
Hard resetting via RTS pin...
Done
```



Espressif IDE (Debug Configuration)

- C/C++ Application
- C/C++ Attach to Application
- C/C++ Postmortem Debugger
- C/C++ Remote Application
- ESP-IDF Application
 - HolaMundoIDE
 - ESP-IDF GDB OpenOCD Debugging
 - HolaMundoIDE Configuration**
 - GDB Hardware Debugging
- Launch Group

Name: HolaMundoIDE Configuration

Main Debugger Startup Source Common SVD Path

OpenOCD Setup

☒ Start OpenOCD locally

Executable path: \${openocd_path}/\${openocd_executable}

Actual executable: /home/edgardog/.espressif/tools/openocd-esp32/v0.12.0-esp32-20230419/openocd-esp32/bin/openocd
(to change it use the [global](#) or [workspace](#) preferences pages or the [project](#) properties page)

GDB port: 3333

Telnet port: 4444

Tcl port: 6666

Flash voltage: default

Target: esp32c3

Board: ESP32-C3 chip (via builtin USB-JTAG)

Config options: -s \${openocd_path}/share/openocd/scripts -f board/esp32c3-builtin.cfg

☒ Allocate console for OpenOCD ☐ Allocate console for the telnet connection

GDB Client Setup

☒ Start GDB session

Actual Executable: /home/edgardog/.espressif/tools/riscv32-esp-elf-gdb/12.1_20221002/riscv32-esp-elf-gdb/bin/riscv32-esp-elf-gdb

Other options:

Commands: set mem inaccessible-by-default off
set remotetimeout 20



```
1  void app_main(void)
2  {
3      #include "freertos/FreeRTOS.h"
4      #include "freertos/timers.h"
5
6      gpio_reset_pin(10);
7      gpio_set_direction(10, GPIO_MODE_OUTPUT);
8      while(1){
9          gpio_set_level(10, 0);
10         vTaskDelay(1000/portTICK_PERIOD_MS);
11         gpio_set_level(10, 1);
12         vTaskDelay(1000/portTICK_PERIOD_MS);
13     }
14 }
```

Registers	
Name	Value
General Registers	
zero	0
ra	0x42014e52 <main_task+92>
sp	0x3fc8f610
an	rv3fre32000

Main Debugger Startup Source Common SVD Path

OpenOCD Options

☒ Flash every time with application binaries

☐ Enable verbose output

Initialization Commands

☒ Initial Reset. Type: init

mon reset halt
flushregs
set remote hardware-watchpoint-limit 2

☒ Enable ARM semihosting

Load Symbols and Executable

☒ Load symbols

☒ Use project binary: main.elf

☐ Use file:

Symbols offset (hex):

☐ Load executable

☒ Use project binary: main.elf

☐ Use file:

Executable offset (hex):

Runtime Options

☐ Debug in RAM

Run/Restart Commands

☒ Pre-run/Restart reset Type: halt (always executed at Restart)

☐ Set program counter at (hex):

☒ Set breakpoint at: app_main

```

1 #include <stdio.h>
2 #include "driver/gpio.h"
3 #include "freertos/FreeRTOS.h"
4 #include "freertos/timers.h"
5 #include "esp_log.h"
6
7 static const char* TAG = "HolaMundo";
8
9 void app_main(void)
10 {
11     uint16_t contador=0;
12     ESP_LOGI(TAG, "Iniciando contador en: %X", contador);
13     gpio_reset_pin(10);
14     gpio_set_direction(10, GPIO_MODE_OUTPUT);
15     while(1){
16         ESP_LOGI(TAG, "Cama Arriba...%X", contador);
17         gpio_set_level(10, 0);
18         vTaskDelay(1000/portTICK_PERIOD_MS);
19         ESP_LOGI(TAG, "Cama Abajo...%X", contador);
20         gpio_set_level(10, 1);
21         vTaskDelay(1000/portTICK_PERIOD_MS);
22         contador++;
23     }
24 }

```

Utilidades de LOG

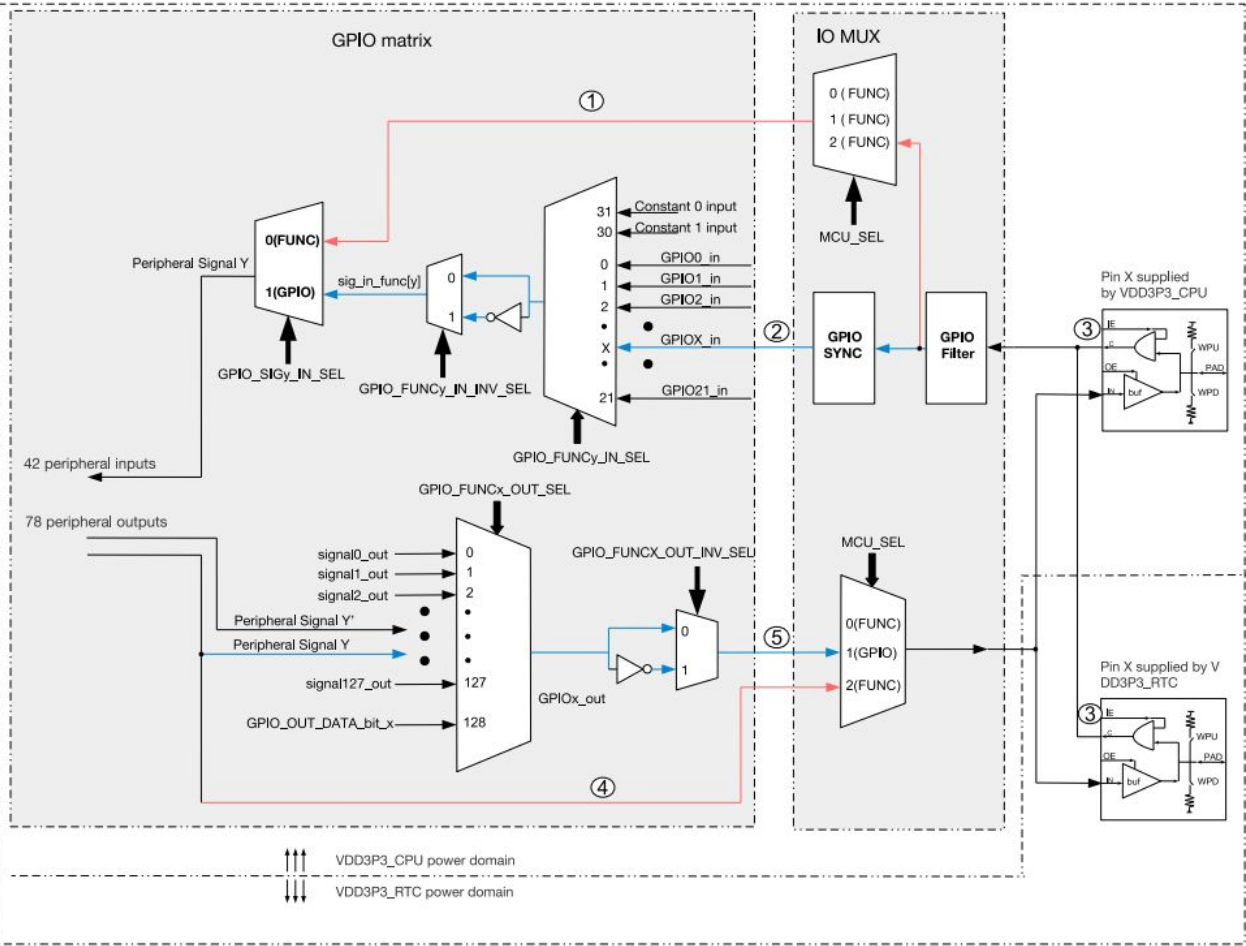
```

I (469) app_start: Starting scheduler on CPU0
I (474) main_task: Started on CPU0
I (474) main_task: Calling app_main()
I (474) HolaMundo: Iniciando contador en: 0
I (474) gpio: GPIO[10]| InputEn: 0| OutputEn: 0|
I (474) HolaMundo: Cama Arriba...0
I (1474) HolaMundo: Cama Abajo...0
I (2474) HolaMundo: Cama Arriba...1
I (3474) HolaMundo: Cama Abajo...1

```

Unidad 2.1

Interfaces de E/S en ESP32C3



```
#ifndef ESP32C3
#define UART_BASE_REG 0x60000000 /* UART0
#define SPI_BASE_REG 0x60002000 /* SPI pin
#define SPI0_BASE_REG 0x60003000 /* SPI pin
#define GPIO_BASE_REG 0x60004000
#define RTCCNTL_BASE_REG 0x60008000 /* RTC Control
#define USB_DEVICE_BASE_REG 0x60043000
#define SYSTEM_BASE_REG 0x600C0000
#endif
```


Name	Description	Address	Access
Configuration Registers			
GPIO_BT_SELECT_REG	GPIO bit select register	0x0000	R/W
GPIO_OUT_REG	GPIO output register	0x0004	R/W/SS
GPIO_OUT_W1TS_REG	GPIO output set register	0x0008	WT
GPIO_OUT_W1TC_REG	GPIO output clear register	0x000C	WT
GPIO_ENABLE_REG	GPIO output enable register	0x0020	R/W/SS
GPIO_ENABLE_W1TS_REG	GPIO output enable set register	0x0024	WT
GPIO_ENABLE_W1TC_REG	GPIO output enable clear register	0x0028	WT
GPIO_STRAP_REG	pin strapping register	0x0038	RO
GPIO_IN_REG	GPIO input register	0x003C	RO
GPIO_STATUS_REG	GPIO interrupt status register	0x0044	R/W/SS
GPIO_STATUS_W1TS_REG	GPIO interrupt status set register	0x0048	WT
GPIO_STATUS_W1TC_REG	GPIO interrupt status clear register	0x004C	WT
GPIO_PCPU_INT_REG	GPIO PRO_CPU interrupt status register	0x005C	RO
GPIO_PCPU_NMI_INT_REG	GPIO PRO_CPU (non-maskable) interrupt status register	0x0060	RO
GPIO_STATUS_NEXT_REG	GPIO interrupt source register	0x014C	RO

wt= write 1 to trigger

```
#define GPIO_OUT_W1TC_REG
```

```
(DR_REG_GPIO_BASE + 0x000c)
```

Register 5.4. GPIO_OUT_W1TC_REG (0x000C)

(reserved)

GPIO_OUT_W1TC

31	26	25	0
0	0	0	0 0 0 0 0 0

0x00000

Reset

GPIO_OUT_W1TC GPIO0 ~ 21 output clear register. Bit0 ~ bit21 are corresponding to GPIO0 ~ 21, and bit22 ~ bit25 are invalid. If the value 1 is written to a bit here, the corresponding bit in **GPIO_OUT_REG** will be cleared. Recommended operation: use this register to clear **GPIO_OUT_REG**. (WT)

Register 5.3. GPIO_OUT_W1TS_REG (0x0008)

(reserved)

GPIO_OUT_W1TS

31	26	25	0
0	0	0	0 0 0 0 0 0

0x00000

Reset

GPIO_OUT_W1TS GPIO0 ~ 21 output set register. Bit0 ~ bit21 are corresponding to GPIO0 ~ 21, and bit22 ~ bit25 are invalid. If the value 1 is written to a bit here, the corresponding bit in **GPIO_OUT_REG** will be set to 1. Recommended operation: use this register to set **GPIO_OUT_REG**. (WT)

Unidad 2.2

Excepciones, Interrupciones y Traps



Manual RISC-V , Seccion 1.3 (Exceptions, Traps, and Interrupts)

Según la arquitectura RISC-V, se define como **excepción** a una condición inusual que se produce durante la ejecución de una instrucción. Ejemplo: se ejecuta una instrucción ebreak (breakpoint sw), se accede a una dirección de manera no alineada, etc. Si bien en otras arquitecturas dividir por cero genera una excepción, en RISC-V no (página 44 del manual).

Una **interrupción** es un evento externo asincrónico (que puede o no ser atendido dependiendo de los registros mstatus, mie y mip).

Tanto en el caso de una excepción o de una interrupción (atendida), se “atrapa” (**trap**) la misma transfiriendo de manera sincrónica el control de la CPU a una rutina manejadora de trap. A tal fin, existen registros específicos en donde la microarquitectura inserta valores necesarios para identificar el estado:

- **mepc**: Este registro almacena el valor del PC cuando se produjo la excepción o interrupción.
- **mcause**: Este registro indica si se produjo una excepción (bMS=0) o una interrupción (bMS=1). El resto almacena un número de identificación.
- **mtval**: Este registro almacena un valor necesario para resolver el problema. Ej: Load Address Misaligned es un acceso no alineado en una instrucción tipo lw, la dirección accedida (desalineada) se almacena en mtval.
- **mscratch**: Registro auxiliar para valor temporal.

Los registros de uso general NO se almacenan automáticamente. Para retornar del trap se utiliza una instrucción especial **mret** que toma la dirección de mepc.

0	Instruction address misaligned
1	Instruction access fault
2	Illegal instruction
3	Breakpoint
4	Load address misaligned
5	Load access fault
6	Store/AMO address misaligned
7	Store/AMO access fault
8	Environment call from U-mode
9	Environment call from S-mode
10	Reserved
11	Environment call from M-mode
12	Instruction page fault
13	Load page fault
14	Reserved
15	Store/AMO page fault
≥16	Reserved

Vector de interrupciones

RISC-V define un registro (**mtvec**) donde se almacena la dirección de comienzo del vector de interrupciones. Los dos bits menos significativos de este registro indica el modo de operación:

- **Modo Directo** : Toda excepción o interrupción es manejada por mtvec[0].
- **Modo Vectorizado**: Las excepciones se manejan por mtvec[0], pero las interrupciones se manejan por mtvec[mcause*4].

En cada posición del vector se almacena una instrucción. Cada una de esas instrucciones debe saltar al trap correspondiente. Cambiando el bit **mstatus.mie=0** se apagan todas las interrupciones (NO las excepciones).

Interrupt	Exception Code	Description
1	0	User software interrupt
1	1	Supervisor software interrupt
1	2	<i>Reserved</i>
1	3	Machine software interrupt
1	4	User timer interrupt
1	5	Supervisor timer interrupt
1	6	<i>Reserved</i>
1	7	Machine timer interrupt
1	8	User external interrupt
1	9	Supervisor external interrupt
1	10	<i>Reserved</i>
1	11	Machine external interrupt
1	≥12	<i>Reserved</i>

RISC-V soporta niveles de privilegio (usuario, supervisor, maquina). En el caso de interrupciones define 3 tipos de interrupción:

- **Software**: Estas interrupciones son generadas por código que escribe en el controlador programable de interrupciones (CLINT).
- **Timer**: Se generan cuando se produce un evento de timer que requiere atención.
- **Externa**: Se producen como resultado de un dispositivo o periférico que interrumpe.

En el caso de esp32 (esp-idf) esto se define en components/riscv/vectors.S

Interrupciones Externas (gpio - inicialización)

```
esp_err_t gpio_isr_handler_add(gpio_num_t gpio_num, gpio_isr_t isr_handler, void *args)
```

Add ISR handler for the corresponding GPIO pin.

Call this function after using `gpio_install_isr_service()` to install the driver's GPIO ISR handler service.

The pin ISR handlers no longer need to be declared with `IRAM_ATTR`, unless you pass the `ESP_INTR_FLAG_IRAM` flag when allocating the ISR in `gpio_install_isr_service()`.

```
esp_err_t gpio_install_isr_service(int intr_alloc_flags)
```

Install the GPIO driver's `ETS_GPIO_INTR_SOURCE` ISR handler service, which allows per-pin GPIO interrupt handlers.

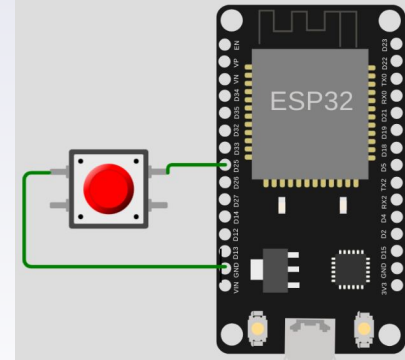
```
esp_err_t gpio_set_intr_type(gpio_num_t gpio_num, gpio_int_type_t intr_type)
```

GPIO set interrupt trigger type.

- Parameters:**
- `gpio_num` – GPIO number. If you want to set the trigger type of e.g. of GPIO16, `gpio_num` should be `GPIO_NUM_16` (16);
 - `intr_type` – Interrupt type, select from `gpio_int_type_t`

Returns:

- `ESP_OK` Success
- `ESP_ERR_INVALID_ARG` Parameter error



```
enum gpio_int_type_t
```

Values:

```
enumerator GPIO_INTR_DISABLE
```

Disable GPIO interrupt

```
enumerator GPIO_INTR_POSEDGE
```

GPIO interrupt type : rising edge

```
enumerator GPIO_INTR_NEGEDGE
```

GPIO interrupt type : falling edge

```
enumerator GPIO_INTR_ANYEDGE
```

GPIO interrupt type : both rising and falling edge

```
enumerator GPIO_INTR_LOW_LEVEL
```

GPIO interrupt type : input low level trigger

```
enumerator GPIO_INTR_HIGH_LEVEL
```

GPIO interrupt type : input high level trigger

Interrupciones Externas (gpio - ejemplo)

```
#define IRQ_PIN 10
#define LED_PIN 2

static uint8_t estadoLed=0;

static void IRAM_ATTR rutinaISR_IRQ(void *args)
{
    estadoLed=(estadoLed+1) %2;
    gpio_set_level(LED_PIN,estadoLed);
}

void app_main(void)
{
    //Defino entradas y salidas
    gpio_reset_pin(LED_PIN);
    gpio_set_direction(LED_PIN,GPIO_MODE_OUTPUT);
    gpio_set_direction(IRQ_PIN, GPIO_MODE_INPUT);
    gpio_pullup_dis(IRQ_PIN);
    gpio_pullup_dis(IRQ_PIN);
    gpio_set_intr_type(IRQ_PIN, GPIO_INTR_POSEDGE);
    gpio_install_isr_service(0);
    //Defino rutina ISR
    gpio_isr_handler_add(IRQ_PIN, rutinaISR_IRQ, NULL);

    gpio_set_level(LED_PIN,estadoLed);

    while(1){

    }
}
```

Interrupciones Externas (gpio)

```
#define IRQ_PIN 10
#define LED_PIN 2

static uint8_t estadoLed=0;

static void IRAM_ATTR rutinaISR_IRQ(void *args)
{
    estadoLed=(estadoLed+1) %2;
    gpio_set_level(LED_PIN,estadoLed);
}

void app_main(void)
{
    //Defino entradas y salidas
    gpio_reset_pin(LED_PIN);
    gpio_set_direction(LED_PIN,GPIO_MODE_OUTPUT);
    gpio_set_direction(IRQ_PIN, GPIO_MODE_INPUT);
    gpio_pullup_dis(IRQ_PIN);
    gpio_pullup_dis(IRQ_PIN);
    gpio_set_intr_type(IRQ_PIN, GPIO_INTR_POSEDGE);
    gpio_install_isr_service(0);
    //Defino rutina ISR
    gpio_isr_handler_add(IRQ_PIN, rutinaISR_IRQ, NULL);

    gpio_set_level(LED_PIN,estadoLed);

    while(1){

    }
```

Mientras se está atendiendo una IRQ, NO puede ingresar otra de la misma prioridad. Si ingresa otra IRQ la misma queda encolada y se atienda cuando finaliza la actual.

```
static void IRAM_ATTR rutinaISR_IRQ(void *args)
{
    estadoLed=(estadoLed+1) %2;
    volatile int contador = 100000000;
    while(contador>0){
        contador--;
    }
    gpio_set_level(LED_PIN,estadoLed);
}
```



Insertamos un delay por sw de aprox 8 segundos y generamos dos IRQ para ver esto.

Timers

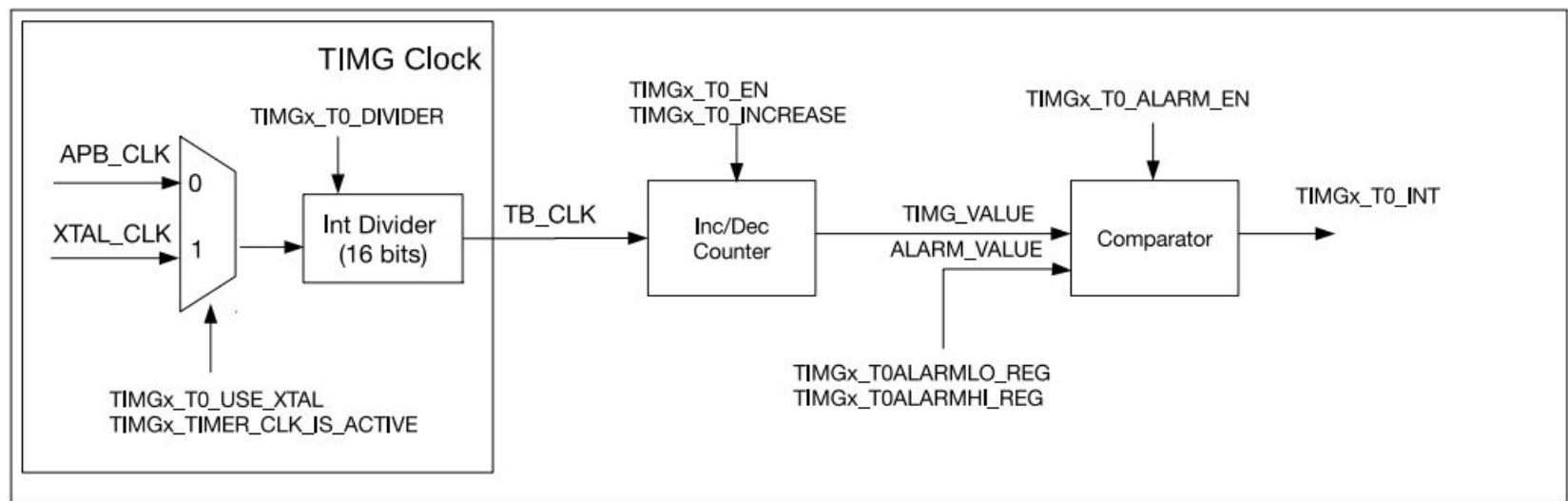


Figure 11-2. Timer Group Architecture

Si bien hay muchas formas de implementar timers, en casi todas las arquitecturas el timer está compuesto por un contador (asc/desc) cuyo clock (generalmente hay más de una opción) suele poder dividirse para regular la frecuencia del contador. Luego existe un comparador que todo el tiempo compara un valor fijo contra el contador, y si ambos son iguales se produce una interrupción. Luego el valor de comparación se vuelve a definir para regular la próxima interrupción. En otros diseños más simples es simplemente un contador inicializable en cualquier valor que genera una interrupción cuando hacer overflow ($FFFFFF \dots F \rightarrow 0$).

Timers (esp-timer.h)

```
#include "esp_timer.h"
```

```
void rutinaTIMER_ISR(void *param)
{
    estadoLed=(estadoLed+1) %2;
    gpio_set_level(LED_PIN,estadoLed);
}
```

```
const esp_timer_create_args_t variableTimer =
{
    .callback = &rutinaTIMER_ISR,
    .name = "Rutina del timer"
};

esp_timer_handle_t timer_handler;
esp_timer_create(&variableTimer, &timer_handler);
esp_timer_start_periodic(timer_handler, 1000000);
```

Timers (esp-timer.h)

```
static uint8_t flagEstado=0;

static void IRAM_ATTR rutinaISR_IRQ(void *args)
{
    flagEstado=1;
    gpio_set_level(LED_PIN,1);
}

void rutinaTIMER_ISR(void *param)
{
    gpio_set_level(LED_PIN,0);
}

void app_main(void)
{
    //Defino entradas y salidas
    gpio_reset_pin(LED_PIN);
    gpio_set_direction(LED_PIN,GPIO_MODE_OUTPUT);
    gpio_set_direction(IRQ_PIN, GPIO_MODE_INPUT);
    gpio_pullup_dis(IRQ_PIN);
    gpio_pullup_dis(IRQ_PIN);
    gpio_set_intr_type(IRQ_PIN, GPIO_INTR_POSEDGE);
    gpio_install_isr_service(0);
    //Defino rutina ISR
    gpio_isr_handler_add(IRQ_PIN, rutinaISR_IRQ, NULL);
    gpio_set_level(LED_PIN,0);
    const esp_timer_create_args_t variableTimer ={
        .callback = &rutinaTIMER_ISR,
        .name = "Rutina del timer"};
    esp_timer_handle_t timer_handler;
    esp_timer_create(&variableTimer, &timer_handler);
    while(1){
        if (flagEstado){
            flagEstado=0;
            //Enciendo el timer:
            esp_timer_start_once(timer_handler, 5000000);
        }
    }
}
```


25% Duty Cycle



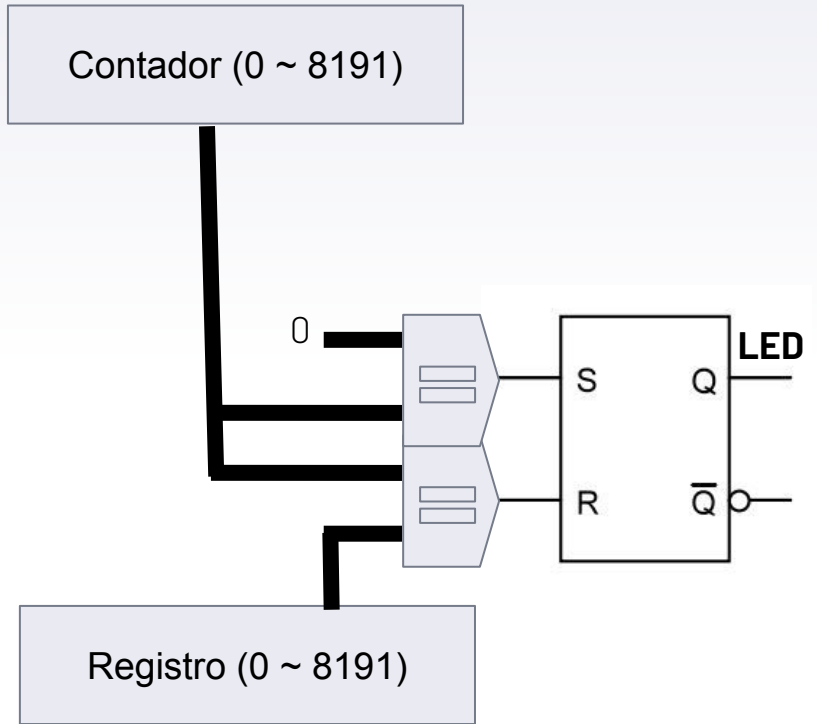
50% Duty Cycle



75% Duty Cycle



← T →



PWM (Pulse Width Modulation - Modulación por ancho de pulso)

OSC=40MHz, Freq PWM = 5KHz , 8000 cuentas en un segundo.

```
#include "driver/ledc.h"

#define LEDC_TIMER          LEDC_TIMER_0
#define LEDC_MODE           LEDC_LOW_SPEED_MODE
#define LEDC_OUTPUT_IO      (2) //GPIO
#define LEDC_CHANNEL        LEDC_CHANNEL_0
#define LEDC_DUTY_RES       LEDC_TIMER_13_BIT //13 bits de contador
#define LEDC_DUTY           (4192) //50% de 8192
#define LEDC_FREQUENCY      (5000) // 5KHz

void app_main(void)
{
    ledc_timer_config_t configTimer = {
        .speed_mode      = LEDC_MODE,
        .timer_num       = LEDC_TIMER,
        .duty_resolution  = LEDC_DUTY_RES,
        .freq_hz         = LEDC_FREQUENCY,
        .clk_cfg         = LEDC_AUTO_CLK
    };
    ledc_timer_config(&configTimer);
    ledc_channel_config_t configChannel = {
        .speed_mode      = LEDC_MODE,
        .channel         = LEDC_CHANNEL,
        .timer_sel       = LEDC_TIMER,
        .intr_type       = LEDC_INTR_DISABLE,
        .gpio_num        = LEDC_OUTPUT_IO,
        .duty            = 0,
        .hpoint          = 0
    };
    ledc_channel_config(&configChannel);
    ledc_set_duty(LEDC_MODE, LEDC_CHANNEL, LEDC_DUTY);
    ledc_update_duty(LEDC_MODE, LEDC_CHANNEL);
    while(1);
}
```

