Internet de las cosas y fundamentos de FreeRTOS con el ESP32WROOM

Circuit and System Society IEEE UNAM
Instructores:
Miguel Maldonado miguelmaldonadomartinez@gmail.com
Daniel Ortiz
Salvador Sarabia salvador_sarabia@outlook.com

Objetivo del curso

- Mostrar las bases de los sistemas operativos en tiempo real y los principios de funcionamiento del internet con el propósito de ser usado en aplicaciones relacionadas con el Internet de las cosas.

Dinámica del curso

- Introducción al tema.
- Preguntas de introducción (de forma aleatoria se preguntará, sin nadie la conoce se responderá y quedará de tarea profundizar más).
- Desarrollo del tema (en caso de ser necesario se desarrollarán ejemplos).
- En casa sesión se dejarán algunas tareas para reforzar lo aprendido en clase.

Contenido

1. Introducción a la programación en la tarjeta de desarrollo ESP32WROOM
2. Fundamentos de FreeRTOS
    - 2.1. ¿Qué es un RTOS?
    - 2.2. Concepto de tareas (tasks)
    - 2.3. Manejo de colas
    - 2.4. Semáforos en interrupciones
    - 2.5. Notificaciones
    - 2.6. Grupos de eventos
    - 2.7. Mutex
3. Modelo TCP/IP
4. Protocolo 802.11 (Wifi) e Internet de las Cosas (IoT)
5. Arquitectura REST y protocolo MQTT

Breve presentación

- Nombre.
- Universidad de procedencia y semestre.
- Microcontradores conocidos.
- Lenguajes de programación conocidos.
- Redes de comunicaciones conocidas.
- ¿Cúal es su experiencia con el IoT's?

1. Introducción a la programación en la tarjeta de desarrollo ESP32WROOM

Preguntas de introducción

- ¿Qué es un microcontrolador?
- ¿Cuál es la diferencia entre un microcontrolador y una computadora?
- ¿Cómo funciona un microcontrolador?
- Componentes de un microcontrolador
- Aplicaciones más comunes de un microcontrolador
- ¿Qué es lo más complejo que hayan hecho con un microcontrolador?

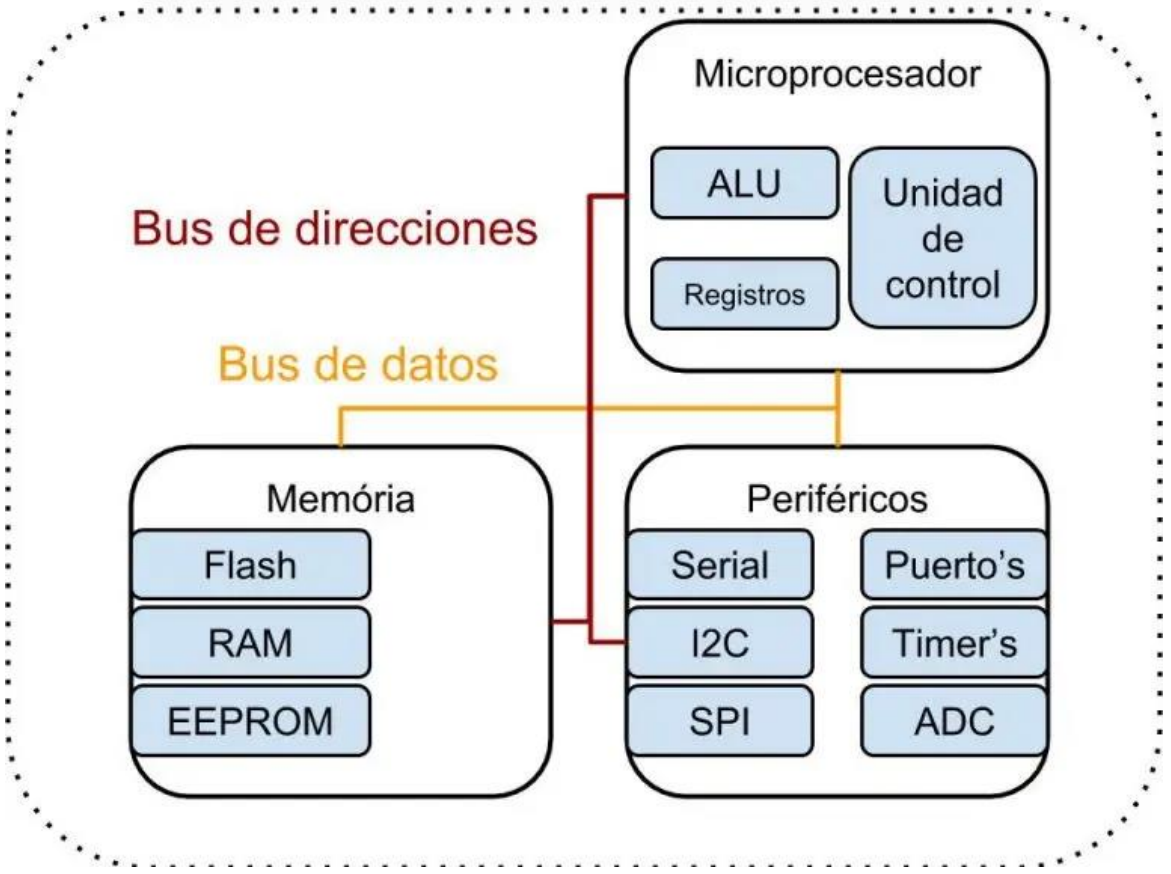
¿Qué es un microcontrolador?

Un microcontrolador es un circuito integrado que en su interior contiene una unidad central de procesamiento (CPU), unidades de memoria (RAM y ROM), puertos de entrada y salida y periféricos.

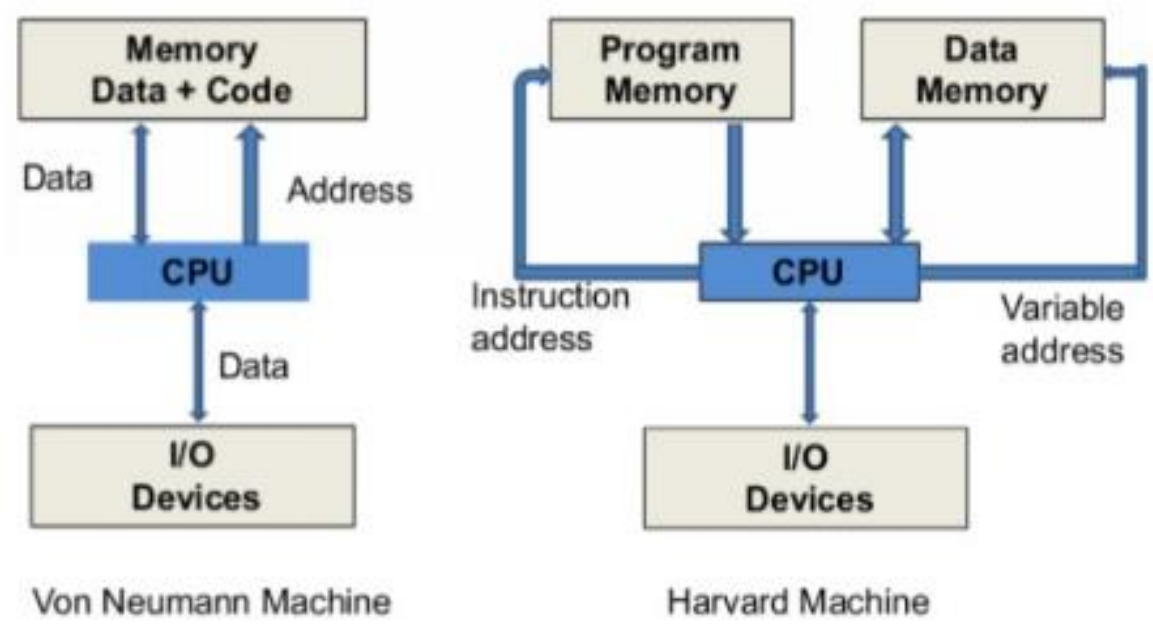
Estas partes están interconectadas dentro del microcontrolador y en conjunto forman lo que se le conoce como microcomputadora.

En conclusión, un microcontrolador es una microcomputadora completa encapsulada en un circuito integrado.

- Algunos periféricos con los que puede contar son:
- Periféricos de comunicación: UART, I2C, SPI, etc.
- Periféricos de adecuación de señales: ADC, DAC
- Periféricos generales: Timers, Controlador de interrupción, Controlador de osciladores.



Arquitectura de microcontroladores vs PC’s



Arquitectura von Neumann

- La arquitectura de Von Neumann tiene un solo bus que se utiliza para obtener instrucciones y transferir datos. Esta operación debe programarse porque no se pueden realizar al mismo tiempo.
- La unidad de procesamiento requeriría dos ciclos de reloj para completar una instrucción.
- Generalmente se usa desde computadoras de escritorio, computadoras portátiles, computadoras de alto rendimiento hasta estaciones de trabajo.

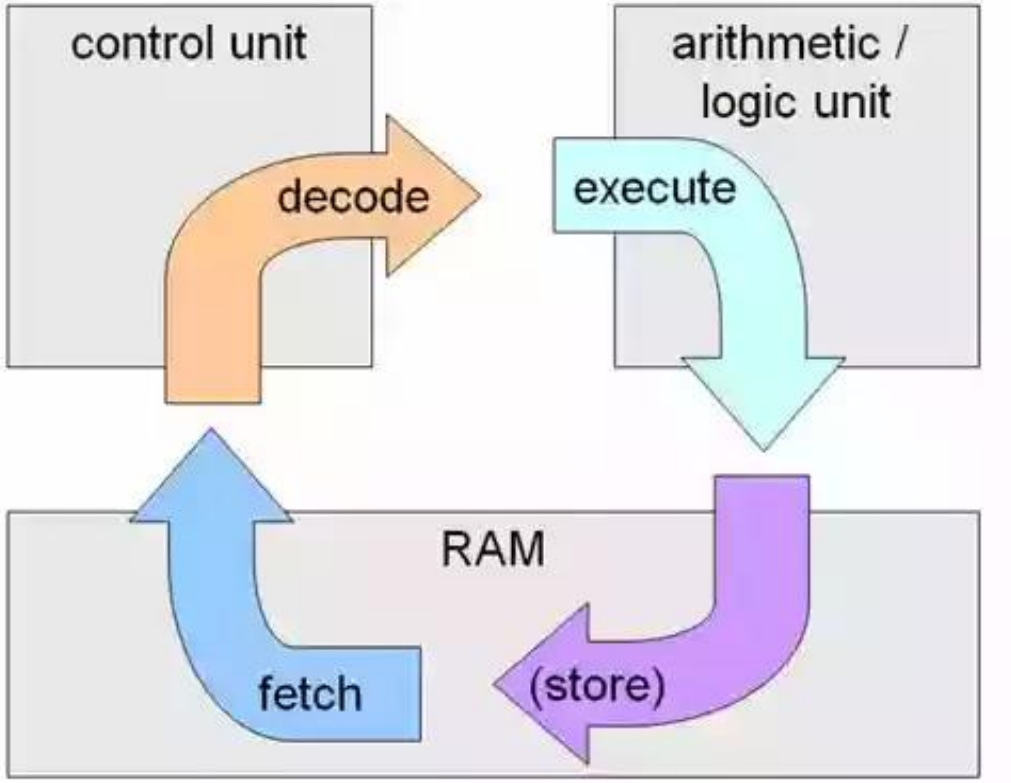
Arquitectura Harvard

- La arquitectura de Harvard tiene un espacio de memoria separado para instrucciones y datos que separa físicamente las señales y el código de almacenamiento y la memoria de datos, lo que a su vez permite acceder a cada uno de los sistemas de memoria simultáneamente.
- La unidad de procesamiento puede completar la instrucción en un ciclo si se han establecido los planes de canalización apropiados.
- Es un concepto utilizado específicamente en microcontroladores y procesamiento de señal digital (DSP).
- La arquitectura de Harvard es un tipo complejo de arquitectura porque emplea dos buses para instrucción y datos, un factor que hace que el desarrollo de la unidad de control sea relativamente más costoso.

Ciclo de fetch

La CPU ejecuta las instrucciones dentro de la memoria. Este proceso se realiza mediante las siguientes etapas:

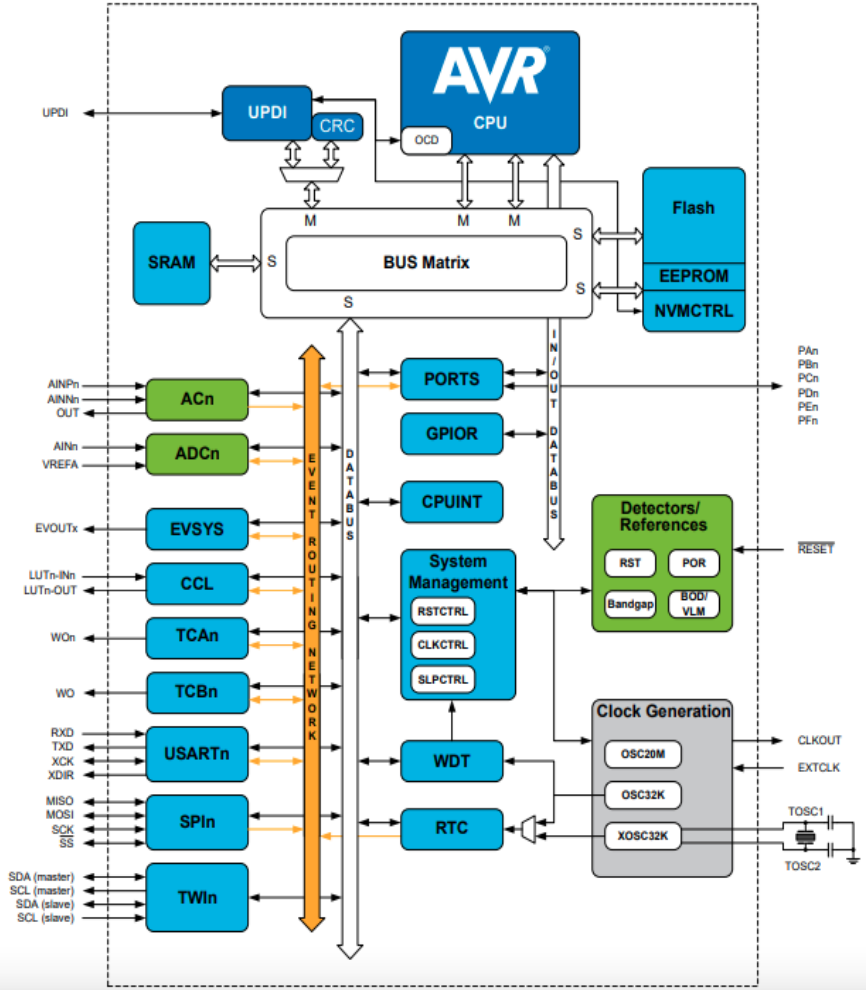
- Fetch o Captación: Etapa en la que la instrucción es captada desde la memoria RAM y copiada a dentro del procesador.
- Decode o Descodificacion: En la que la instrucción previamente captada es descodificada y enviada a las unidades de ejecución
- Execute o Ejecución: Donde la instrucción es resuelta y el resultado escrito en los registros internos del procesador o en una dirección de memoria de la RAM



Tarea moral 1 Diagrama de bloques ESP-WROOM-32

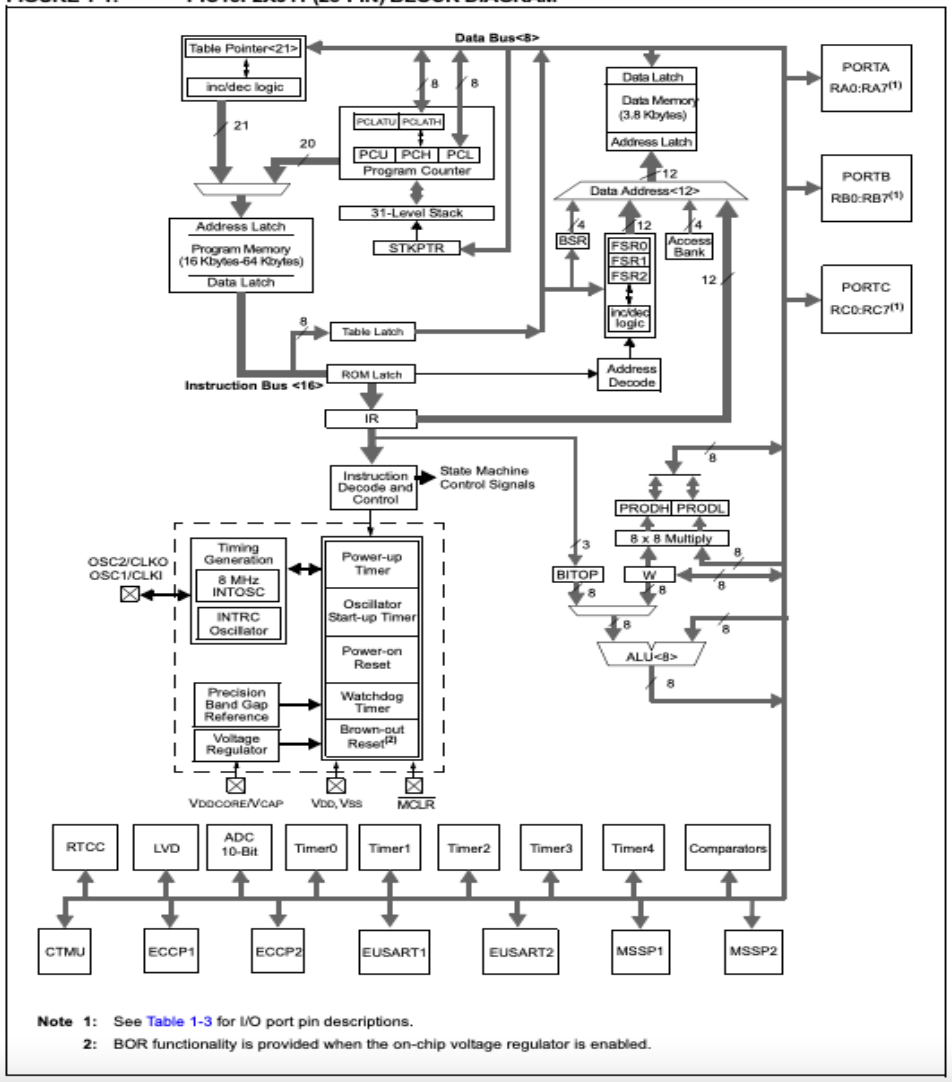
ATmega809/1609/3209/4809 – 48-pin
Block Diagram

1. Block Diagram



PIC18F46J11 FAMILY

FIGURE 1-1: PIC18F2XJ11 (28-PIN) BLOCK DIAGRAM



Note 1: See Table 1-3 for I/O port pin descriptions.
2: BOR functionality is provided when the on-chip voltage regulator is enabled.

Tarjeta de desarrollo ESP32WROOM

- El ESP32-WROOM-32 de Espressif Systems es una tarjeta de desarrollo basado en el **ESP32-D0WD** e integra módulos para aplicaciones con Wi-Fi / BT / BLE.
- Sus principales características son el incluir dos núcleos de CPU, frecuencia de reloj configurable de 80MHz a 240MHz e integrar varios periféricos tales como sensores táctiles capacitivos, UART, I2C, I2S, entre otros.
- Orientados a una amplia variedad de aplicaciones que van desde redes de sensores de baja potencia hasta las tareas más exigentes, como codificación de voz, transmisión de música y decodificación de MP3.
- El módulo ESP32-WROOM-32U es diferente de ESP32-WROOM-32D ya que integra un conector U.FL que debe conectarse a una antena IPEX externa.

ESP32-WROOM-32U

https://www.digikey.com.mx/es/products/detail/espressif-systems/ESP32-WROOM-32U-4MB/9381719

ESP32-WROOM-32D

https://www.mouser.mx/ProductDetail/Espressif-Systems/ESP32-WROOM-32D?qs=MLItCLRbWszx2KabkKPu5A%3D%3D

Propiedades del ESP32-WROOM-32

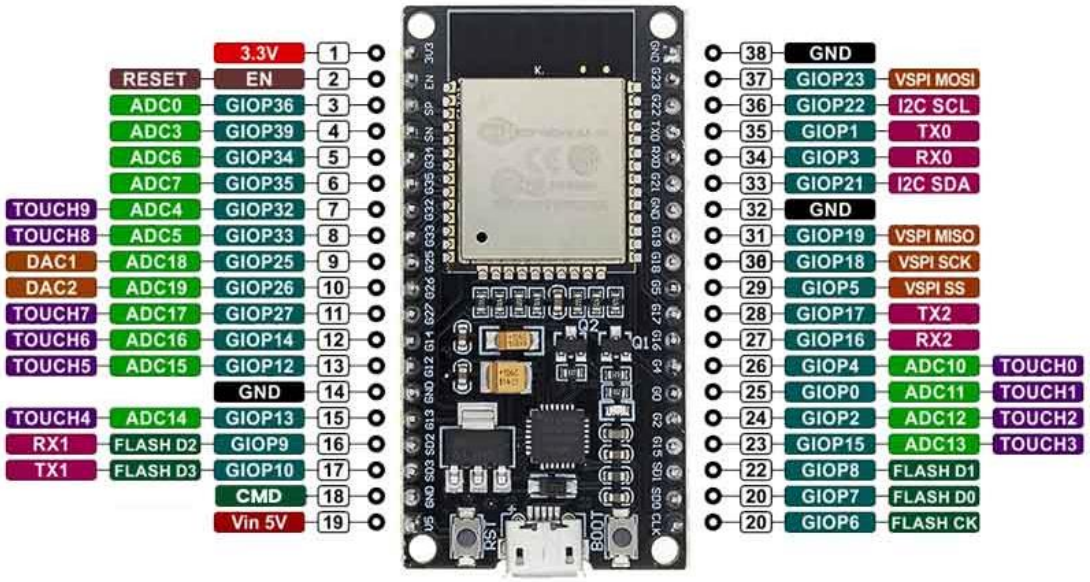
Características

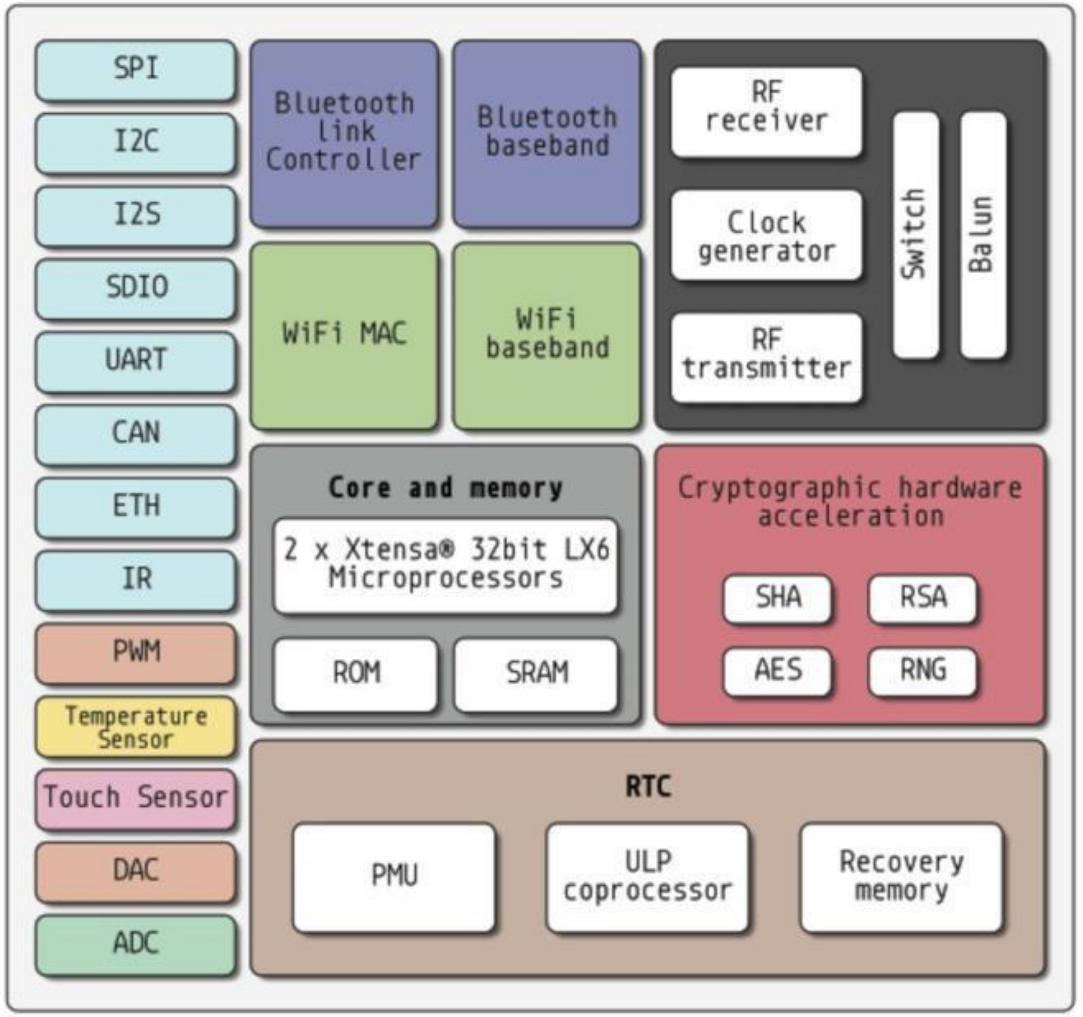
- Dos microprocesadores Xtensa® LX6 de 32 bits de bajo consumo
- Núcleo ESP32-D0WD
- Admite múltiples chips SRAM y flash QSPI externos
- Flash SPI de 32 Mbits y 3,3 V
- Cristal de 40MHz
- Antena de placa de circuito impreso para ESP32-WROOM-32D
- Conector U.FL (que debe conectarse a una antena IPEX externa) para ESP32-WROOM-32U
- Tecnologías avanzadas de administración de energía

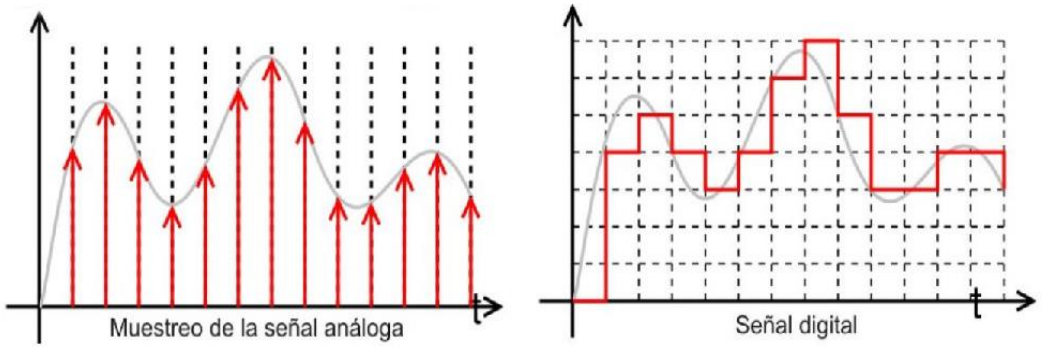
Especificaciones

- Protocolo Wi-Fi 802.11 b/g/n (802.11n hasta 150 Mbps)
- Bluetooth v4.2 BR / EDR y BLE
- Interfaces de tarjeta SD, UART, SPI, SDIO, I2C, LED PWM, Motor PWM, I2S y módulo IR
- Sensor de efecto Hall
- Suministro de voltaje de funcionamiento de 2,7 V a 3,6 V
- Corriente de funcionamiento media de 80 mA
- Rango de temperatura de funcionamiento de -40 ° C a 85 ° C

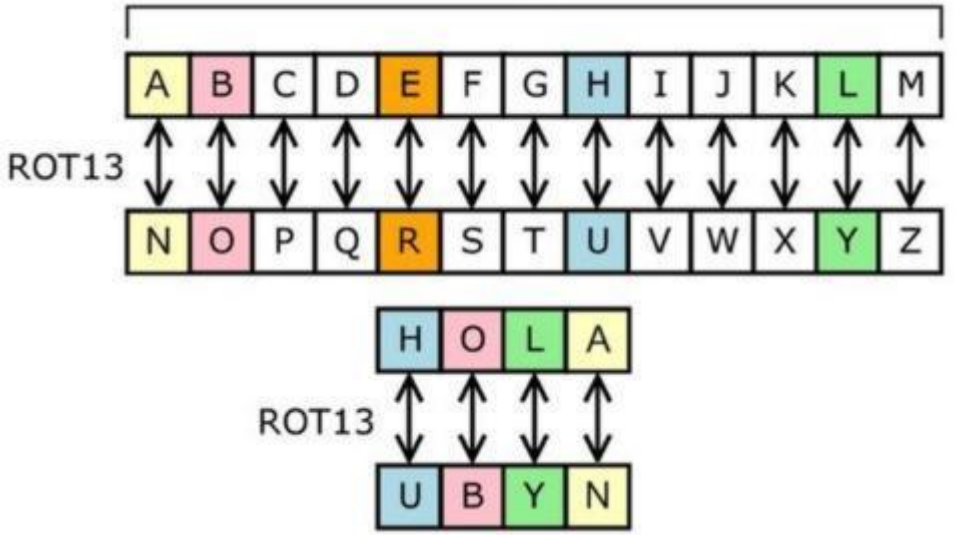
Diagrama de bloques e identificación de pines del ESP32-WROOM



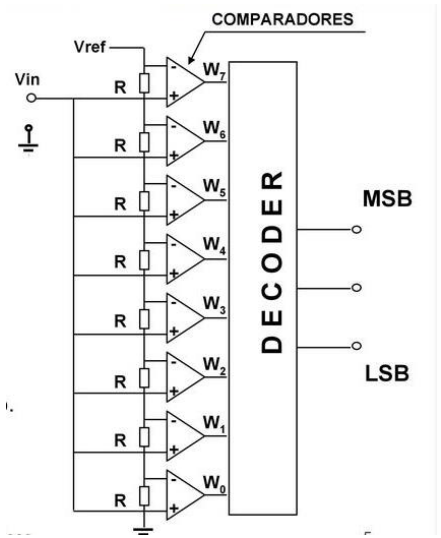




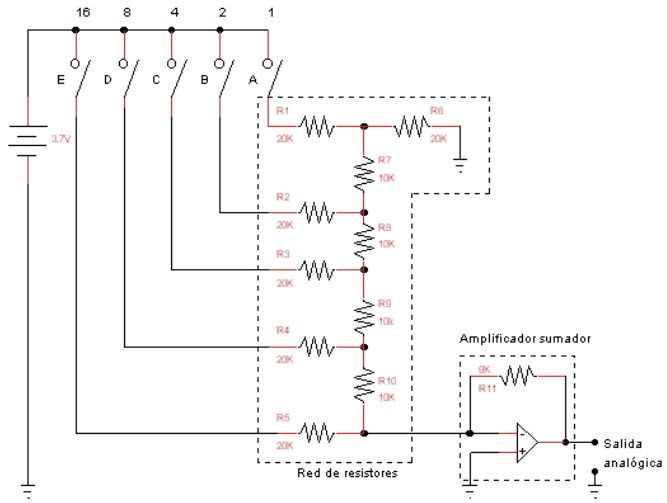
Señal analógica vs señal digital



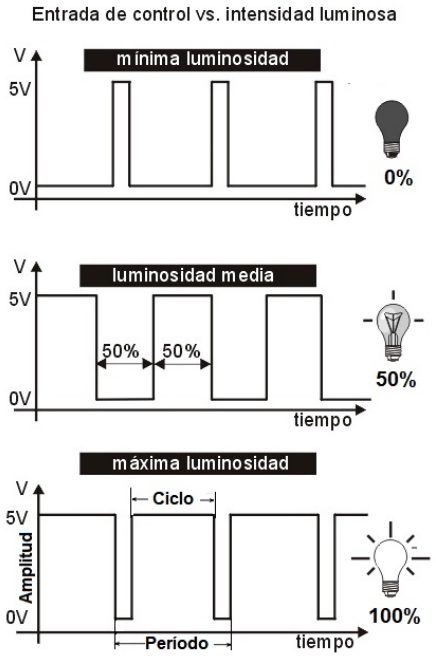
Ejemplo cifrado tipo cesar



Convertidor analógico-digital



Convertidor digital analógico



PWM

Name	No.	Type	Function
GND	1	P	Ground
3V3	2	P	Power supply.
EN	3	I	Chip-enable signal. Active high.
SENSOR_VP	4	I	GPIO36, SENSOR_VP, ADC_H, ADC1_CH0, RTC_GPIO0
SENSOR_VN	5	I	GPIO39, SENSOR_VN, ADC1_CH3, ADC_H, RTC_GPIO3
IO34	6	I	GPIO34, ADC1_CH6, RTC_GPIO4
IO35	7	I	GPIO35, ADC1_CH7, RTC_GPIO5
IO32	8	I/O	GPIO32, XTAL_32K_P (32.768 kHz crystal oscillator input), ADC1_CH4, TOUCH9, RTC_GPIO9
IO33	9	I/O	GPIO33, XTAL_32K_N (32.768 kHz crystal oscillator output), ADC1_CH5, TOUCH8, RTC_GPIO8
IO25	10	I/O	GPIO25, DAC_1, ADC2_CH8, RTC_GPIO6, EMAC_RXD0
IO26	11	I/O	GPIO26, DAC_2, ADC2_CH9, RTC_GPIO7, EMAC_RXD1
IO27	12	I/O	GPIO27, ADC2_CH7, TOUCH7, RTC_GPIO17, EMAC_RX_DV

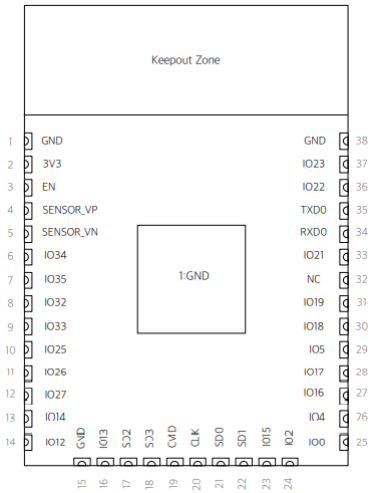


Tabla de descripción de pines y diagrama de distribución

[esp-wroom-32_datasheet_en-1223836.pdf (mouser.com)]

Name	No.	Type	Function
IO14	13	I/O	GPIO14, ADC2_CH6, TOUCH6, RTC_GPIO16, MTMS, HSPICLK, HS2_CLK, SD_CLK, EMAC_TXD2
IO12	14	I/O	GPIO12, ADC2_CH5, TOUCH5, RTC_GPIO15, MTDI, HSPIQ, HS2_DATA2, SD_DATA2, EMAC_TXD3
GND	15	P	Ground
IO13	16	I/O	GPIO13, ADC2_CH4, TOUCH4, RTC_GPIO14, MTCK, HSPID, HS2_DATA3, SD_DATA3, EMAC_RX_ER
SHD/SD2*	17	I/O	GPIO9, SD_DATA2, SPIHD, HS1_DATA2, U1RXD
SWP/SD3*	18	I/O	GPIO10, SD_DATA3, SPIWP, HS1_DATA3, U1TXD
SCS/CMD*	19	I/O	GPIO11, SD_CMD, SPICS0, HS1_CMD, U1RTS
SCK/CLK*	20	I/O	GPIO6, SD_CLK, SPICLK, HS1_CLK, U1CTS
SDO/SD0*	21	I/O	GPIO7, SD_DATA0, SPIQ, HS1_DATA0, U2RTS
SDI/SD1*	22	I/O	GPIO8, SD_DATA1, SPID, HS1_DATA1, U2CTS
IO15	23	I/O	GPIO15, ADC2_CH3, TOUCH3, MTDO, HSPICS0, RTC_GPIO13, HS2_CMD, SD_CMD, EMAC_RXD3
IO2	24	I/O	GPIO2, ADC2_CH2, TOUCH2, RTC_GPIO12, HSPIWP, HS2_DATA0, SD_DATA0
IO0	25	I/O	GPIO0, ADC2_CH1, TOUCH1, RTC_GPIO11, CLK_OUT1, EMAC_TX_CLK
IO4	26	I/O	GPIO4, ADC2_CH0, TOUCH0, RTC_GPIO10, HSPIHD, HS2_DATA1, SD_DATA1, EMAC_TX_ER
IO16	27	I/O	GPIO16, HS1_DATA4, U2RXD, EMAC_CLK_OUT
IO17	28	I/O	GPIO17, HS1_DATA5, U2TXD, EMAC_CLK_OUT_180
IO5	29	I/O	GPIO5, VSPICS0, HS1_DATA6, EMAC_RX_CLK
IO18	30	I/O	GPIO18, VSPICLK, HS1_DATA7
IO19	31	I/O	GPIO19, VSPIQ, U0CTS, EMAC_TXD0
NC	32	-	-
IO21	33	I/O	GPIO21, VSPIHD, EMAC_TX_EN
RXD0	34	I/O	GPIO3, U0RXD, CLK_OUT2
TXD0	35	I/O	GPIO1, U0TXD, CLK_OUT3, EMAC_RXD2
IO22	36	I/O	GPIO22, VSPIWP, U0RTS, EMAC_TXD1
IO23	37	I/O	GPIO23, VSPID, HS1_STROBE
GND	38	P	Ground

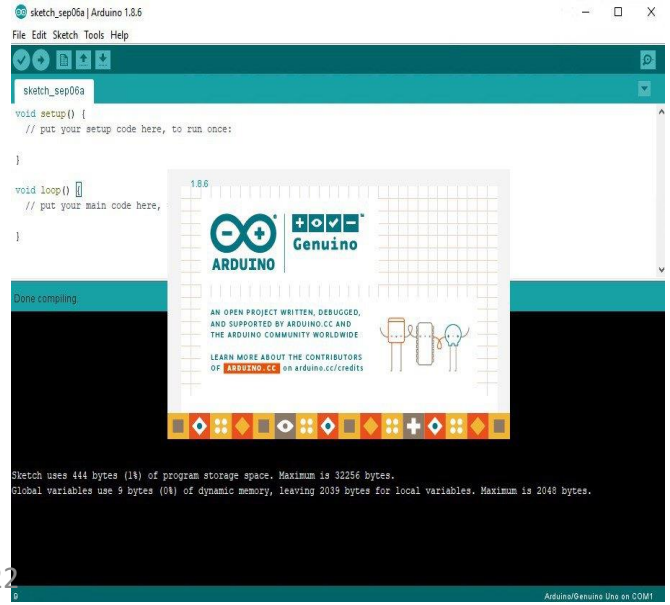
Adquisición del ESP32-WROOM (en México)

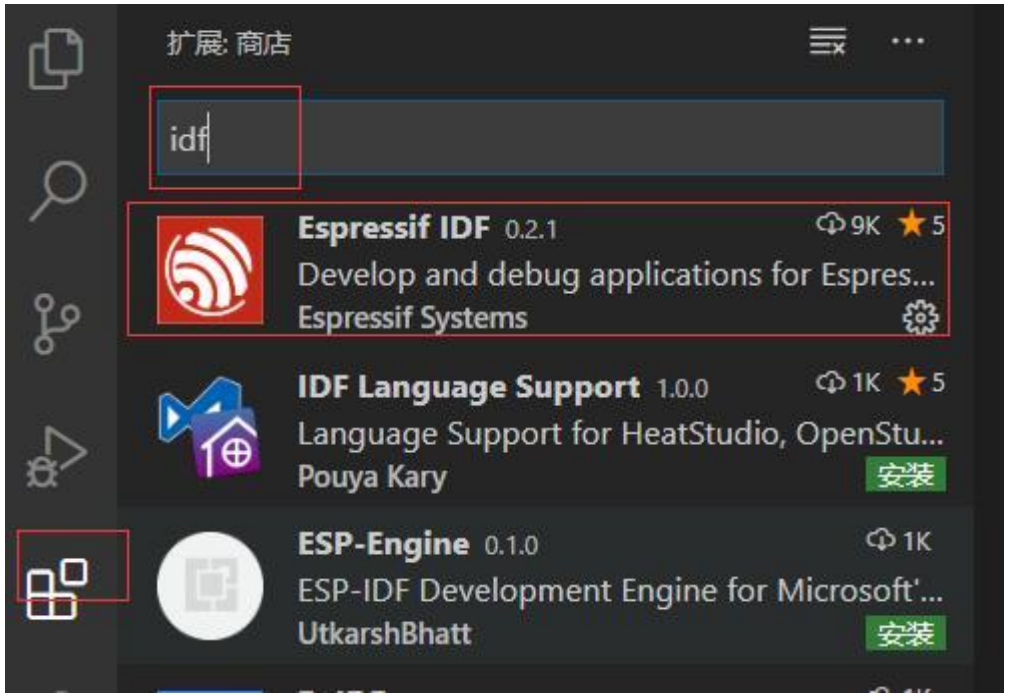
- UNIT Electronics: https://uelectronics.com/producto/esp32-38-pines-esp-wroom-32/
- Mercado Libre: https://articulo.mercadolibre.com.mx/MLM-587686290-esp32-wifi-bluetooth-42-ble-nodemcu-esp8266-libro-gratis-_JM#position=1&search_layout=grid&type=item&tracking_id=997b90a0-f7ca-4c68-b677-f324f3819908
- Alli Express: https://es.aliexpress.com/item/1005002786840436.html?src=google&aff_fcid=299d742febdd4e39a8dbc013ff3b803d-1640174257090-00607-UneMJZVf&aff_fsk=UneMJZVf&aff_platform=aaf&sk=UneMJZVf&aff_trace_key=299d742febdd4e39a8dbc013ff3b803d-1640174257090-00607-UneMJZVf&terminal_id=a8cdeae5369b4b7e82db58873a31ea0b

Entornos de desarrollo para ESP32-WROOM

- Arduino IDE (Basado en processing) + ESP IDF
- Visual Studio Code + ESP-IDF + node.js
- ISD-IDF + Windows Terminal o Bash (avanzado)

En nuestro curso veremos el uso Visual Studo Code como entonrno de desarrollo basado en C. (Revisar “Manual de instalación de ESP-IDF para el sistema operativo Windows”)





Tarea 1

- Descargar y realizar los pasos indicados en el manual “IEEE CAS Manual Software IoT + RTOS ESP32.pdf”
- Usar github para descargar el repositorio (https://github.com/IEEE-CAS-UNAM/IoT-RTOS.git) –Habilitar acceso-
- Usar la función “clone” para descargar el repositorio.
- Se mostrará ejemplo gráfico.

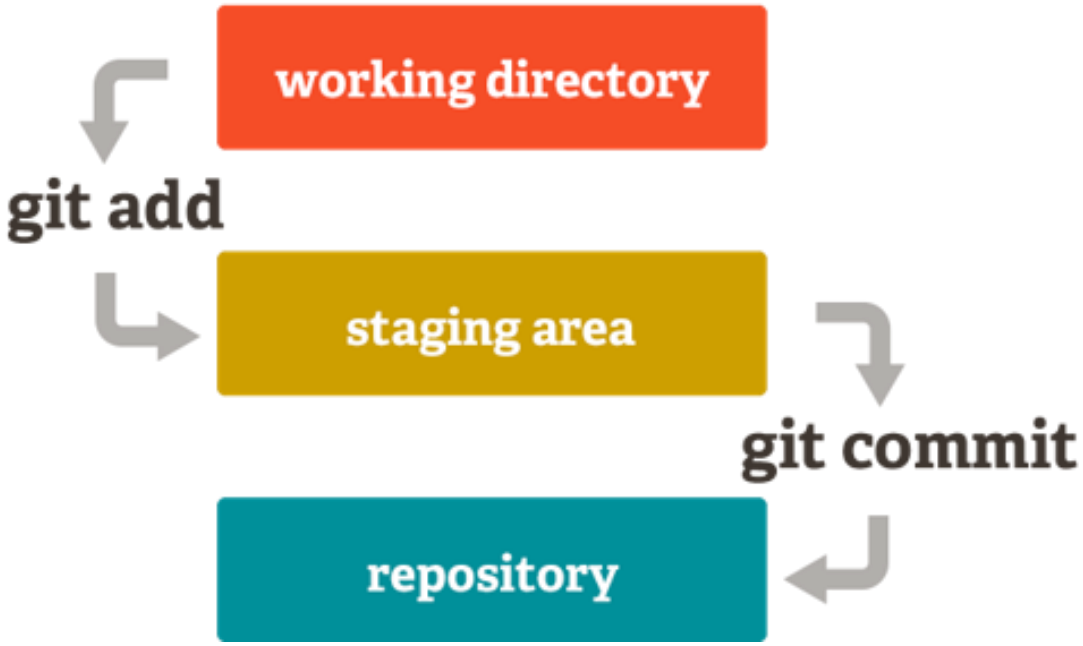
Enlaces de interés

- https://docs.github.com/en/get-started/quickstart/hello-world
- https://git-scm.com/
- https://desktop.github.com/

git

Sistema de control de versiones que permite manejar un proyecto de manera local.
Algunos comandos conocidos son:

- `sudo apt-get install git`
- `touch README.md`
- `git init`
- `git init [nombre del proyecto]`
- `git add README.md`
- `git commit -m "comentario"`
- `git remote add origin https://github.com/Proyecto/Repositorio.git`
- `git push -u origin master`
- `git clone nombredeusuario@host:/path/to/repository`
- `git pull`



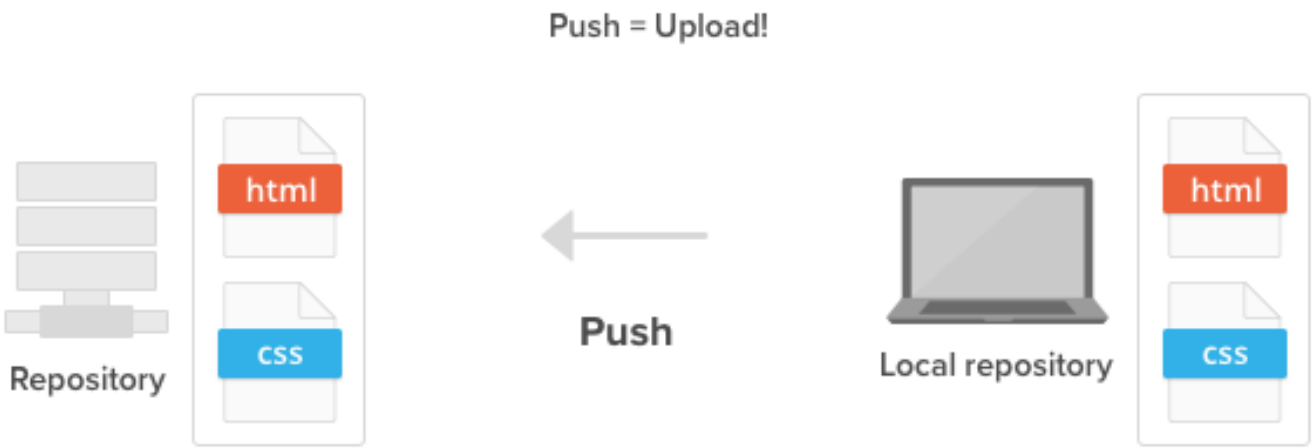
https://git-scm.com/

GitHub

GitLab

ATLASSIAN Bitbucket

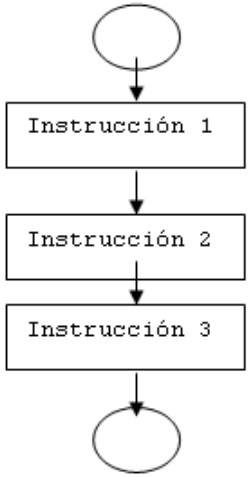
Git branch –m master main
Git remote add origin https



2. Fundamentos de FreeRTOS

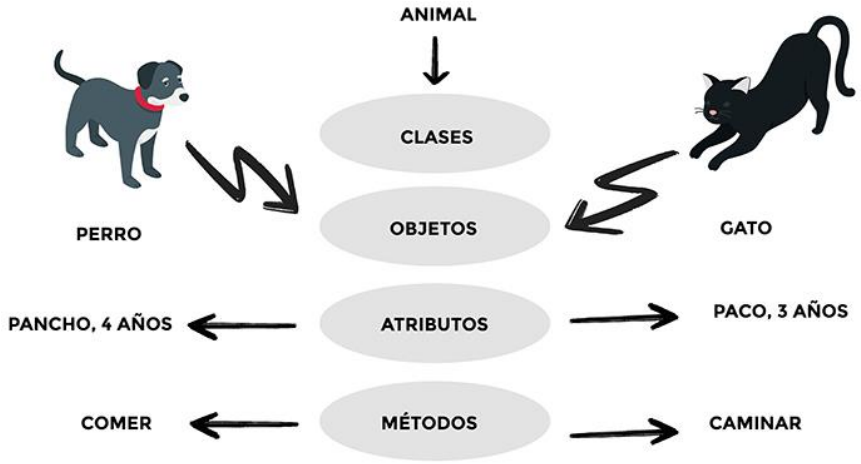
Preguntas de introducción

- Tipos de métodos de programación y en que consisten cada uno “Estructurada, orientada a objetos, por eventos”
- ¿Cuál es la diferencia entre un script y un binario?
- ¿Qué es un lenguaje de etiquetas?

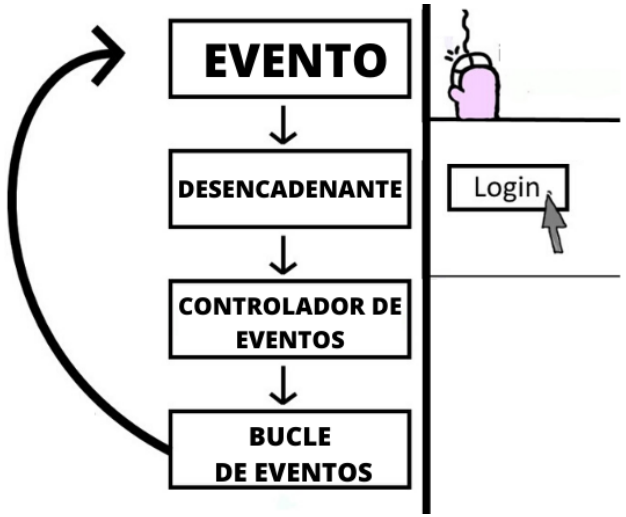



```
Inicio

Instrucción 1
Instrucción 2
      .
      .
      .
Instrucción n

fin
```

Programación estructurada



Programación orientada a objetos



Programación orientada a eventos

```
 GNU nano 2.2.6                File: /home/sofia/scripts/ejemplo.s

#!/bin/bash
ruta=/home/sofia/ejemploRes
arch=/home/sofia/ejemploRes/respaldo.sql
if [ -d $ruta ]; then
        echo "Si existe"
else
        mkdir $ruta
```

Script

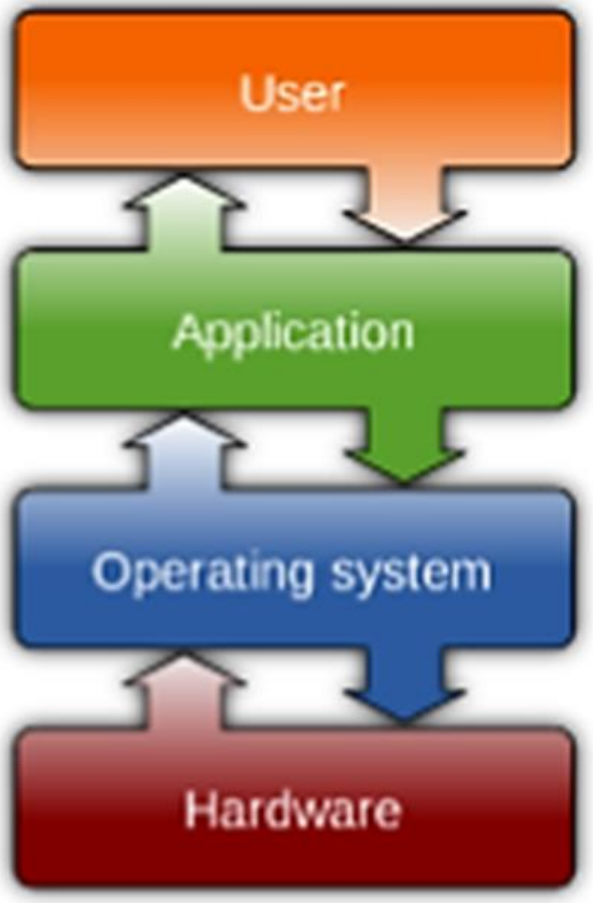

```
<HTML>
<!-- este es un comentario -->
<HEAD>
    <TITLE>Título de la página</TITLE>
</HEAD>
  <BODY>
    <ETIQUETA propiedad="valor" propiedad2="valor">
      <H1>Título</H1>
      <P>Este es un párrafo
    </ETIQUETA>
  </BODY>
</HTML>
```

Lenguaje de etiquetas

Definición de sistema operativo

Un sistema operativo (S.O.) es una colección de elementos de software que gestiona los recursos entre las aplicaciones de software y los elementos de hardware de un dispositivo. Algunas de las principales funcionalidades de las que se encarga son las siguientes:

- Manejo de memoria: permiten el alojamiento de recursos de memoria a los diferentes programas por ejecutar, emplean normalmente memorias virtuales.
- Manejo de procesos: permite el diseño del software a través de diferentes pedazos de programas, cada uno independiente entre sí, denominados procesos (en RTOS se le llaman procesos).
- Manejo de dispositivos: compuesto por un framework que permite organizar y acceder a los distintos recursos de hardware del dispositivo.
- Manejo de archivos: a través del nombramiento, organización, protección e intercambio de archivos.
- Manejo de red
- Seguridad: protege la información de un dispositivo contra amenazas y accesos no autorizados.



Tipos de sistemas operativos

De propósito general (GPOS)

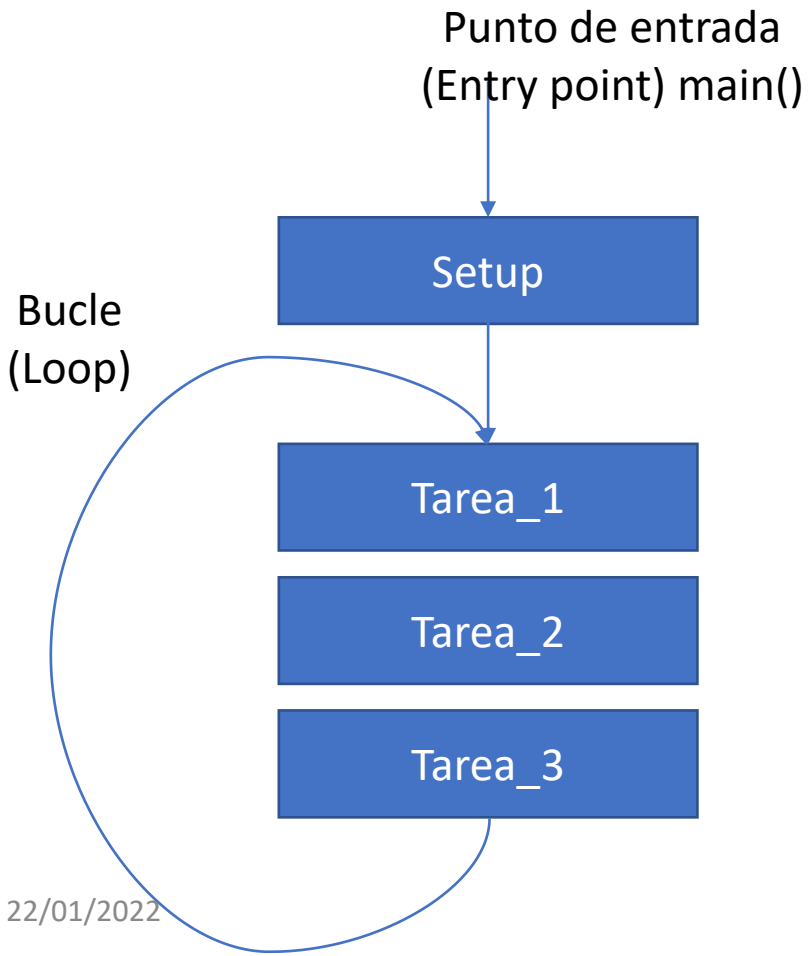
- Ejecución de varios procesos en segundo plano al mismo tiempo.
- Mapeo de memoria dinámica.
- Ejecuciones aleatorias.
- Algunas tareas pueden perder algunos plazos de tiempo y/o pequeños retrasos en la capacidad de respuesta que un humano no se percate de ello.
- Tiempos de respuesta no garantizados.
- Ejemplo: Microsoft Windows, macOS, Ubuntu, Android.
- Usos: Computadoras de escritorio, tablets, celulares, etc.

De tiempo real (RTOS)

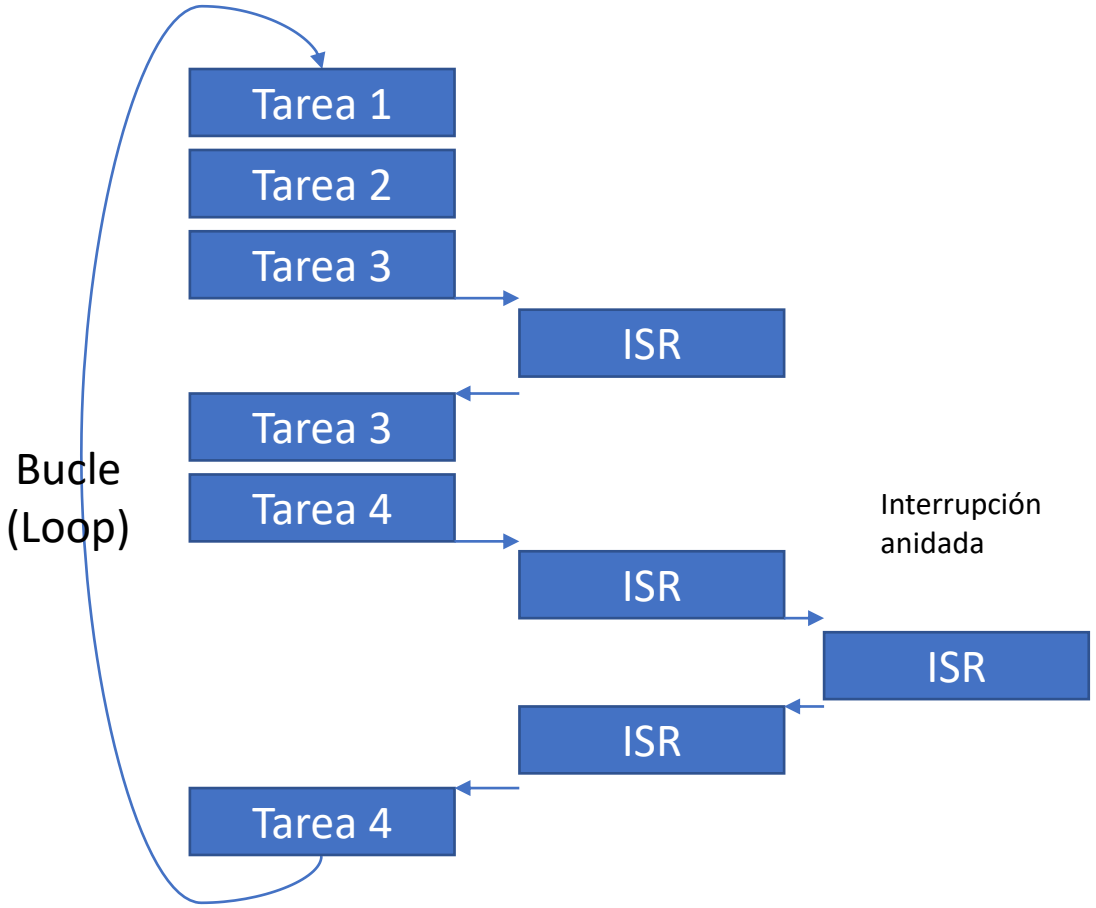
- Diseñado para que un desarrollador pueda garantizar los plazos de tiempo de cada tarea.
- Ejecución de tareas de forma determinístico (sin ejecuciones aleatorias)
- Tiempos de respuesta predictivos.
- Limitados en tiempo.
- Ejemplos: Contiki, FreeRTOS, Zephyr.
- Usos: Sistemas embebidos.

Implementaciones comunes de tareas en un microcontrolador

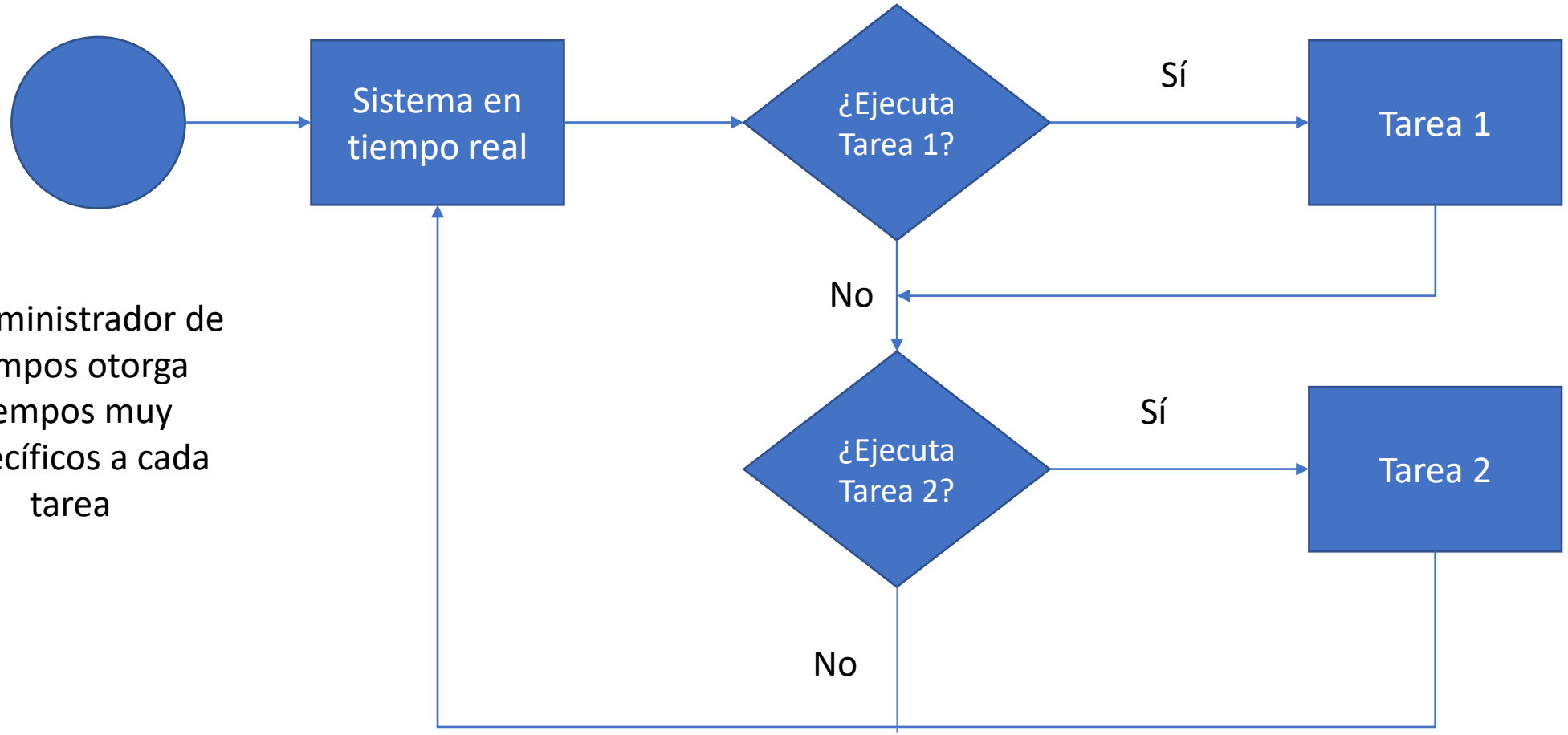
Basado en ciclos infinitos



Basado en ciclos infinitos con interrupciones



Implementación de tareas en RTOS

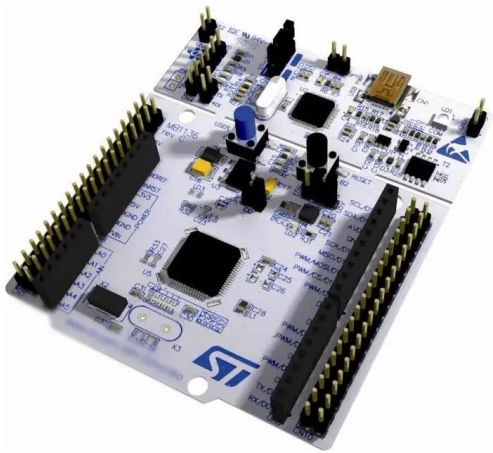


El Administrador de tiempos otorga tiempos muy específicos a cada tarea

Comparaciones de especificaciones técnicas entre tarjetas de desarrollo

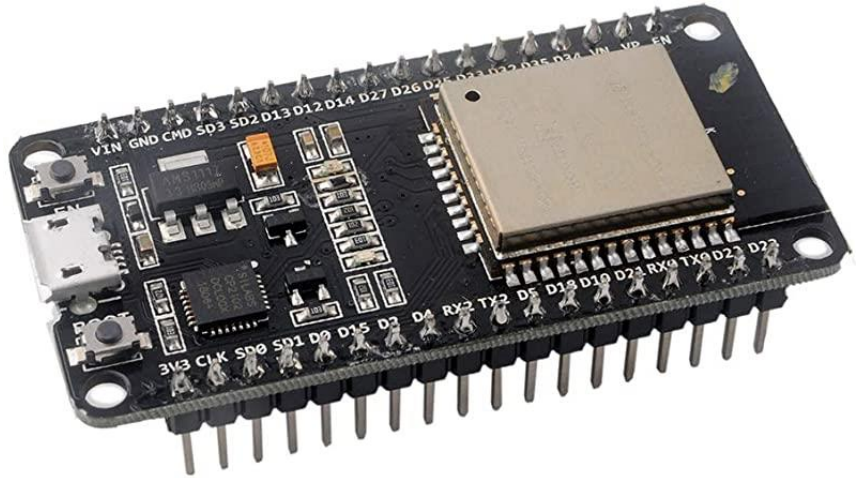
Arduino UNO

ATmega328P
16 MHz
32kb flash
2 kB RAM



STM32L476RG

80 MHz
1Mb flash
128 kB RAM

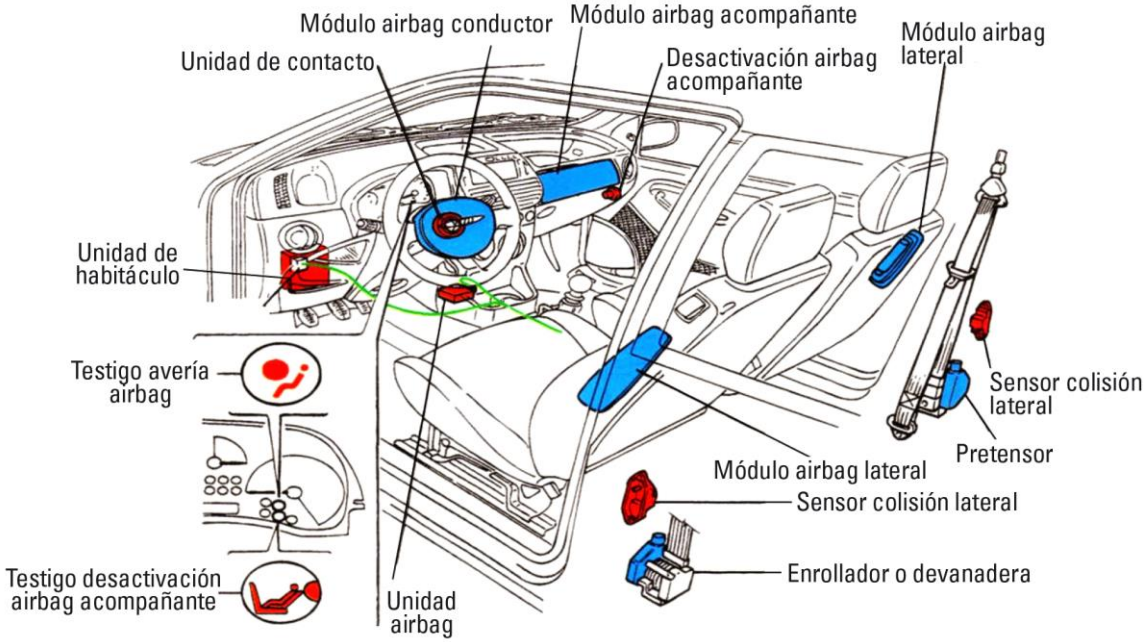


ESP-WROOM-32

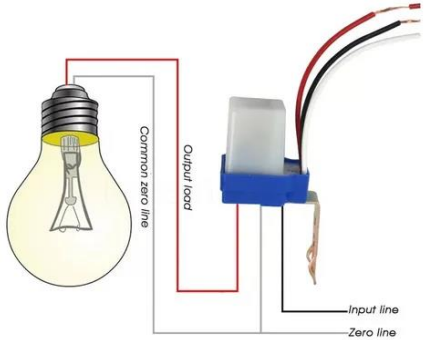
240 MHz (dual core)
4 Mb flash
520 kB RAM

Ejemplo de aplicación

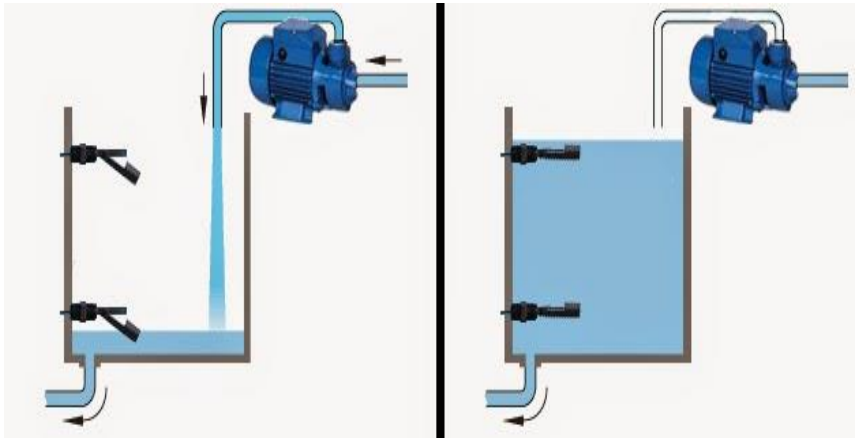
Ejemplo 1



Ejemplo 2



Ejemplo 3



Ejemplo 4

Características de los RTOS

Ventajas

- Puede emplearse en sistemas embebidos gracias al bajo espacio en memoria requerido con respecto a un GPOS.
- Permite la funcionalidad del sistema en conjunto con rutinas de servicio de interrupción (ISR, por sus siglas en inglés)
- Posee una estructura sólida para la comunicación entre tareas mediante el uso de colas, así como la sincronización entre ellas mediante el uso de semáforos y grupos de eventos.
- Brinda escalabilidad gracias a su estructura modular debido a la independencia entre las diferentes tareas.
- Permite reutilizar código para otras aplicaciones, debido a que parte de la arquitectura RTOS.
- Deslinda al planificador la responsabilidad sobre el manejo del tiempo de ejecución de las tareas.

Desventajas

- El Consumo de tiempo cada que el procesador debe determinar y cambiar la tarea por ejecutar a continuación.
- El costo en términos de memoria de código para implementar las funcionalidades RTOS, junto con el consumo de memoria RAM por cada tarea generada.
- Requiere de un análisis de tiempo de ejecución más riguroso dado que ahora es el planificador quien se encarga de esta función y ya no el programador en sí.

¿Qúe es FreeRTOS?

- FreeRTOS es un sistema operativo en tiempo real de código abierto para microcontroladores que facilita la programación, la implementación, la protección, la conexión y la administración de los dispositivos de borde pequeños y de bajo consumo.
- FreeRTOS, distribuido de forma gratuita con la licencia de código abierto MIT, incluye un kernel y un conjunto de bibliotecas de software en crecimiento apto para su uso en todos los segmentos y aplicaciones del sector.
- FreeRTOS esta diseñado con un énfasis en la fiabilidad y la facilidad de uso, y ofrece la predictibilidad de las versiones de soporte de largo plazo.
- El kernel maneja varios objetos como son las tareas, pilas, semáforos, notificaciones y grupos de eventos que permiten la comunicación entre tareas, sincronización entre tareas e interrupciones, y la definición de secciones críticas.

¿Por qué RTOS?

- Requiere un bajo espacio de memoria (En algunas implementaciones, Linux requiere un mínimo de 200MB de memoria libre).
- Garantiza la ejecución de tareas en un plazo de tiempo.
- Implementación rápida de proyectos, ya que la gestión de tiempos ya esta implementada, mediante el planificador.
- Implementación de múltiples tareas independientes entre si. Brinda escalabilidad gracias a su estructura modular.
- Se puede procesar un flujo de datos de comunicaciones (De red, USB,etc).
- Uso en equipo Médico, Estaciones meteorológicas y sismológicas, Despliegue de Satélites.

Principales bibliotecas de FreeRTOS

FreeRTOS es sumistrado bajo archivos fuentes basados en C estandar por lo que se puede usar en otros proyectos basados en C. Los archivos fuentes de FreeRTOs se distribuyen a partir de un archivo zip (previamente instalados).

La pagina de RTOS source code organisation describe la estructura de los archivos dentro del zip.

En cada proyecto, se deben incluir como minimo las siguientes bibliotecas:
•FreeRTOS/Source/tasks.c
•FreeRTOS/Source/queue.c
•FreeRTOS/Source/list.c
•FreeRTOS/Source/portable/[compiler]/[architecture]/port.c.
•FreeRTOS/Source/portable/MemMang/heap_x.c donde 'x' es 1, 2, 3, 4 or 5.

Notación de variables

- *TickType_t* tipo de dato utilizado como contador de *Ticks* y para especificar tiempos, este puede seleccionarse para ser de tipo no signado a 16 bits (1) o no signado a 32 bits (0) de acuerdo con el valor *configUSE_16_BIT_TICKS* dentro de *FreeRTOSConfig.h*.
- *BaseType_t* el tipo de dato más eficiente para la arquitectura, conformado por 32 bits, 16 bits u 8 bits de acuerdo con la arquitectura del procesador a utilizar. Generalmente se utiliza como valor de retorno con valores pdPASS o pdFAIL.

ulTaskCode	

Indicador
u: no signado
p: apuntador
	Tipo de variable
	`c: char`
	`s: int16_t (short)`
	`l: int32_t (long)`
	`x: BaseType_t` u otra estructura no definida (estructuras, tareas, pilas, etc.)

usQueueRecieve()		
Indicador **u: no signado** **p: apuntador**	Valor de retorno *c: char* *s: int16_t (short)* *l: int32_t (long)* *x: BaseType_t* u otra estructura no definida (estructuras, tareas, pilas, etc.) v: vacío (void)	Archivo dentro del cual se encuentra definida la función

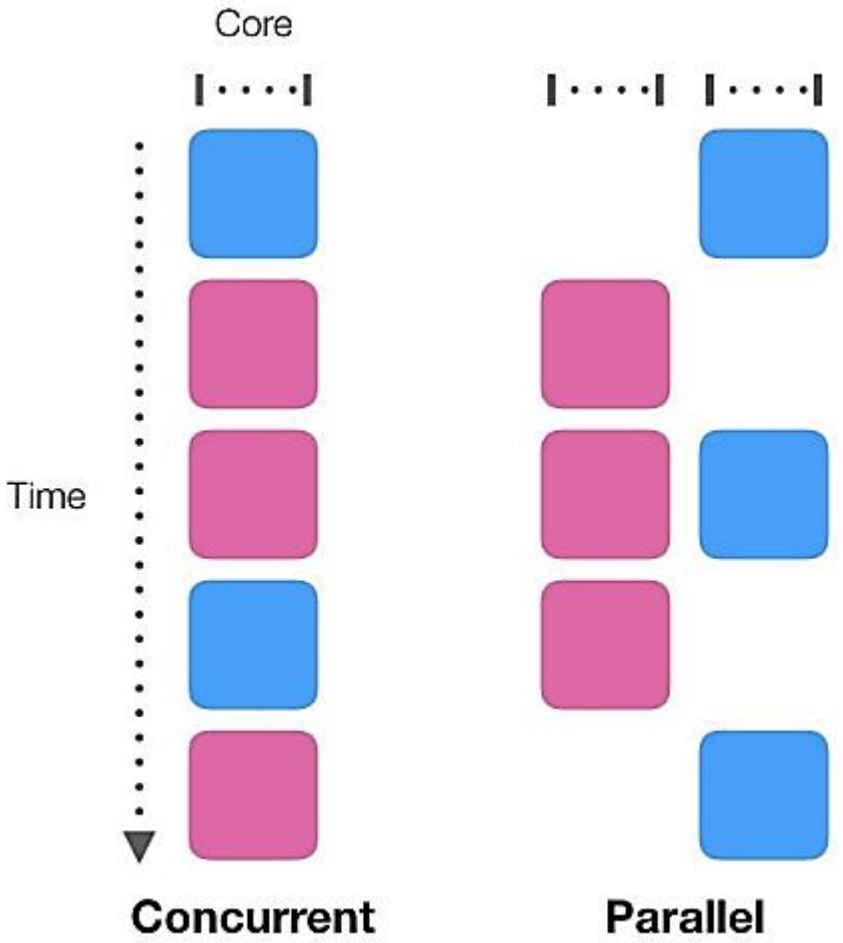
Objetos kernel

- Tareas
- Colas
- Semáforos
  - Mutex
- Notificaciones
- Grupos de eventos

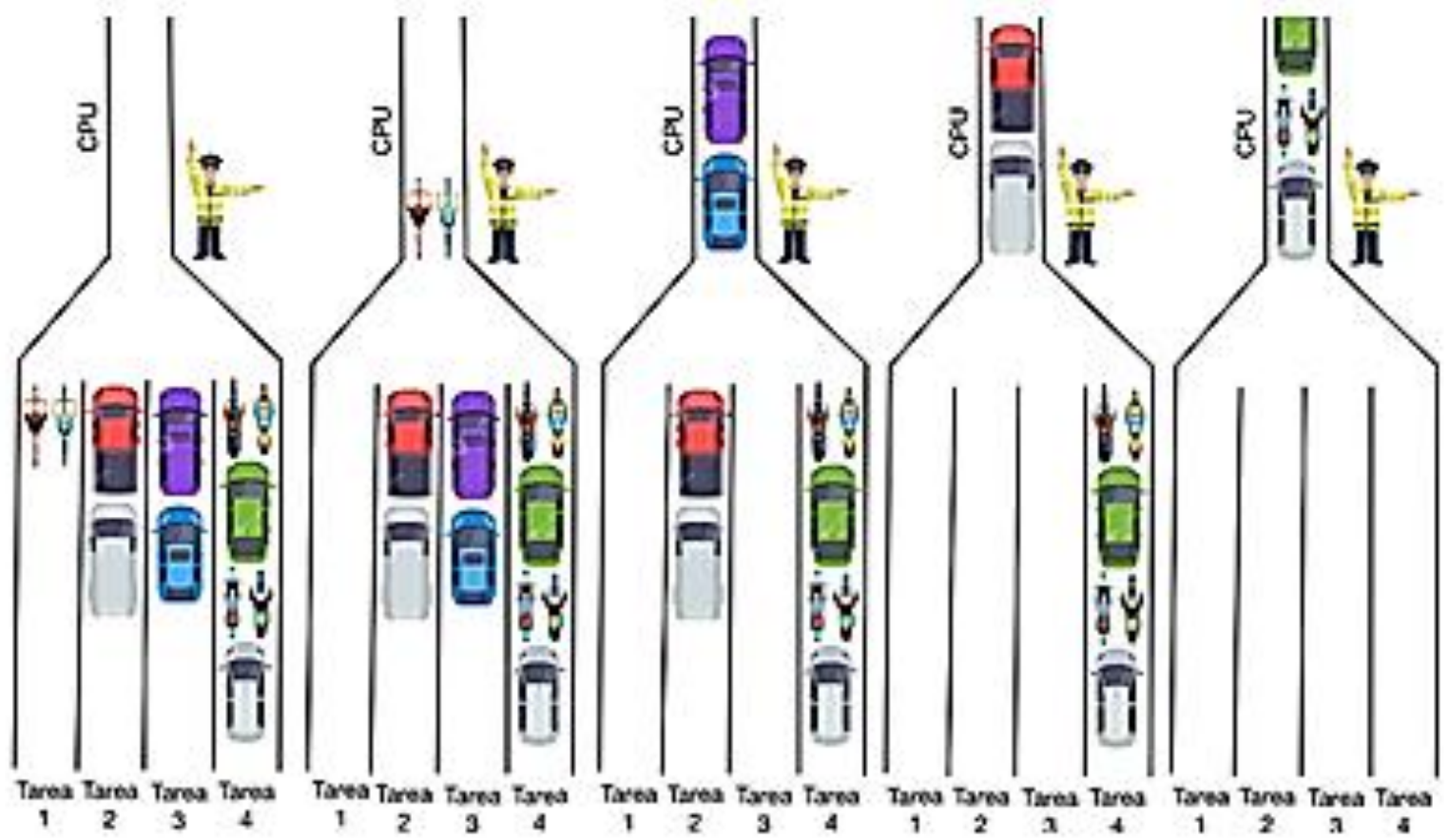
Comunicación, sincronización e interrupción de tareas. Definición de secciones criticas 0

Kernel: Capa principal entre el sistema operativo y el hardware encargado de administrar el manejo de memoria y procesos

Manejo de procesos

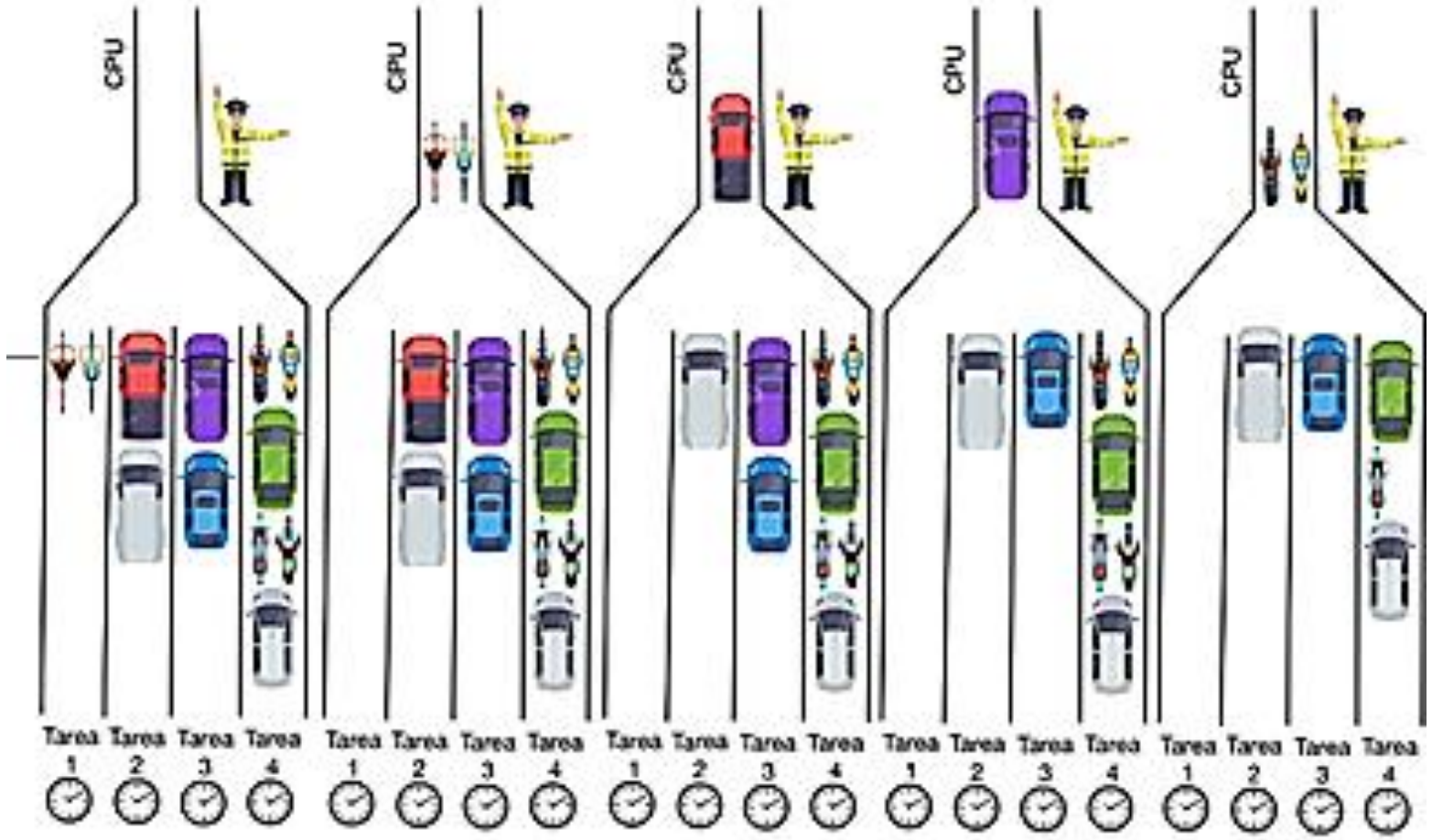


Paralelo



Oficial de transito -> Planificador
Carril -> Proceso (Tarea)
Vehículos -> Carga computacional

Concurrente



Oficial de transito -> Planificador
Carril -> Proceso (Tarea)
Vehículos -> Carga computacional

Tareas

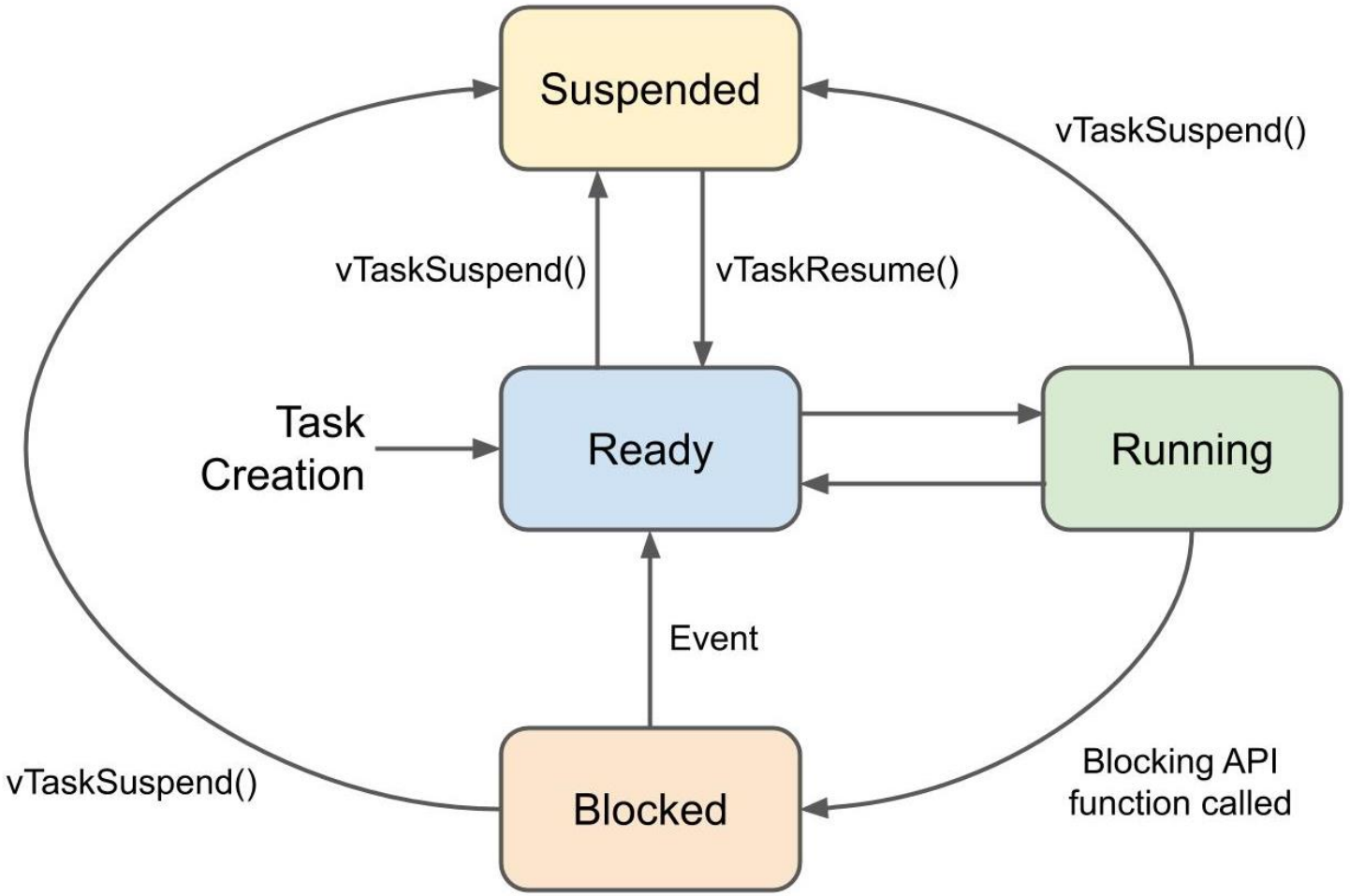
Funciones cuya estructura, poseen los siguientes elementos: (thread o parte del cpu)

- Contador de programa
- Stack, Valores de registro
- Planificador de tareas

Tipos de tareas

- Tareas periódicas: aquellas cuya ejecución es realizada de forma constante al finalizar un lapso determinado que a su vez corresponde a su tiempo límite.
- Tareas aperiódicas: aquellas cuya ejecución es espontánea pero no cuentan con un tiempo límite.
- Tareas esporádicas: aquellas cuya ejecución es espontánea pero sí cuentan con un tiempo límite.

Estados de una tarea



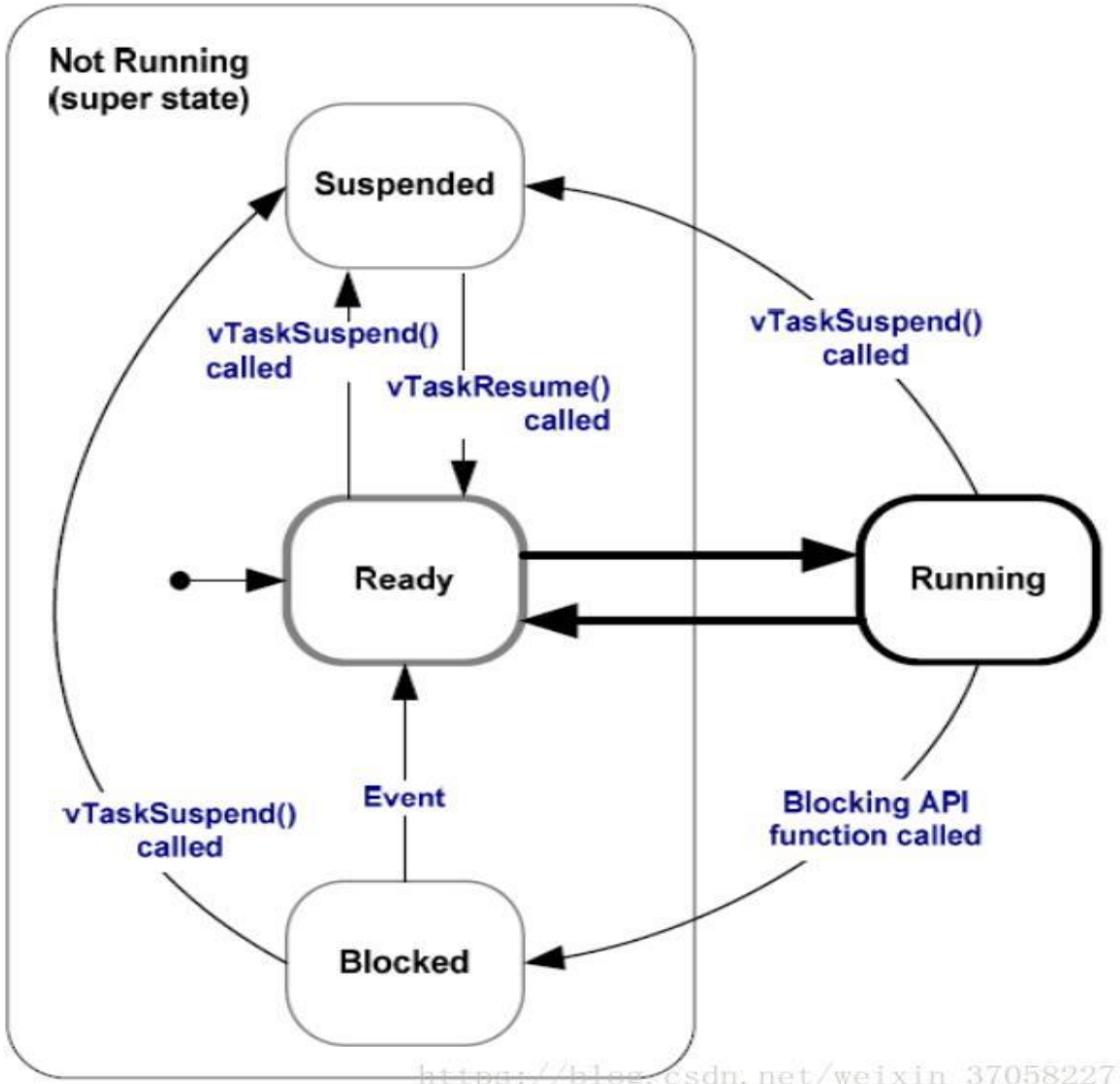
- Listo: sucede cuando alguna otra tarea está ocupando el procesador, pero nuestra tarea aún cuenta con actividades por realizar en cuanto éste se encuentre disponible. Normalmente, una gran variedad de tareas se encuentra en este estado.

- Bloqueado: implica que nuestra tarea ya no tiene ningún trabajo que hacer por el momento, incluso si el procesador se encuentra disponible. Las tareas en este estado se encuentran esperando a que se suscite algún evento externo. Igualmente, una gran variedad de tareas suele encontrarse en este estado. Las tareas pueden entrar a este modo por dos razones:

- Eventos síncronos: definidos a permanecer en este estado por un periodo establecido de tiempo. En FreeRTOS esto puede lograrse mediante la función vTaskDelay() que coloca a una tarea en este estado durante un número determinado de Ticks, no obstante, esta función solo se encuentra disponible al habilitar con 1 la variable INCLUDE_vTaskDelay en el archivo "FreeRTOSConfig.h".

- Eventos asíncronos: se originan a través de una interrupción o mediante la invocación por una tarea. Elementos como las pilas, los semáforos, los grupos de eventos y los mutex pueden utilizarse para conformar eventos asíncronos o síncronos.

- Suspendido: implica que la tarea sigue habilitada pero ya no se encuentra disponible para el planificador. La única manera de acceder a este estado es mediante la invocación de la función en FreeRTOS vTaskSuspend() y la única forma de salir de ella es mediante la función vTaskResume(). Este estado es raramente empleado.



Declaración como función del contenido de una tara en FreeRTOS

```
void ATaskFunction( void *pvParameters )
{
        /* Variables can be declared just as per a normal function.
        * Each instance of a task created using this example function
        * will have its own copy of the lVariableExample variable. This
        * would not be true if the variable was declared static - in
        * which case only one copy of the variable would exist, and
        * this copy would be shared by each created instance of the
        * task. */
        int32_t lVariableExample = 0;
        /* A task will normally be implemented as an infinite loop. */
        for( ;; ){
        /* The code to implement the task functionality will go here. */
        }
        /* Should the task implementation ever break out of the above
        loop, then the task must be deleted before reaching the end
        * of its implementing function. The NULL parameter passed to
        * the vTaskDelete() API function indicates that the task to be
        * deleted is the calling (this) task. */
        vTaskDelete( NULL );
}
```

https://www.freertos.org/a00125.html

Creación de una tarea (en el main)

Declaración:

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        uint16_t usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask );
```

Parámetros:

- **pxTaskCode:** Tarea (función escrita por el usuario, generalmente un bucle infinito)
- **pcName**: Nombre de la tarea (utilizado para el seguimiento y la depuración, la longitud máxima no excede configMAX_TASK_NAME_LEN) pcName
- **usStackDepth**: El tamaño de la pila de tareas en bytes(nota: el tamaño de la pila solicitada es 4 veces el parámetro, que se obtiene mediante definiciones de macros)
- **pvParameters**: Parámetro pasado a la tarea (normalmente NULL)
- **uxPriority**: Prioridad de la tarea UxPriotiry uxPriotiry (máximo y mínimo no deseable, obtenido a través de la definición de macro)
- **pxCreatedTask** Identificador de tarea (identificador de tarea, el identificador de tarea de esta tarea se devolverá después de que la tarea se haya creado correctamente. Este identificador es en realidad la pila de tareas de la tarea. Este parámetro se utiliza para guardar este identificador de tarea y puede ser utilizado por otras funciones de la API).

Valor de retorno:

- pdPASS: se creó con éxito.
- errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY: Falló la creación de la tarea debido a que no hay suficiente memoria de pila

https://programmerclick.com/article/16381319627/

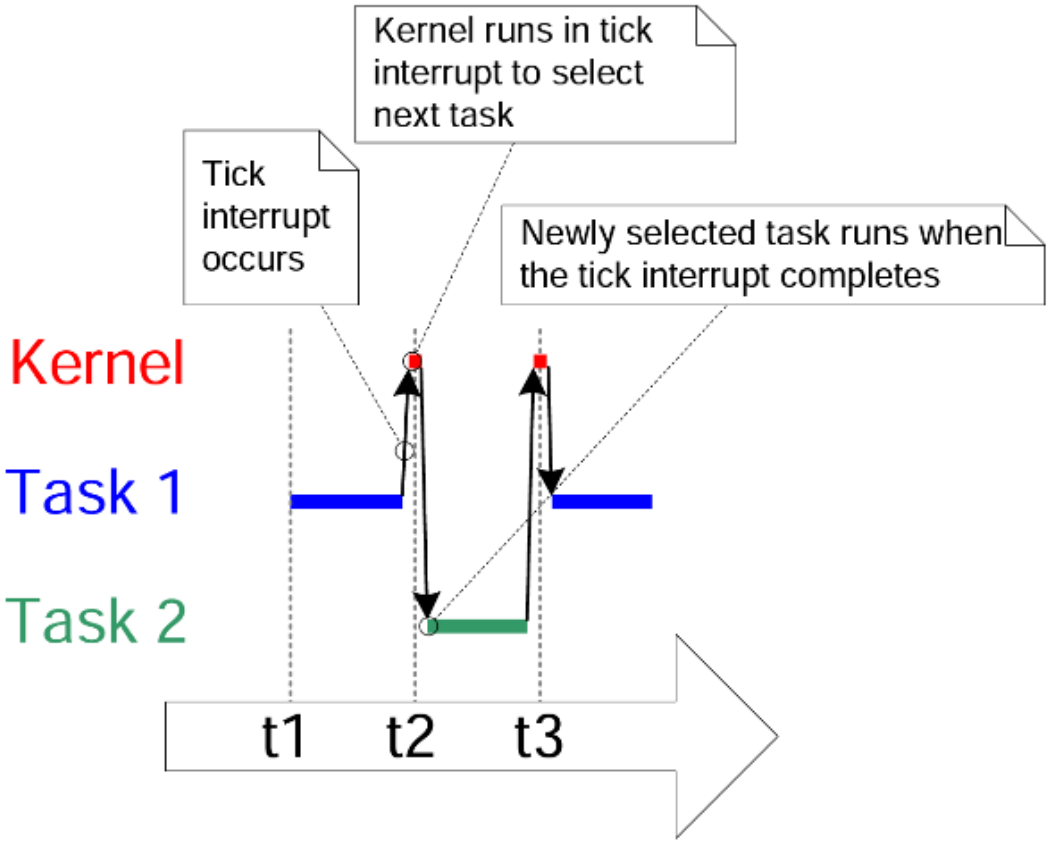
Resumen de funciones asociadas a las tareas

Nombre	Descripción	Código
xTaskCreate()	Creación de una tarea	
vTaskDelay()	Coloca a una tarea en el sub-estado “Bloqueado” por un numero fijo de Ticks **Parámetro**: xTicksToDelay: Número de ticks bloqueada	`void vTaskDelay( const TickType_t xTicksToDelay );`
vTaskResume()	Salida del sub-estado “Suspendido” **Parámetros:** xTaskToResume: identificador de tarea que salide de “Suspendido”	`void vTaskResume( TaskHandle_t xTaskToResume );`
vTaskSuspend()	Acceso al sub-estado “Suspendido” **Parámetro:** xTaskToSuspend: identificador de tarea que se suspende	`void vTaskSuspend( TaskHandle_t xTaskToSuspend );`
vTaskPrioritySet()	Cambio del nivel de prioridad a una tarea **Parámetros:** TaskHandle_t: Identidicador de la tarea que modifica su prioridad UBaseType_t: Valor de la nueva prioridad	`void vTaskPrioritySet( TaskHandle_t xTask, UBaseType_t uxNewPriority );`
vTaskStartScheduler()	Pone en funcionamiento el planificador junto con la tarea Idle	`void vTaskStartScheduler( void );`
vTaskDelete()	Eliminación de una tarea **Parámetro:** xTask: identificador de la tarea a eliminar	`void vTaskDelete( TaskHandle_t xTask );`

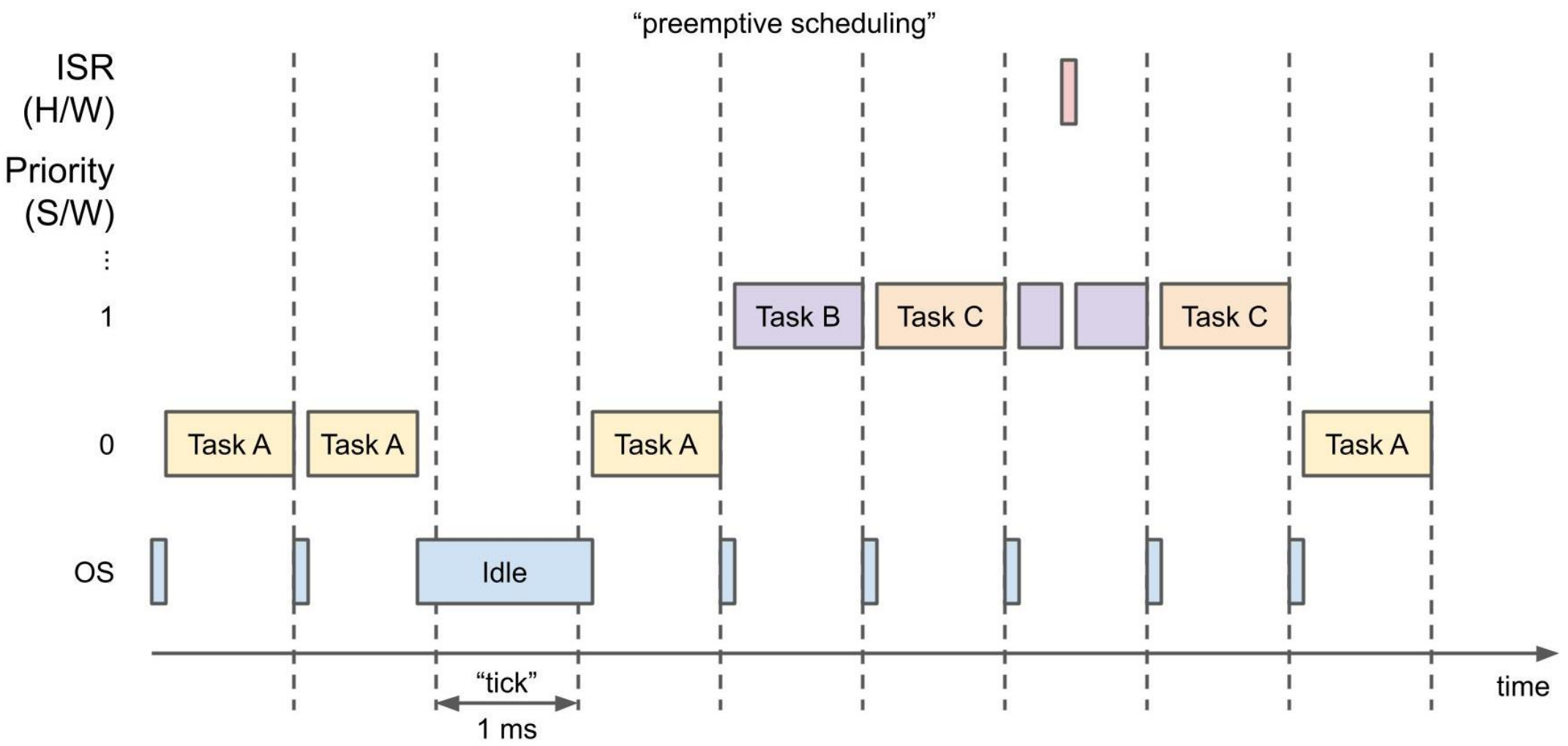
https://www.freertos.org/a00106.html

Planificador de tareas

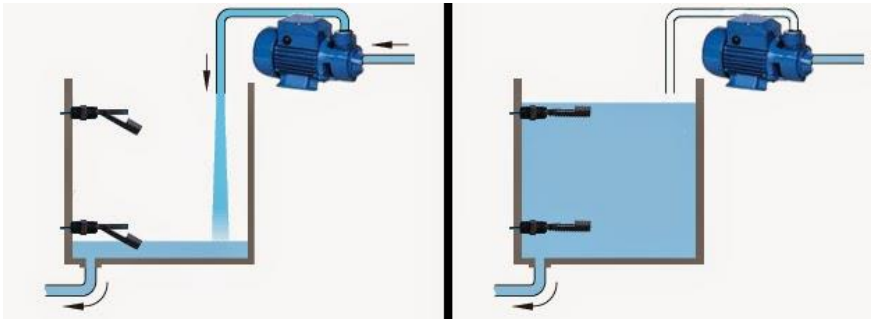
- Ejecuta las diferentes tareas en un RTOS y permite una programación en concurrencia, funciona mediante un algoritmo apropiativo.
- Las diferentes tareas que conforman al programa alternan el uso del procesador. A esta acción se denomina cambio de contexto (acompañado de un pequeño consumo de tiempo).



*assuming single-core processor



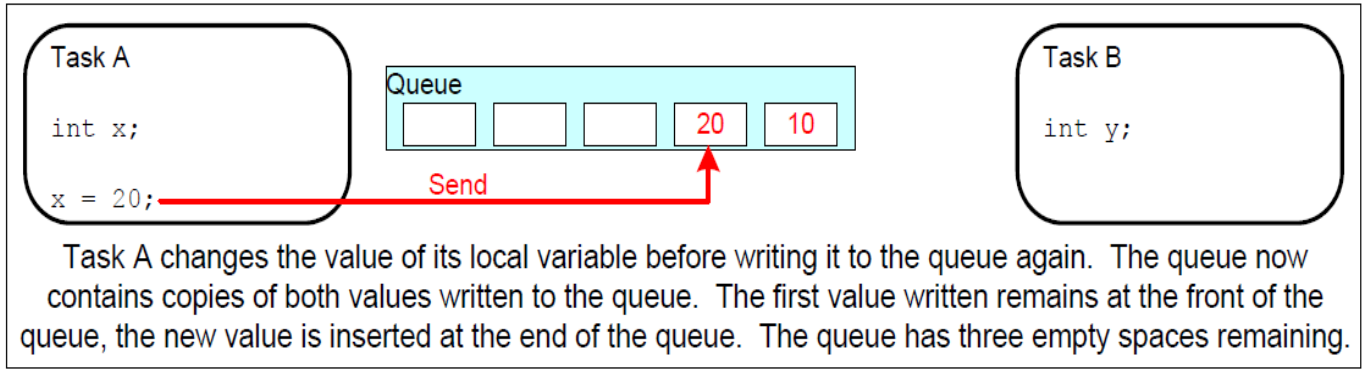
Planteamiento de problema a resolver



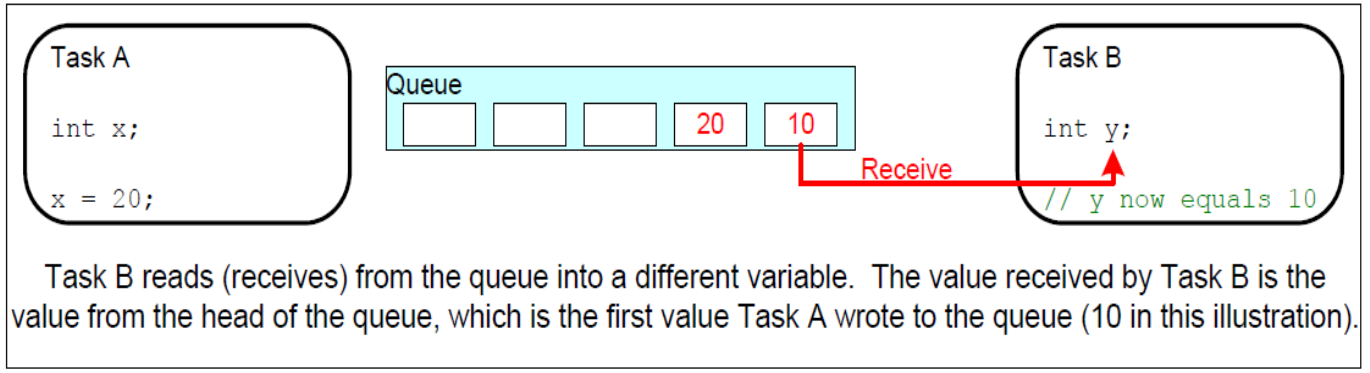
Tienes que diseñar dos tareas, una de llenado y otra de suministro de medicamento para un dispensador de insulina.

Colas

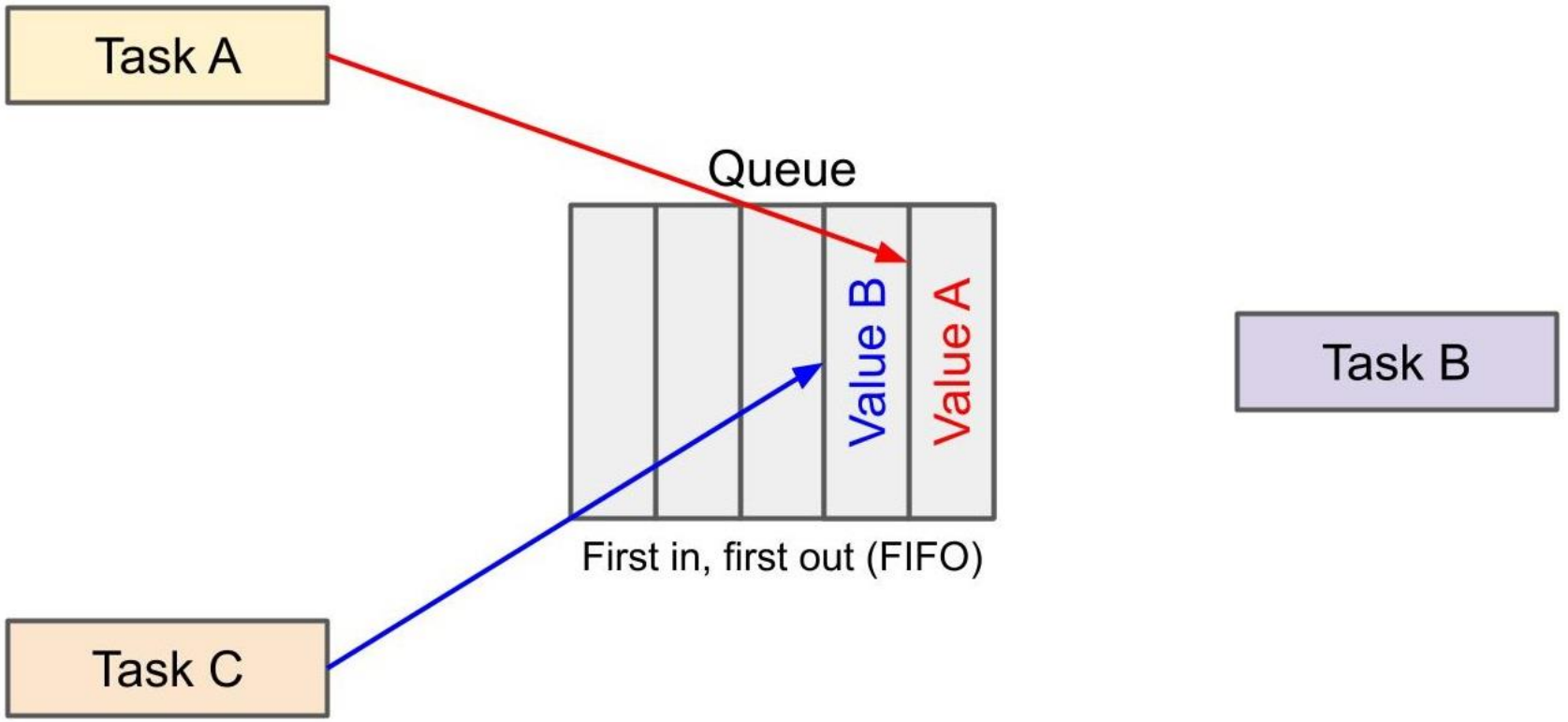
- Las colas son un mecanismo independiente que permite la comunicación entre tareas, de una tarea a una interrupción o de una interrupción a una tarea.
- Estas adquieren normalmente la forma de un buffer First In First Out (FIFO) que poseen una extensión fija al crearse dentro del programa.
- La operación de las colas en FreeRTOS es mediante la copia de elementos, por lo que permite el desacople de las variables origen que se depositan en ella y las variables destino que las recogen.



Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.



Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).


Task A
Queue
Value B
Value A
Task B
First in, first out (FIFO)
Task C

Creación de un cola

Declaración:

```
QueueHandle_t xQueueCreate(
                              UBaseType_t uxQueueLength,
                               UBaseType_t uxItemSize );
```

Parámetros:

* **uxQueueLength:** Número máximo de elementos que la cola puede contener

* **uxItemSize:** Tamaño en bytes de cada uno de los elementos que puede alamacenar la cola

Valor de retorno:

*Return: NULL

Creación de Escritor

Declaración:

```
BaseType_t xQueueSend(
        QueueHandle_t xQueue,
        const void * pvItemToQueue,
        TickType_t xTicksToWait );
```

Parámetros:

* **xQueue**: identificador de la cola a la cual se va a enviar la información.

* **pvItemToQueue:** Puntero de la información que se va escribir dentro de la cola.

*** xTicksToWait:** Cantidad máxima ticks que la tarea tiene que esperar en el estado “Bloqueado” hasta que la cola tenga un espacio disponible

Valor retorno:

* pdPASS: Si la información se escribio de manera correcta

* errQUEUE_FULL Si la informacion no se pudo escribir

Lector

Declaración:

```
BaseType_t xQueueReceive(
        QueueHandle_t xQueue,
        void * const pvBuffer,
        TickType_t xTicksToWait );
```

Parámetros:

* xQueue: Identificador de la cola a la cual se ve a leer la información.

* pvBuffer: Un puntero de un espacio de memoria a la cual se va a copiar la información recibida.

* xTicksToWait: Cantidad máxima ticks que la tarea tiene que esperar en el estado “Bloqueado” hasta que la cola tenga un información disponible

Valor retorno:

pdPASS:Si la información se lee de manera correcta.

errQUEUE_EMPTY: Si la información no se lee de manera correcta.

Resumen de funciones asociadas a las colas

Nombre	Descripción	Código
xQueueCreate()	Creación de una cola.	`QueueHandle_t xQueueCreate( UBaseType_t uxQueueLength, UBaseType_t uxItemSize );`
xQueueSendtoBack()/ xQueueSend()	Escribe un dato al final de la cola.	`BaseType_t xQueueSendToBack( QueueHandle_t xQueue, const void * pvItemToQueue, TickType_t xTicksToWait );`
xQueueSendtoFront()	Escribe un dato al frente de la cola	`BaseType_t xQueueSendToFront( QueueHandle_t xQueue, const void * pvItemToQueue, TickType_t xTicksToWait );`
xQueueReceive()	Recoge el dato al frente de la cola.	`BaseType_t xQueueReceive( QueueHandle_t xQueue, void *pvBuffer, TickType_t xTicksToWait );`
uxQueueMessagesWaiting()	Consulta del número de elementos inertes en la cola **Parámetros:** xQueue: Identificador de la cola **Valor retorno:** El numero de mensajes almacenados en la cola	`UBaseType_t uxQueueMessagesWaiting( QueueHandle_t xQueue );`
xQueueOverwrite()	Sobre escribe el contenido del buzón con un nuevo dato **Parámetros:** xQueue: Identificador de la cola pvItemToQueue: Puntero **Valor retorno:** pdTRUE si se reescribi el valor,	`BaseType_t xQueueOverwrite( QueueHandle_t xQueue, const void * pvItemToQueue );`
xQueuePeek()	Adquiere el elemento del buzón sin eliminarlo del mismo **Parámetros:** xQueue: Identificador de la cola que vamos a leer pvItemToQueue: Puntero donde se va copiar el elemento xTicksToWait: Cantidad de ticks que se bloque la tarea hasta exista un elemento en la cola **Valor retorno:** BaseType_T: pdTrue o pdFalse	`BaseType_t xQueuePeek( QueueHandle_t xQueue, void *pvBuffer, TickType_t xTicksToWait );`

https://www.freertos.org/a00106.html

Semáforos

ISR: Las ISR tiene mayor prioridad que cualquier tarea y dentro de las mismas se puede llevar acaba un cambio de contexto (levantándose una bandera de tarea de alta prioridad)

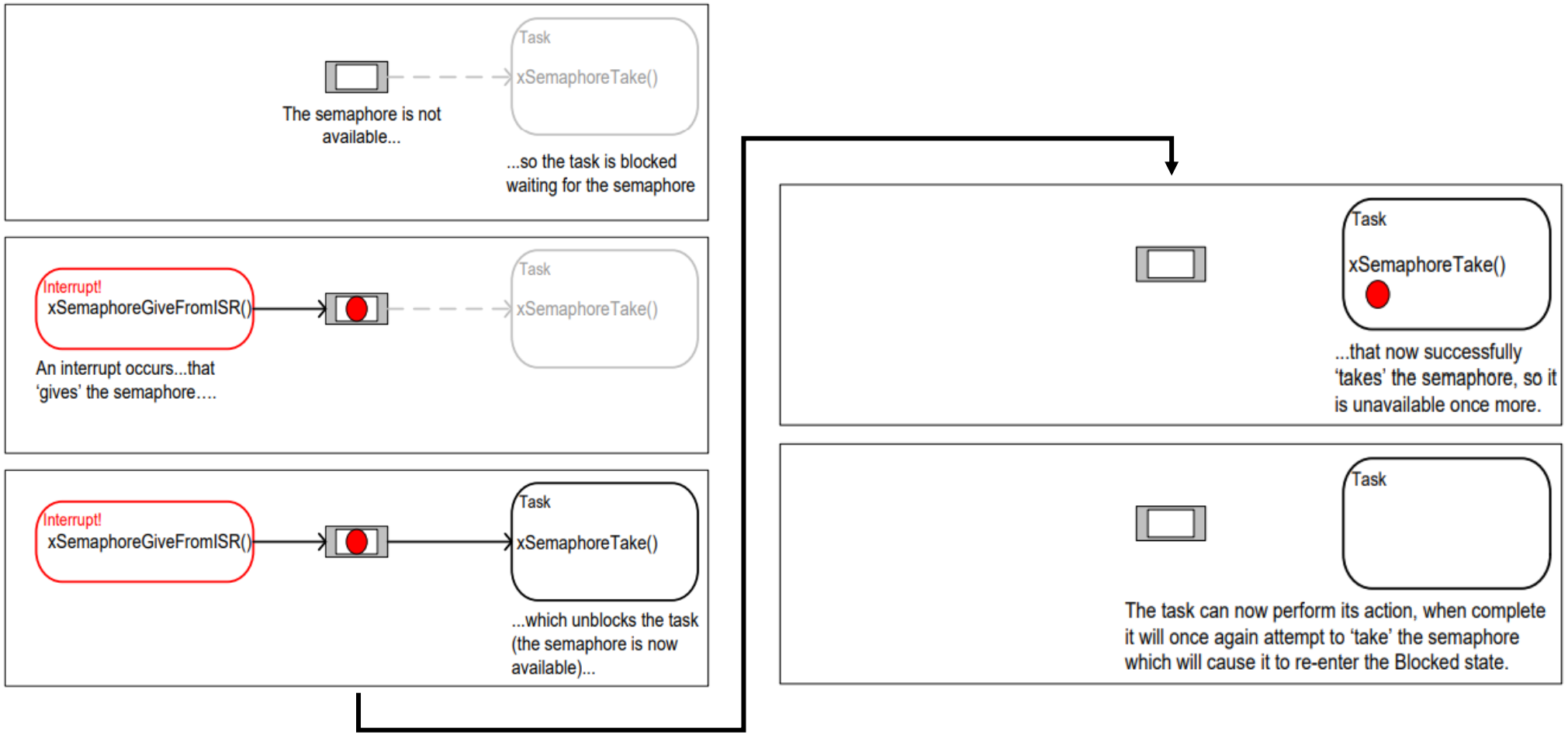
Es necesario duren lo menos posible ya al consumir demasiado tiempo del procesador, se evita la atención de las demás tareas.

Los semáforos son usados para sincronizar tareas.

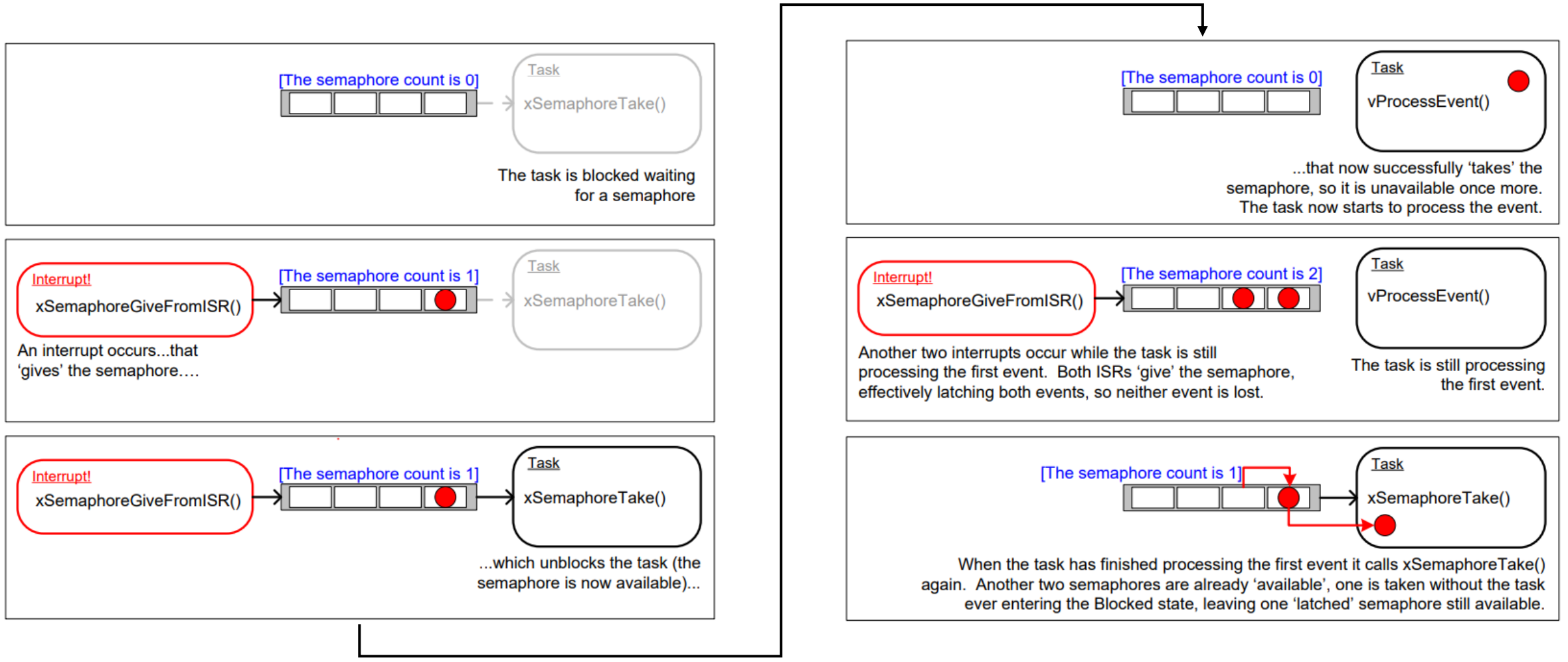
Pueden ser de dos tipos:

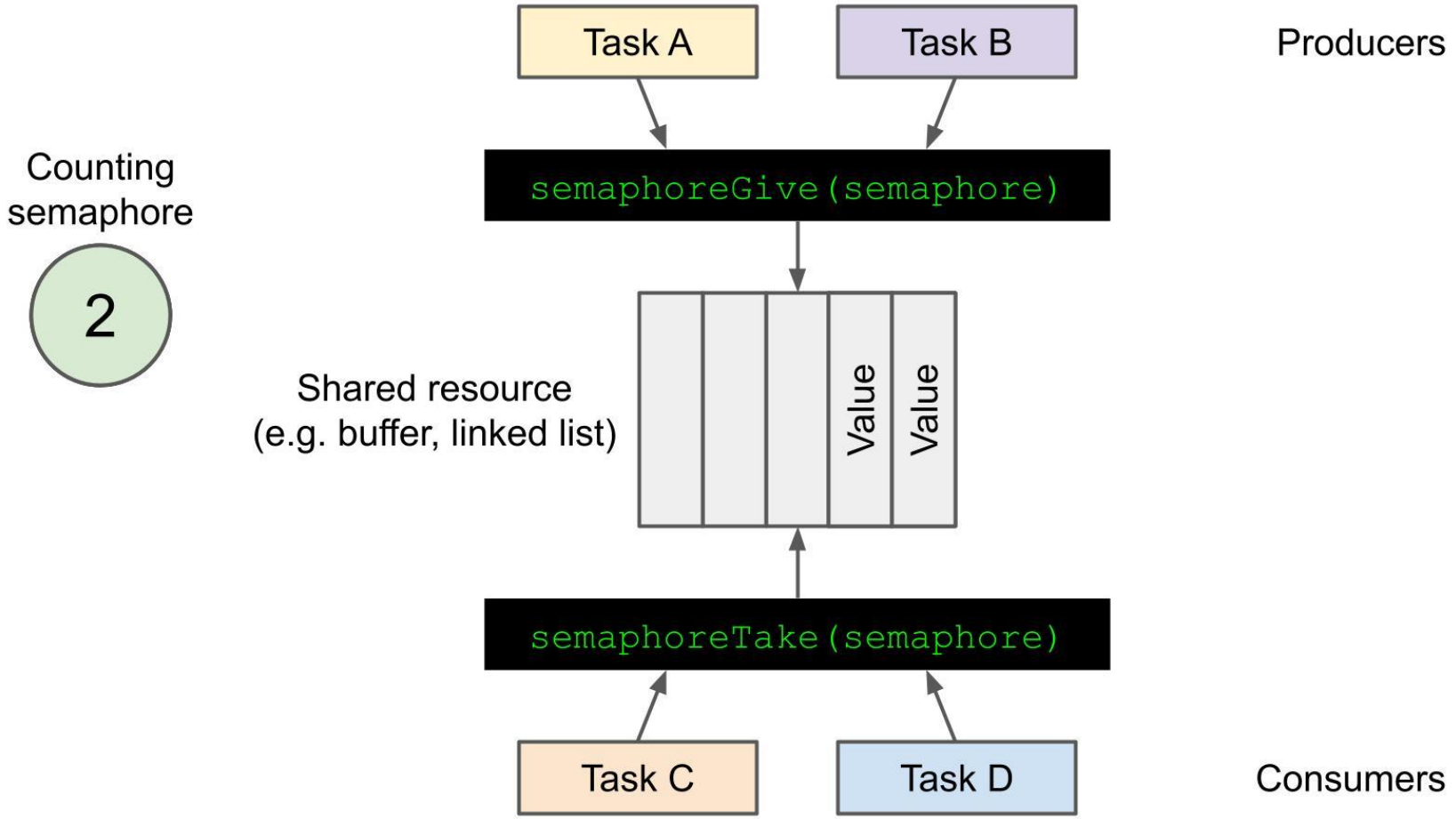
- Binario: Toman dos estados. Cola de longitud 1.
- Como Contador: Pueden tomar varios recursos. Cola de longitud mayor a uno, donde lo importante es la cantidad de elementos dentro de la cola.

Semáforo binario



Semáforo contador (concurrente)




Task A
Task B
Producers
Counting semaphore
2
semaphoreGive(semaphore)
Shared resource (e.g. buffer, linked list)
Value
Value
semaphoreTake(semaphore)
Task C
Task D
Consumers

Consumidor

Declaración:

```
BaseType_t xSemaphoreTake(
        SemaphoreHandle_t xSemaphore,
        TickType_t xTicksToWait );
```

Parámetros:

* xSemaphore: Identificador del semáforo que es tomado.

* xTicksToWait: Maxima cantidad de ticks que la tarea tiene que esperar en el estado “bloquedo” hasta que el semáforo este disponible.

Valor retorno:

* pdTrue: Si el semáforo se pudo obtener.

* pdFalse: Si el semáforo no se pudo obtener.

Productor

Declaración:

```
BaseType_t xSemaphoreGive(
      SemaphoreHandle_t xSemaphore,
      BaseType_t *pxHigherPriorityTaskWoken);
```

Parámetros:

* xSemaphore: Identificador del semáforo que es regresado.

Valor de retorno:

* pdTrue: Si fue entregado.
* pdFalse: Si algun error se presenta.

Resumen de funciones asociadas a los semáforos

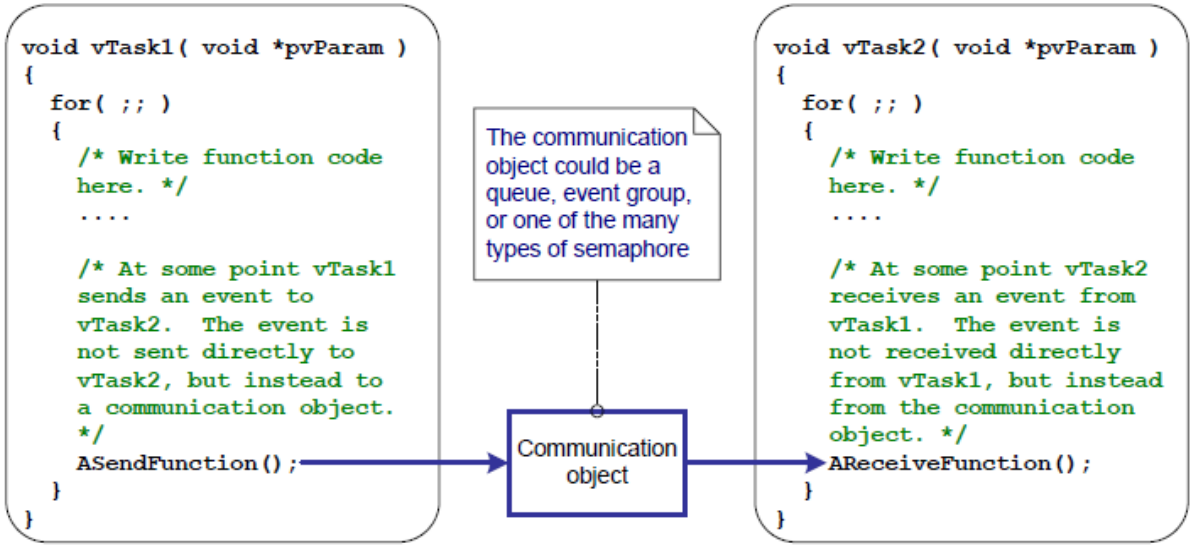
Nombre de función	Descripción	Código
xSemaphoreCreateBinary()	Creación de un semáforo binario Valor de retorno: NULL: Si el semáforo no se pudo crear debido a falta de memoria SemaphoreHandle_t : Indicador a tráves del cual el semáforo va a ser referenciado.	`SemaphoreHandle_t xSemaphoreCreateBinary( void );`
xSemaphoreTake()	Acción de retirar una ficha del semáforo **Parámetros:** * xSemaphore: Identificador del semáforo que es tomado. * xTicksToWait: Maxima cantidad de ticks que la tarea tiene que esperar en el estado “bloquedo” hasta que el semáforo este disponible. **Valor retorno:** * pdTrue: Si el semáforo se pudo obtener. * pdFalse: Si el semáforo no se pudo obtener.	`BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait );`
xSemaphoreGive ()	Acción de colocar una ficha en el semáforo **Parámetros**: * xSemaphore: Identificador del semáforo que es regresado. **Valor de retorno:** * pdTrue: Si fue entregado. * pdFalse: Si algun error se presenta.	`BaseType_t xSemaphoreGive( SemaphoreHandle_t xSemaphore, BaseType_t *pxHigherPriorityTaskWoken);`

https://www.freertos.org/a00106.html

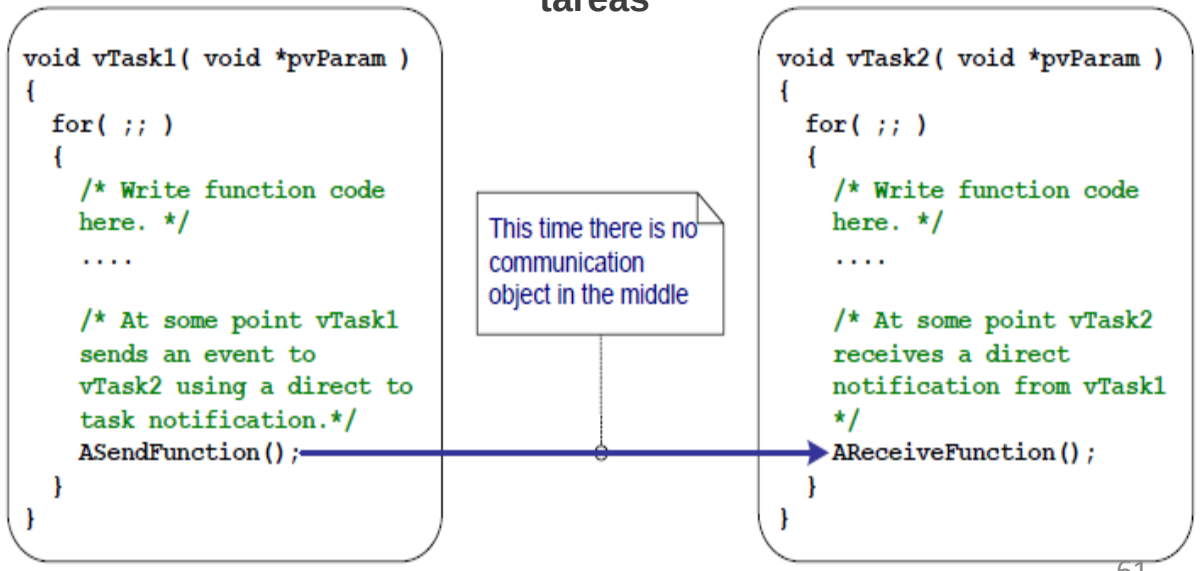
Notificaciones

- En el entorno FreeRTOS existen numerosos mecanismos para permitir la intercomunicación de información entre tareas como han sido las colas y los semáforos.
- Sin embargo, estos elementos suelen requerir de una cantidad considerable de memoria RAM para su operación, así como consumir cierto tiempo del procesador para la atención de cada una de sus funcionalidades. Cada tarea en RTOS tiene una notificación de 32 bits que se inicializa en cero cuando la tarea es creada
- Por lo tanto, existen ciertos elementos denominados “notificaciones” quienes disponen de una rápida ejecución y cuentan con un bajo consumo de memoria RAM de tan sólo 8 bytes por tarea para su funcionamiento (en comparación con el uso de semáforos su uso puede ser hasta 45% más rápido).
- Las notificaciones de tareas permiten tareas, comunicación directa entre interrupciones sin objetos intermedios.
- La notificación de tareas en RTOS es equivalente a enviar un evento directamente a la tarea
- El uso de notificaciones dentro del entorno FreeRTOS es opcional y para habilitar su uso debe asignarse un valor de 1 al parámetro configUSE_TASK_NOTIFICATION dentro del archivo “FreeRTOSConfig.h”. De esta forma las tareas adquieren dos nuevos estados denominados “Pendiente” o “No Pendiente”, junto con un valor de notificación que corresponde a un valor de 32 bits entero no signado.
- Ventajas:
  - Rápido
  - Memoria provisional
- Desventajas
  - No se puede enviar datos a ISR a través de una notificación de mensaje
  - Solo puede contener a un valor
  - La transmisión de la notificación de la tarea no puede ingresar al estado de bloqueo

Comunicación a través del objeto intermedio.



Comunicación directa entre tareas - Notificación de tareas



Implementación de una notificación

Función:

```
BaseType_t xTaskNotify(TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction eAction );
```

Uso:

Envía el valor de notificación especificado a la tarea especificada. Si tiene la intención de usar notificaciones de tareas RTOS para implementar semáforos binarios o contadores livianos, se recomienda la función API xTaskNotifyGive () en lugar de esta función.

Parámetros:

- **xTaskToNotify:** el identificador de tarea que se notifica.
- **ulValue**: notifica el valor de actualización
- **eAction**: tipo de enumeración, que indica el método para actualizar el valor de notificación

Descripción de valores que puede tomar eAction:

Miembro Enum	Descripción
eNoAction	Enviar notificación sin actualizar el valor de notificación, lo que significa que el parámetro ulValue no se usa
eSetBits	El valor de notificación de la tarea que se notifica es bit a bit o en ulValue. El uso de este método puede reemplazar grupos de eventos en algunos escenarios, pero la velocidad de ejecución es más rápida
eIncrement	El valor de notificación de la tarea notificada aumenta en 1. En este caso, el parámetro ulValue no se usa y la función API xTaskNotifyGive () es equivalente a esto.
eSetValueWithOverwrite	El valor de notificación de la tarea notificada se establece en ulValue. Con este método, puede reemplazar xQueueOverwrite () en algunos escenarios, pero se ejecuta más rápido.
eSetValueWithOverwrite	Si la tarea notificada no se notifica actualmente, el valor de notificación de la tarea notificada se establece en ulValue; si la tarea notificada no ha recibido la notificación anterior y ha recibido otra notificación, el valor de la notificación se descarta esta vez. En este caso, La función xTaskNotify () no puede llamar y devuelve pdFALSE. Use este método para obtener xQueueSend () con una longitud de 1 en algunos escenarios, pero es más rápido.

Resumen de funciones asociadas a las notificaciones

Nombre	Descripción	Código
xTaskNotifyGive()	Envía una notificación a una tarea **Párametro** **xTaskToNotify**: el identificador de tarea que se notifica.	`BaseType_t xTaskNotifyGive(` `TaskHandle_t` **`xTaskToNotify`** `);`

ulTaskNotifyTake()	Bloquea una tarea hasta recibir una notificación.
	Parámetro
	xClearCountOnExit: si este parámetro se establece en pdFALSE, la función API xTaskNotifyTake () disminuye el valor de notificación de la tarea antes de salir; si este parámetro se establece en pdTRUE, la función API xTaskNotifyTake () borra el valor de notificación de la tarea a cero.
	xTicksToWait: el tiempo máximo para ingresar a un estado bloqueado mientras espera una notificación. La unidad de tiempo es el ciclo del sistema. La macro pdMS_TO_TICKS se usa para convertir el tiempo de milisegundos especificado en el número correspondiente de tics del sistema.
	Valor de retorno
	Devuelve el valor de notificación actual de la tarea, que es 0 o el valor de notificación menos 1 antes de llamar a la función API xTaskNotifyTake ().
		`uint32_t ulTaskNotifyTake(`
		`BaseType_t` **`xClearCountOnExit,`**
		`TickType_t` **`xTickToWait`**`);`

xTaskNotify()	Lectura del grupo de eventos o espera a la suscitación de alguno(s) de ello. **Parámetros:** **xTaskToNotify:** el identificador de tarea que se notifica. **ulValue:** notifica el valor de actualización **eAction:** tipo de enumeración, que indica el método para actualizar el valor de notificación **Valor Retorno.** Cuando el parámetro eAction es eSetValueWithoutOverwrite, si la tarea notificada no ha recibido la notificación anterior y recibió otra notificación, el valor de la notificación no se actualiza y devuelve pdFALSE esta vez, de lo contrario, devuelve pdPASS.	`BaseType_t xTaskNotify(TaskHandle_t` **`xTaskToNotify,`** `uint32_t` **`ulValue,`** `eNotifyAction` **`eAction`** `);`
xTaskNotifyWait()	Bloquea una tarea hasta recibir una notificación, posee más opciones de configuración **Parámetros** **ulBitsToClearOnEntry**: antes de usar la notificación, primero realice la operación AND bit a bit en la negación bit a bit del valor de notificación de la tarea y el parámetro ulBitsToClearOnEntry. Establezca el parámetro ulBitsToClearOnEntry en 0xFFFFFFFF (ULONG_MAX), lo que significa que el valor de notificación de la tarea se borra. **ulBitsToClearOnExit:** antes de que la función xTaskNotifyWait () salga, realice la operación AND a nivel de bit en el valor de la notificación de tarea y la negación a nivel de bit del parámetro ulBitsToClearOnExit. Establezca el parámetro ulBitsToClearOnExit en 0xFFFFFFFF (ULONG_MAX), lo que significa que el valor de notificación de la tarea se borra. **pulNotificationValue:** el valor de notificación utilizado para devolver la tarea. Este valor de notificación copia el valor de notificación en * pulNotificationValue antes de que el parámetro ulBitsToClearOnExit tenga efecto. Si no necesita devolver el valor de notificación de la tarea, configúrelo como NULL aquí. **xTicksToWait**: el tiempo máximo para ingresar a un estado bloqueado mientras espera una notificación. La unidad de tiempo es el ciclo del sistema. La macro pdMS_TO_TICKS se usa para convertir el tiempo de milisegundos especificado en el número correspondiente de tics del sistema. **Valor de retorno** Si se recibe una notificación, devuelve pdTRUE. Si la función API xTaskNotifyWait () espera el tiempo de espera, devuelve pdFALSE.	`BaseType_t xTaskNotifyWait(uint32_t` **`ulBitsToClearOnEntry,`** `uint32_t` **`ulBitsToClearOnExit,`** `uint32_t` **`*pulNotificationValue,`** `TickType_t` **`xTickToWait`**`);`

https://programmerclick.com/article/731487462/

Ejemplo de notificaciones 1

```
//Cabeceras requeridas
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include “freertos/task.h”

//Declaración de constantes
static TaskHandle_t receiverHandler = NULL;

//Función/tarea sender
void sender(void * params)
{
    while (true)
    {
xTaskNotifyGive(receiverHandler);
      vTaskDelay(5000 / portTICK_PERIOD_MS);
    }

}


//Función/tarea receiver
void receiver(void * params)
{
  while (true)
  {
    int count = ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
    printf("received notification %d times\n", count);
  }

}

//Función main con declaración de tareas
void app_main(void)
{
   xTaskCreate(&receiver, "sender", 2048, NULL, 2,
&receiverHandler);
   xTaskCreate(&sender, "receiver", 2048, NULL, 2, NULL);
}
```

Ejemplo de notificaciones 2

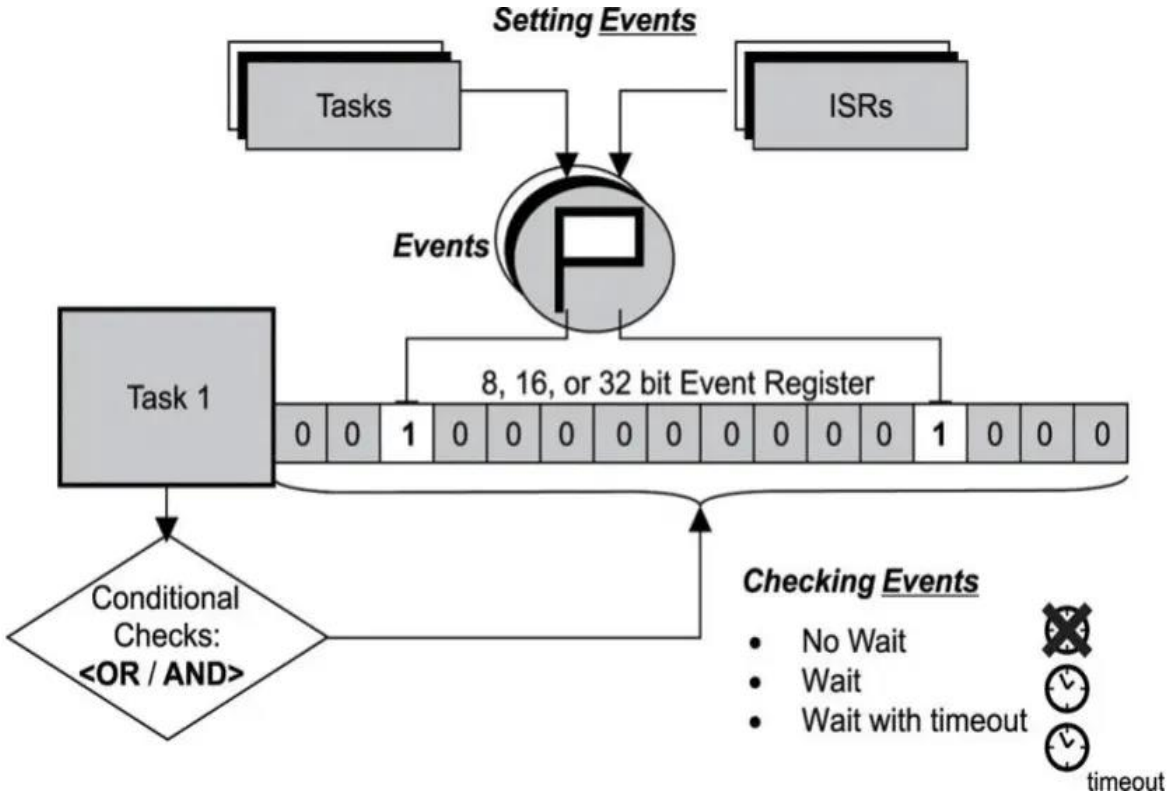
```
//Cabeceras requeridas
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h“

//Declaración de constantes
static TaskHandle_t receiverHandler = NULL;

//Función/tarea sender
void sender(void *params)
{
  while (true)
  {
    xTaskNotify(receiverHandler, (1 << 0), eSetBits);
    xTaskNotify(receiverHandler, (1 << 1), eSetBits);
    vTaskDelay(1000 / portTICK_PERIOD_MS);
    xTaskNotify(receiverHandler, (1 << 2), eSetBits);
    xTaskNotify(receiverHandler, (1 << 3), eSetBits);
    vTaskDelay(1000 / portTICK_PERIOD_MS);
  }
}
```

```
//Función/tarea receiver
void receiver(void *params)
{
  uint state;
  while (true)
  {
    xTaskNotifyWait(0xffffffff, 0, &state, portMAX_DELAY);
    printf("received state %d times\n", state);
  }
}
//Función main con declaración de tareas
void app_main(void)
{
  xTaskCreate(&receiver, "sender", 2048, NULL, 2,
&receiverHandler);
  xTaskCreate(&sender, "receiver", 2048, NULL, 2, NULL);
}
```

Grupos de Eventos

- El uso de grupos de eventos sirve para la sincronización de tareas (así como las colas y los semáforos)
- Un grupo de eventos (G.E.) puede almacenar múltiples eventos (flags) con la posibilidad de dejar una tarea en espera(estado bloqueado) hasta que ocurran uno o más eventos específicos.
- Los eventos se presentan mediante banderas (flags) representadas en forma binaria e indican cuando ocurrió una acción o no como por ejemplo la activación de una bandera al terminar el muestreo de un sensor, una interrupción asignada a un botón que indica que se ha activado.
- Por lo tanto los grupos de eventos son variables de 16 o 32 bits(según lo requiera el usuario). Cada objeto del grupo de eventos puede controlar hasta 24 eventos, logrando ahorrar valioso espacio en memoria.
- A diferencia de los semáforos y las colas, un grupo de eventos tiene algunas características importantes:

  1. Se pueden despertar tareas con la combinación de una o varias banderas diferentes.
  2. Reduce el consumo de RAM en comparación con el uso de múltiples semáforos binarios o colas para la sincronización;
  3. **Todas las** tareas que están en espera (estado bloqueado) de banderas que las desbloqueadas, funcionando como un "mensaje de difusión" y no solo la prioridad más alta, como en el caso de los semáforos y las colas.



Resumen de funciones asociadas a los grupos de eventos

Nombre de función	Descripción	Código
xEventGroupCreate()	Creación de un grupo de eventos Valor de retorno: cuando se devuelve NULL, la creación falló debido a la memoria insuficiente; de lo contrario, la creación se realizó correctamente.	`EventGroupHandle_t xEventGroupCreate( void );`
xEventGroupSetBits()	Levantamiento de las banderas de evento de un grupo de eventos. Parámetros: • **xEventGroup:** nombre del grupo de eventos. • **uxBitsToSet** La posición del evento que debe establecerse, por ejemplo, 0x04, lo que indica que bit3 debe establecerse. Puede configurar varios bits a la vez, como 0x05. Valor de retorno: el valor del grupo de eventos.	`EventBits_t xEventGroupSetBits( EventGroupHandle_t **xEventGroup**, const EventBits_t **uxBitsToSet** );`
xEventGroupWaitBits()	Lectura del grupo de eventos o espera a la suscitación de alguno(s) de ellos. Parámetros: • **uxBitsToWaitFor**: indica qué bits deben probarse. • **xWaitForAllBits**: Indica si se debe salir del estado de bloqueo siempre que haya un bit compuesto, o si se sale del estado de bloqueo después de configurar todos los bits en uxBitsToWaitFor. • **xEventGroup**: nombre del grupo de eventos. • **xClearOnSalir:** Si es pdTRUE, la posición relevante será 0 antes de que la función salga, si es pdFALSE, no operará en el grupo de tiempo. (El bit de bandera también se puede borrar mediante EventGroupClearBits ()). • **xTicksToWait:** el tiempo de espera de bloqueo más largo. Valor de retorno: si se establece el bit relevante, el valor de retorno es el valor antes del reinicio del grupo de eventos (xClearOnSalir es pdTRUE). Si se agota el tiempo de espera, el valor devuelto es el tiempo transcurrido en el estado bloqueado.	`EventBits_t xEventGroupWaitBits( const EventGroupHandle_t **xEventGroup**, const EventBits_t **uxBitsToWaitFor**, const BaseType_t **xClearOnExit**, const BaseType_t **xWaitForAllBits**, TickType_t **xTicksToWait** );`
xEventGroupSync()	Creación de un punto de sincronización. **xEventGroup:** indica el nombre del grupo de eventos. **uxBitsToSet:** bits que deben configurarse. **uxBitsToWaitFor:** el valor del grupo de eventos establecido que se espera. **xTicksToWait:** tiempo para bloquear la espera. Valor de retorno: si todos los bits relevantes están configurados, el valor devuelto es el valor del grupo de datos antes del reinicio, y si se agota, el valor devuelto es el tiempo de espera de bloqueo	`EventBits_t xEventGroupSync( EventGroupHandle_t **xEventGroup**, const EventBits_t **uxBitsToSet**, const EventBits_t **uxBitsToWaitFor**, TickType_t **xTicksToWait** );`

https://programmerclick.com/article/85971841840/

Ejemplo de implementación grupo de eventos

```
//Cabeceras requeridas
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/semphr.h"
#include "freertos/event_groups.h“

//Declaración de constantes
EventGroupHandle_t evtGrp;
const int gotHttp = BIT0;
const int gotBLE = BIT1;

//Función tareas listenForHTTP
void listenForHTTP(void *params)
{
  while (true)
  {
    xEventGroupSetBits(evtGrp, gotHttp);
    printf("got http\n");
    vTaskDelay(2000 / portTICK_PERIOD_MS);
  }
}

//Función tareas listenForBluetooth
void listenForBluetooth(void *params)
{
  while (true)
  {
    xEventGroupSetBits(evtGrp, gotBLE);
    printf("got BLE\n");
    vTaskDelay(5000 / portTICK_PERIOD_MS);
  }
}
//Función task1, espera los eventos de listenForHTTP y
listenForBluetooth
void task1(void *params)
{
  while (true)
  {
    xEventGroupWaitBits(evtGrp, gotHttp | gotBLE, true, true,
portMAX_DELAY);
    printf("received http and BLE\n");
  }
}
//Función main con declaración de tareas + G.E.
void app_main(void)
{
  evtGrp = xEventGroupCreate();
  xTaskCreate(&listenForHTTP, "get http", 2048, NULL, 1,
NULL);
  xTaskCreate(&listenForBluetooth, "get BLE", 2048, NULL, 1,
NULL);
  xTaskCreate(&task1, "do something with http", 2048, NULL,
1, NULL);
}
```

Resultado ejemplo grupo de eventos

```
I (282) spi_flash: detected chip: generic
I (286) spi_flash: flash io: dio
W (290) spi_flash: Detected size(4096k) larger than the size in the binary image header(2048k). Using the size in the bi
nary image header.
I (304) cpu_start: Starting scheduler on PRO CPU.
I (0) cpu_start: Starting scheduler on APP CPU.
got http
got BLE
received http and BLE
got http
got http
got BLE
received http and BLE
got http
got http
got http
got BLE
received http and BLE
got http
got http
got BLE
received http and BLE
got http
got http
got http
got BLE
received http and BLE
got http
got http
```

MUTEX

En ocasiones es común encontrar escenarios donde múltiples tareas o ISRs desean acceder a un mismo recurso del MCU simultáneamente (e.g. acceso a periféricos, operaciones de lectura, escritura a algún espacio en memoria), sin embargo, de hacerlo ocasionaría una corrupción de los datos por acceder.

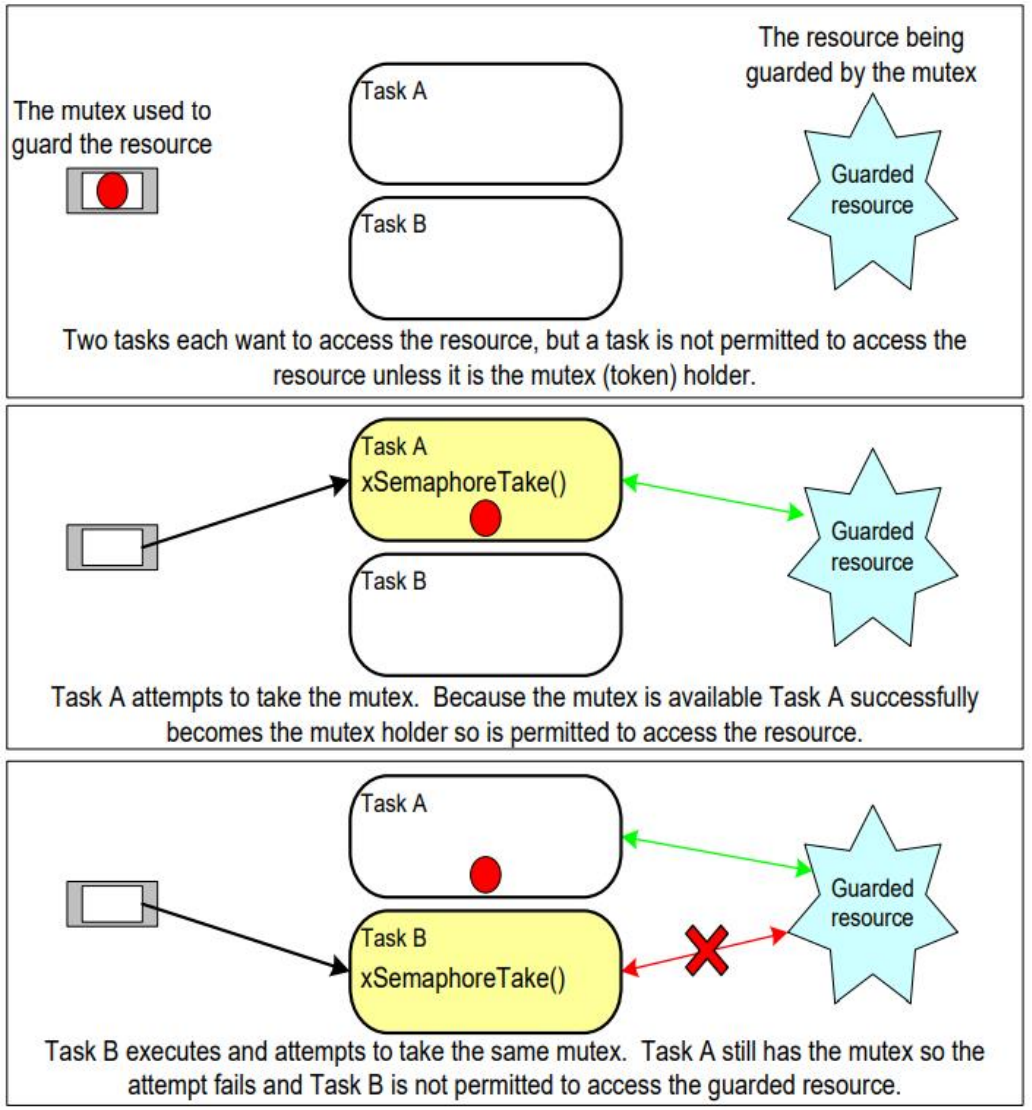
Para resolver este problema se recurre a métodos que permitan el bloqueo de los recursos, es decir métodos que permitan escoger quien tendrá acceso al recurso mientras excluye al resto de interesados.

En principio, el mejor método para ello es el diseño apropiado de la aplicación a realizar para evitar el acceso simultaneo de los recursos por múltiples tareas o ISRs. No obstante, dentro del entorno FreeRTOS es posible emplear el elemento denominado como “mutex” cuyo funcionamiento es muy similar al de un semáforo binario, (ver diagrama), dado que se brinda una ficha a una determinada tarea o ISR para otorgarle el permiso de acceso al recurso deseado, por lo que el resto de los elementos que deseen acceder al recurso tienen que aguardar a la liberación de la ficha para poder reclamar el acceso al recurso.

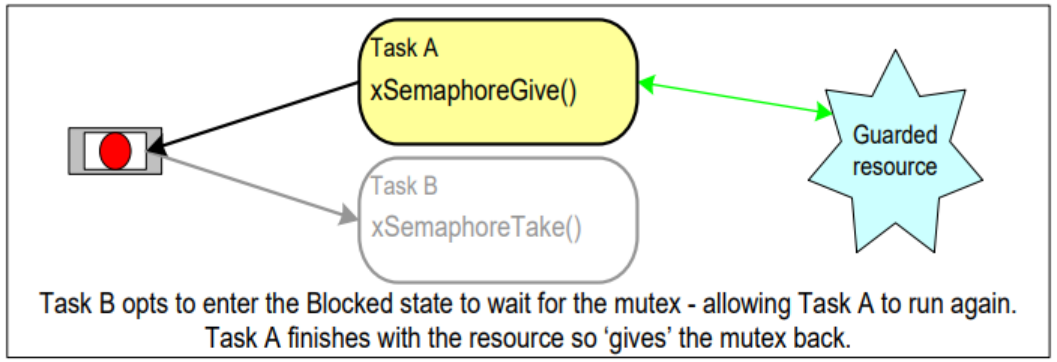
No obstante, la diferencia entre un semáforo y un mutex radica en la acción que se toma tras la adquisición de la ficha para ambos elementos, debido a que:

•La ficha en los mutex siempre debe ser regresada.

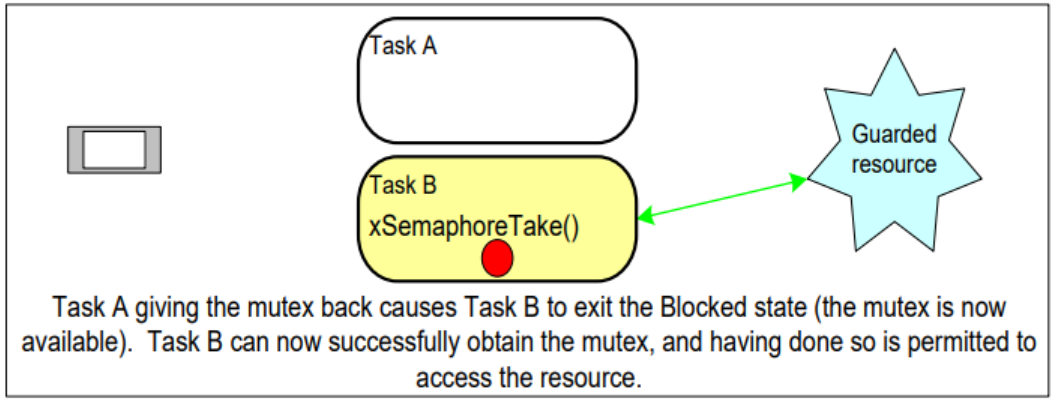
•Un semáforo tomado normalmente es descartado tras finalizarse la ejecución de una tarea y nunca es entregado de vuelta. La entrega en este caso la realiza normalmente un ISR.



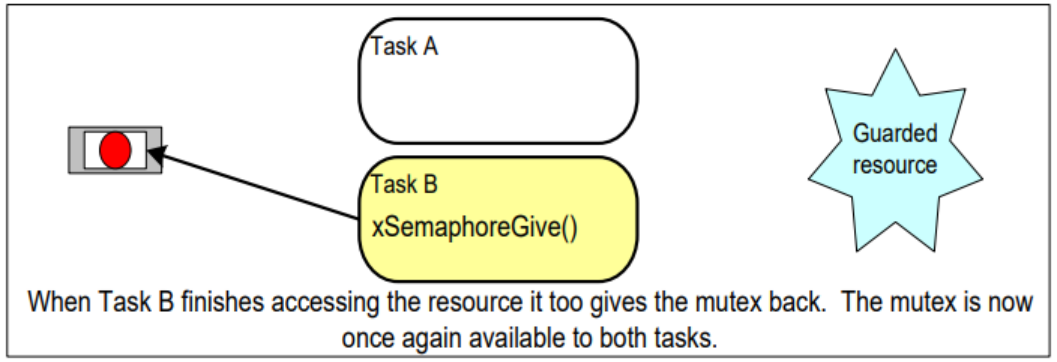
continuación



Task B opts to enter the Blocked state to wait for the mutex - allowing Task A to run again. Task A finishes with the resource so 'gives' the mutex back.



Task A giving the mutex back causes Task B to exit the Blocked state (the mutex is now available). Task B can now successfully obtain the mutex, and having done so is permitted to access the resource.



When Task B finishes accessing the resource it too gives the mutex back. The mutex is now once again available to both tasks.

Creación del Mutex

Declaración:

SemaphoreHandle_t xSemaphoreCreateMutex(void);

Valor de retorno:

* NULL: No se pudo crear el identificador del mutex.
* non-NULL: Se pudo crear el identificador del mutex .

Resumen de funciones asociadas a los semáforos

Nombre	Descripción	Código
xSemaphoreCreateMutex()	Se crea el identificador del mutex Valor de retorno: * NULL: No se pudo crear el identificador del mutex. * non-NULL: Se pudo crear el identificador del mutex .	`SemaphoreHandle_t xSemaphoreCreateMutex( void )`
xSemaphoreTake()	Acción de retirar una ficha del semáforo **Parámetros:** * xSemaphore: Identificador del semáforo que es tomado. * xTicksToWait: Maxima cantidad de ticks que la tarea tiene que esperar en el estado “bloquedo” hasta que el semáforo este disponible. **Valor retorno:** * pdTrue: Si el semáforo se pudo obtener. * pdFalse: Si el semáforo no se pudo obtener.	`BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait );`
xSemaphoreGive ()	Acción de colocar una ficha en el semáforo **Parámetros**: * xSemaphore: Identificador del semáforo que es regresado. **Valor de retorno:** * pdTrue: Si fue entregado. * pdFalse: Si algun error se presenta.	`BaseType_t xSemaphoreGive( SemaphoreHandle_t xSemaphore, BaseType_t *pxHigherPriorityTaskWoken);`

https://www.freertos.org/a00106.html

Diferencias entre Mutex y Semáforo binario

Mutex

```
//Tarea 1
semaphoreTake(mutex)
// Usa el recurso compartido
semaphoreGive(mutex)

//Tarea 2
semaphoreTake(mutex)
// Usa el recurso compartido
semaphoreGive(mutex)
```

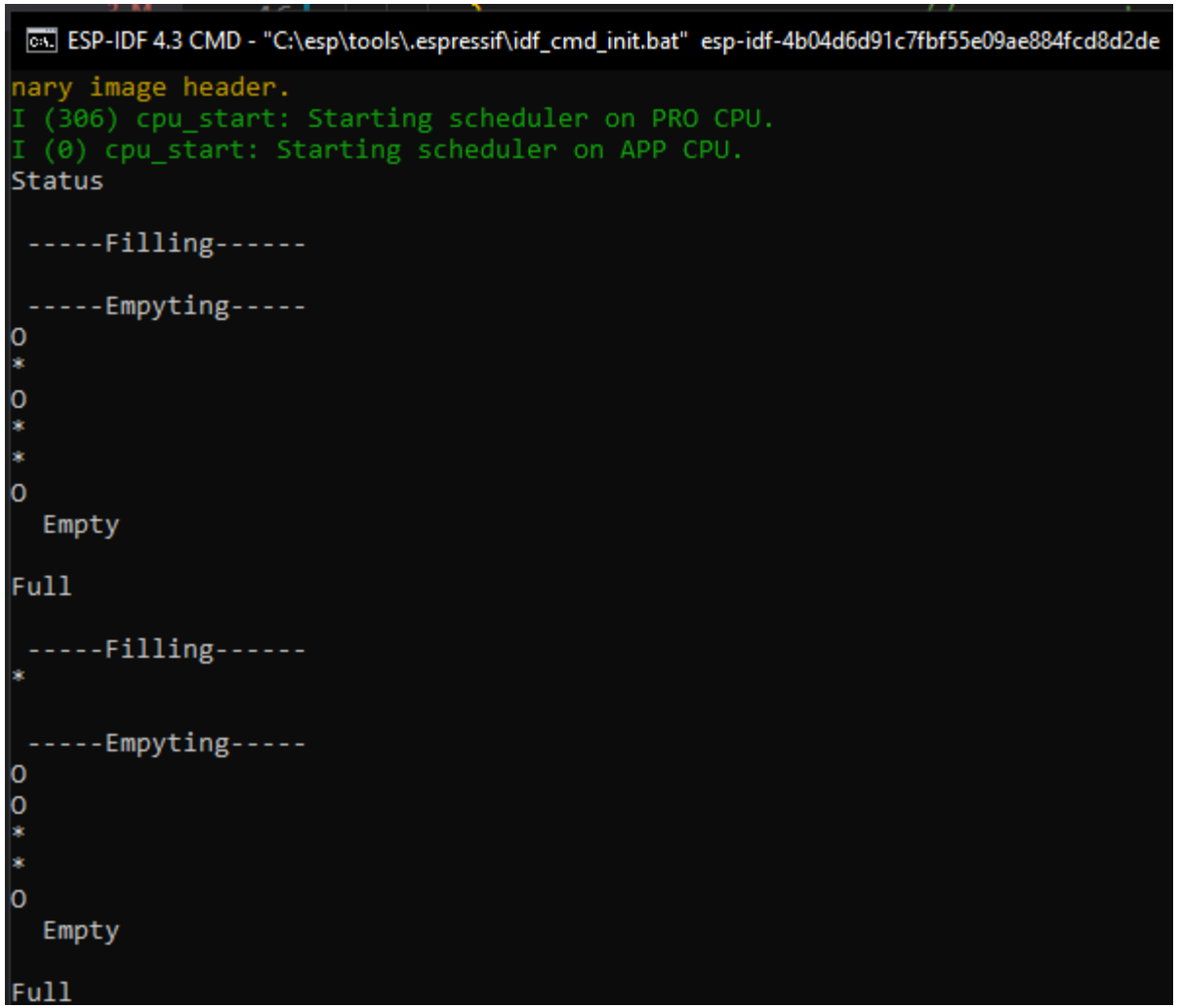
Semáforo

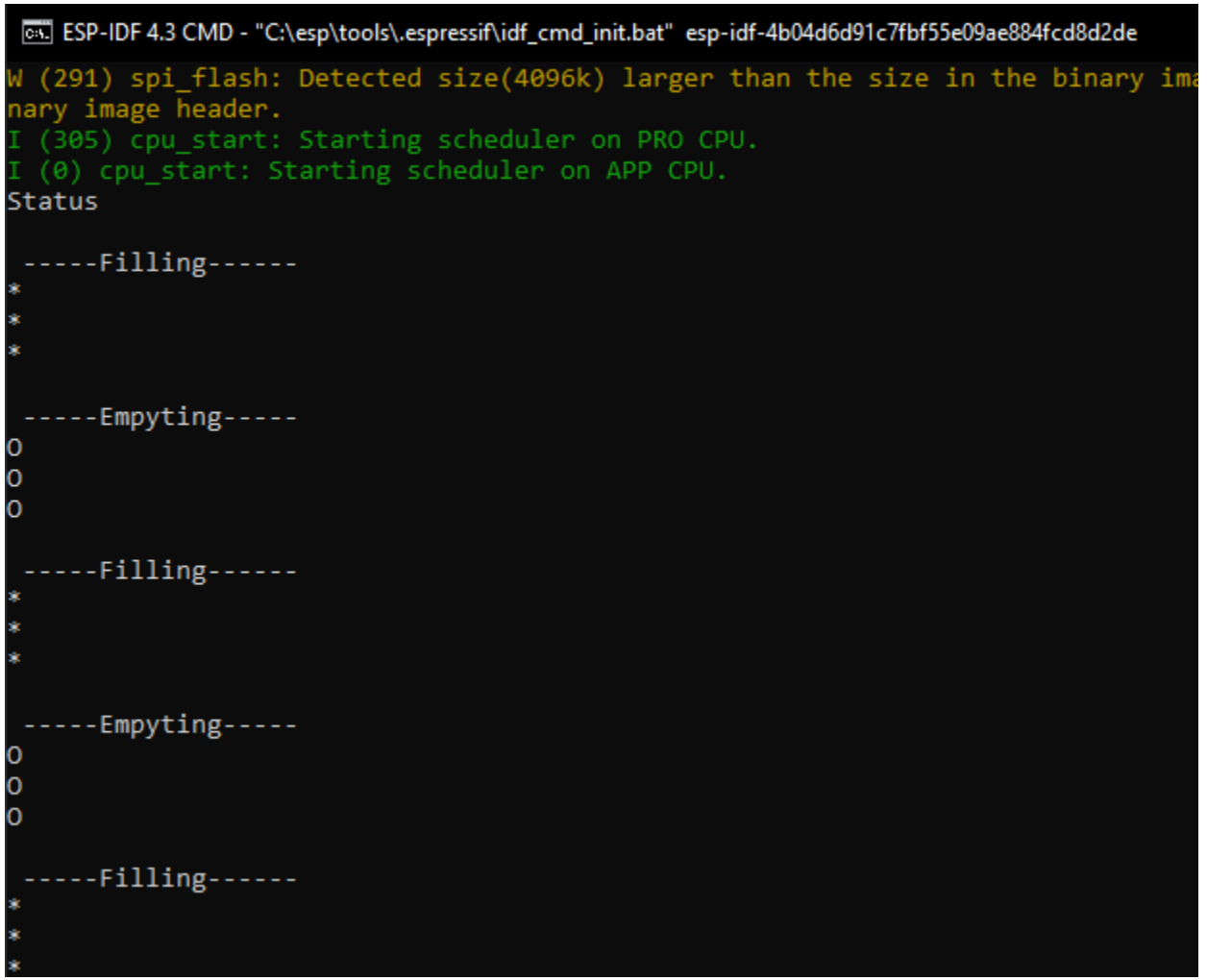
```
//Tarea 1 (productor)
//Se agrega un elemento al
//recurso compartido
semaphoreGive(mutex)

//Tarea 2 (consumidor)
semaphoreTake(mutex)
//Se retira un elemento del
//recurso compartido
```

Resumen características Kernel

	Cola	**Semáforo**	**Notificación**	**Mutex**
Tamaño	Variable	Unitario	Unitario	Unitario
Acción	Lectura/Escritura	Tomar/Entregar	Tomar/Entregar	Tomar/Entregar
Memoria	Mucha	Poca	Muy poca	Poca
Contenido	Importa	No importa	Importa	No importa
Resultado	Desplazamiento	Retira	Sobreescribe	Retira/debe volver
Acceso	Público	**Público**	Privado	Privado
Extras	No	Si	Si	No
Propósito	Intercomunicación	Sincronización	Intercomunicación	Sincronización
Espera	Si ambas	Si al tomar	Si al tomar	Si al tomar

Ejercicio resultado





API’s a utilizar con FreeRTOS en ESP-IDF

- El término API es una abreviatura de Application Programming Interfaces.
- Se trata de un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones, permitiendo la comunicación entre dos aplicaciones de software a través de un conjunto de reglas.
- Algunas APIS desarrolladas por ESPRESSIF son:

- Bluetooth
- Networking (Wi-Fi, Ethernet)
- Peripherals (ADC,DAC, Timersm GPIO’s, LCD, SD, PWM)
- Protocols (ESP-MQTT, OpenSSL, HTTP Client, HTTPS server, Modbus, IP Network Layer)
- Storage (FAT File system)
- System (FreeRTOS)

Guía de referencia de API’s:

https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/index.html

Wi-Fi

- La biblioteca de WiFi proporciona la configuración y monitoreo de las funcionalidades de red del ESP32 Wi-Fi
- Modo de estación (también conocido como modo STA o modo cliente Wi-Fi). ESP32 se conecta a un punto de acceso.
- Modo AP (también conocido como modo Soft-AP o modo de punto de acceso). Las estaciones se conectan al ESP32.
- Modo de coexistencia de estación / AP (ESP32 es simultáneamente un punto de acceso y una estación conectada a otro punto de acceso).
- Varios modos de seguridad para los anteriores (WPA, WPA2, WEP, etc.) Escaneo de puntos de acceso (escaneo activo y pasivo). Modo promiscuo para la monitorización de paquetes Wi-Fi IEEE802.11.

Biblioteca de referencias: components/esp_wifi/include/esp_wifi.h

Sitio de interés: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/network/esp_wifi.html

Convertidor analógico-digital

Canales ADC: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/adc.html

- El ESP32 integra 2 ADC SAR (registro de aproximación sucesiva), que admiten un total de 18 canales de medición (pines analógicos habilitados).
- Se enlistan los pines :

- ADC1: 8 canales: GPIO32 - GPIO39

- ADC2: 10 canales: GPIO0, GPIO2, GPIO4, GPIO12 - GPIO15, GOIO25 - GPIO27

- Headers a incluir: #include "driver/adc.h“, #include "esp_adc_cal.h“
- Ejemplo práctico de lectura de ADC: https://github.com/espressif/esp-idf/tree/fa9cc49/examples/peripherals/adc/single_read

Ejemplo Blink

```
/* Blink Example

   This example code is in the Public Domain (or CC0 licensed, at
your option.)

   Unless required by applicable law or agreed to in writing, this
   software is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES
OR
   CONDITIONS OF ANY KIND, either express or implied.
*/
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"
#include "sdkconfig.h"

/* Can use project configuration menu (idf.py menuconfig) to choose
the GPIO to blink,
   or you can edit the following line and set a number here.
*/
#define BLINK_GPIO CONFIG_BLINK_GPIO

void app_main(void)
{
    /* Configure the IOMUX register for pad BLINK_GPIO (some pads
are
       muxed to GPIO on reset already, but some default to other
       functions and need to be switched to GPIO. Consult the
       Technical Reference for a list of pads and their default
       functions.)
    */
    gpio_reset_pin(BLINK_GPIO);
    /* Set the GPIO as a push/pull output */
    gpio_set_direction(BLINK_GPIO, GPIO_MODE_OUTPUT);
    while(1) {
        /* Blink off (output low) */
        printf("Turning off the LED\n");
        gpio_set_level(BLINK_GPIO, 0);
        vTaskDelay(1000 / portTICK_PERIOD_MS);
        /* Blink on (output high) */
        printf("Turning on the LED\n");
        gpio_set_level(BLINK_GPIO, 1);
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

```
/*
 * SPDX-FileCopyrightText: 2021 Espressif Systems (Shanghai) CO LTD
 * SPDX-License-Identifier: Apache-2.0
 */
/* ADC1 Example
   This example code is in the Public Domain (or CC0 licensed, at your option.)
   Unless required by applicable law or agreed to in writing, this
   software is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR
   CONDITIONS OF ANY KIND, either express or implied.
*/
#include <stdio.h>
#include <stdlib.h>
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/adc.h"
#include "esp_adc_cal.h"
//ADC Channels
#if CONFIG_IDF_TARGET_ESP32
#define ADC1_EXAMPLE_CHAN0          ADC1_CHANNEL_6
#define ADC2_EXAMPLE_CHAN0          ADC2_CHANNEL_0
static const char *TAG_CH[2][10] = {{"ADC1_CH6"}, {"ADC2_CH0"}};
#else
#define ADC1_EXAMPLE_CHAN0          ADC1_CHANNEL_2
#define ADC2_EXAMPLE_CHAN0          ADC2_CHANNEL_0
static const char *TAG_CH[2][10] = {{"ADC1_CH2"}, {"ADC2_CH0"}};
#endif

//ADC Attenuation
#define ADC_EXAMPLE_ATTEN           ADC_ATTEN_DB_11

//ADC Calibration
#if CONFIG_IDF_TARGET_ESP32
#define ADC_EXAMPLE_CALI_SCHEME     ESP_ADC_CAL_VAL_EFUSE_VREF
#elif CONFIG_IDF_TARGET_ESP32S2
#define ADC_EXAMPLE_CALI_SCHEME     ESP_ADC_CAL_VAL_EFUSE_TP
#elif CONFIG_IDF_TARGET_ESP32C3
#define ADC_EXAMPLE_CALI_SCHEME     ESP_ADC_CAL_VAL_EFUSE_TP
#elif CONFIG_IDF_TARGET_ESP32S3
#define ADC_EXAMPLE_CALI_SCHEME     ESP_ADC_CAL_VAL_EFUSE_TP_FIT
#endif

static int adc_raw[2][10];
static const char *TAG = "ADC SINGLE";

static esp_adc_cal_characteristics_t adc1_chars;
static esp_adc_cal_characteristics_t adc2_chars;

static bool adc_calibration_init(void)
{
    esp_err_t ret;
    bool cali_enable = false;

    ret = esp_adc_cal_check_efuse(ADC_EXAMPLE_CALI_SCHEME);
    if (ret == ESP_ERR_NOT_SUPPORTED) {
        ESP_LOGW(TAG, "Calibration scheme not supported, skip software calibration");
    } else if (ret == ESP_ERR_INVALID_VERSION) {
        ESP_LOGW(TAG, "eFuse not burnt, skip software calibration");
    } else if (ret == ESP_OK) {
        cali_enable = true;
        esp_adc_cal_characterize(ADC_UNIT_1, ADC_EXAMPLE_ATTEN, ADC_WIDTH_BIT_DEFAULT, 0,
&adc1_chars);
        esp_adc_cal_characterize(ADC_UNIT_2, ADC_EXAMPLE_ATTEN, ADC_WIDTH_BIT_DEFAULT, 0,
&adc2_chars);
    } else {
        ESP_LOGE(TAG, "Invalid arg");
    }

    return cali_enable;
}

void app_main(void)
{
    esp_err_t ret = ESP_OK;
    uint32_t voltage = 0;
    bool cali_enable = adc_calibration_init();
    //ADC1 config
    ESP_ERROR_CHECK(adc1_config_width(ADC_WIDTH_BIT_DEFAULT));
    ESP_ERROR_CHECK(adc1_config_channel_atten(ADC1_EXAMPLE_CHAN0, ADC_EXAMPLE_ATTEN));
    //ADC2 config
    ESP_ERROR_CHECK(adc2_config_channel_atten(ADC2_EXAMPLE_CHAN0, ADC_EXAMPLE_ATTEN));
    while (1) {
        adc_raw[0][0] = adc1_get_raw(ADC1_EXAMPLE_CHAN0);
        ESP_LOGI(TAG_CH[0][0], "raw  data: %d", adc_raw[0][0]);
        if (cali_enable) {
            voltage = esp_adc_cal_raw_to_voltage(adc_raw[0][0], &adc1_chars);
            ESP_LOGI(TAG_CH[0][0], "cali data: %d mV", voltage);
        }
        vTaskDelay(pdMS_TO_TICKS(1000));

        do {
            ret = adc2_get_raw(ADC2_EXAMPLE_CHAN0, ADC_WIDTH_BIT_DEFAULT, &adc_raw[1][0]);
        } while (ret == ESP_ERR_INVALID_STATE);
        ESP_ERROR_CHECK(ret);

        ESP_LOGI(TAG_CH[1][0], "raw  data: %d", adc_raw[1][0]);
        if (cali_enable) {
            voltage = esp_adc_cal_raw_to_voltage(adc_raw[1][0], &adc2_chars);
            ESP_LOGI(TAG_CH[1][0], "cali data: %d mV", voltage);
        }
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}
```

Ejemplo practico ADC

Tarea

- Revisar el sitio oficial de configuración de API’s de FreeRTOS –ESPIDF

https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/index.html