

UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL PARANÁ

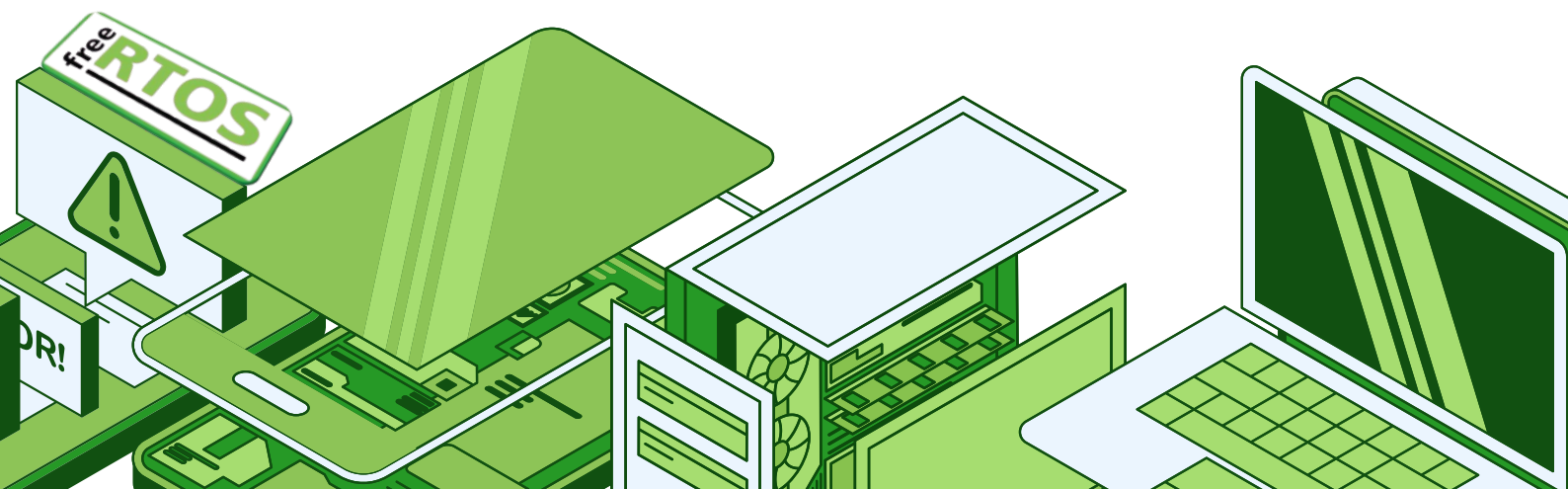
DOCUMENTACIÓN

Beca de servicio FreeRTOS

Docentes: Caballero Raúl, Maggiolo Gustavo.

Alumnos: Espindola Agustín, Glas Sebastián.

2024



Índice

INSTALACIÓN DE PLATFORMIO	2
SISTEMA OPERATIVO EN TIEMPO REAL (RTOS)	5
RTOS y diferencias con un sistema de propósito general	5
¿Por qué usar RTOS?	6
Super Loop vs Multitarea	6
Terminología en RTOS	7
ADMINISTRACIÓN Y COMUNICACIÓN ENTRE TAREAS	8
Tarea	8
Cola	9
SINCRONIZACIÓN Y CONTROL DE ACCESO A RECURSOS	10
Mutex	10
Semáforo	11
Diferencia entre un mutex y un semáforo	13
PROBLEMAS DE CONCURRENCIA Y GESTIÓN DE PRIORIDADES	14
Deadlock y Starvation	14
Inversión de prioridades	16
GESTIÓN DE TIEMPO Y EVENTOS	19
Temporizadores de software	19
Interrupciones de hardware	20
ADMINISTRACIÓN DE MEMORIA	21
Memoria	21
BIBLIOGRAFÍA	22

Instalación de PlatformIO

PlatformIO es un entorno de desarrollo integrado (IDE) fácil de usar y extensible, con un conjunto de herramientas profesionales de desarrollo, que ofrece características modernas y potentes para acelerar y simplificar la creación y entrega de productos embebidos. Incorpora más de 20 frameworks con más de 1500 placas de desarrollo, siendo una opción muy flexible. Durante el desarrollo de los ejemplos presentados utilizaremos PlatformIO dentro de Visual Studio Code. A continuación, describiremos el paso a paso de la instalación y de la creación de un proyecto:

- 1) Para instalar PlatformIO necesitamos instalar VSCode, para ello vamos a la página de descarga (<https://code.visualstudio.com/download>) y seleccionamos la opción que se adecue a nuestro sistema operativo.
- 2) Ejecutamos el instalador, aceptamos los términos y presionamos siguiente hasta terminar la instalación.
- 3) Una vez instalado, abriremos el programa y buscaremos la pestaña de extensiones (ctrl+shift+x).

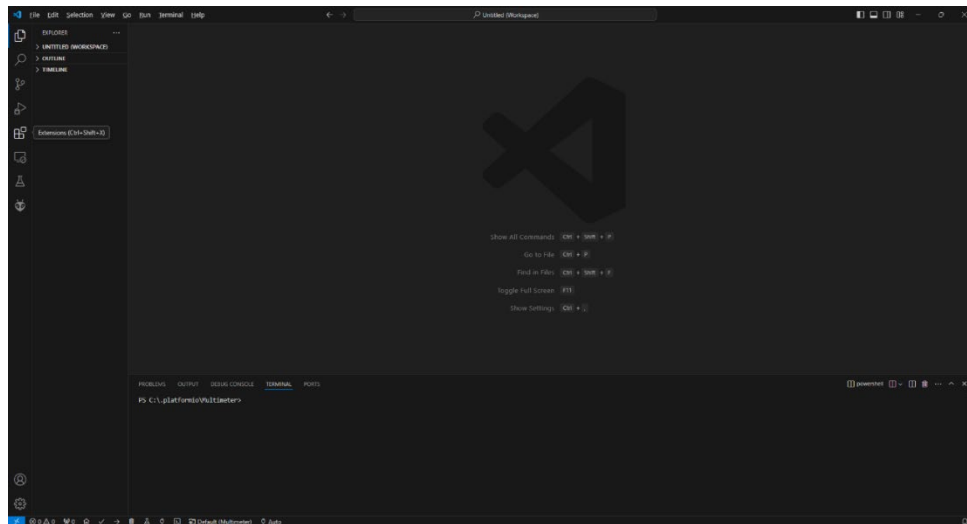


Figura 1 - Ventana principal VsCode

- 4) En el buscador de extensiones escribimos PlatformIO y aparecerá una extensión llamada "PlatformIO IDE", tal como se ve en la imagen. Ahí simplemente se elige la opción instalar y ya estará listo para utilizar.

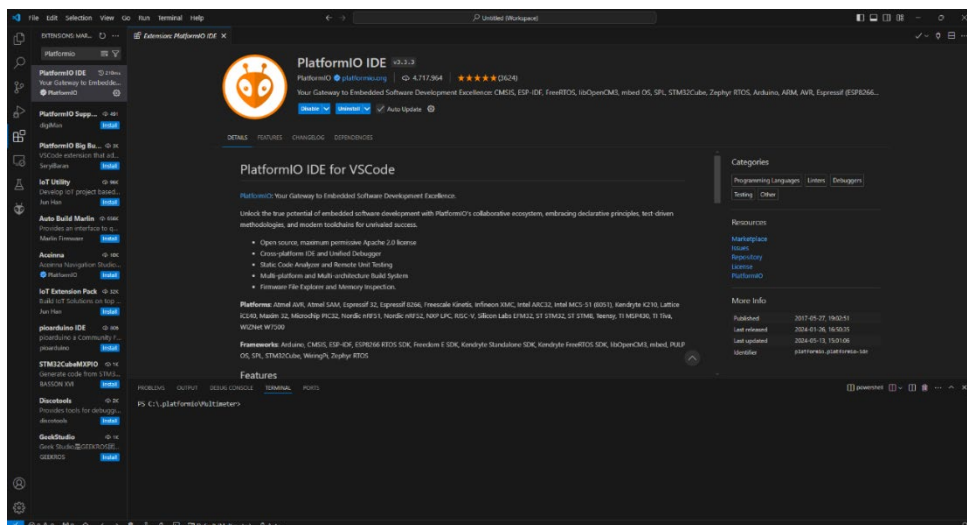


Figura 2 - Ventana PlatformIO IDE en VsCode



- 5) Para crear un nuevo proyecto debemos ir a la pestaña de PlatformIO (en la barra de la izquierda aparecerá el icono de una hormiga luego de la instalación) y dentro de la división “PIO Home” elegimos “Open”.

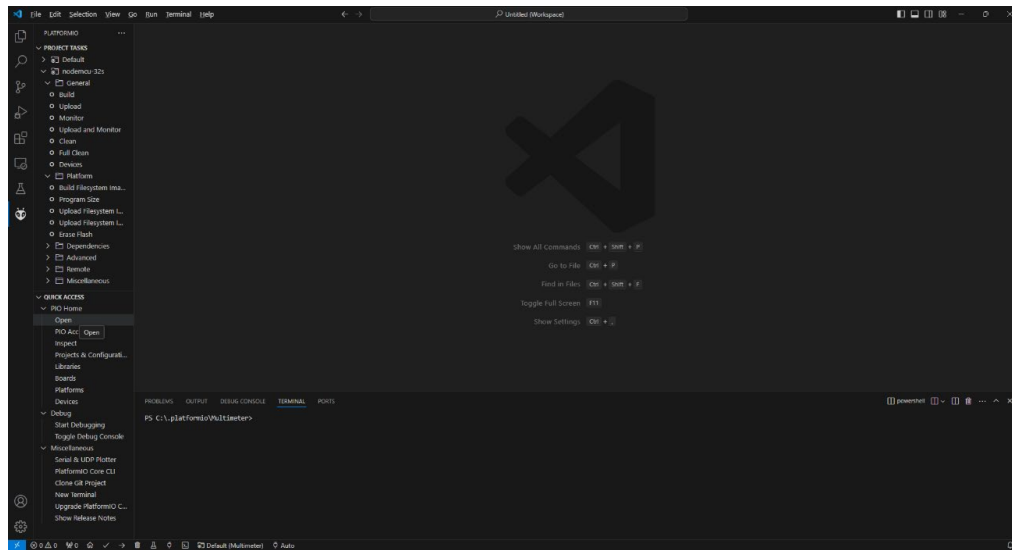


Figura 3 - Creación de proyecto

- 6) Dentro de esta ventana elegimos nuevo proyecto.

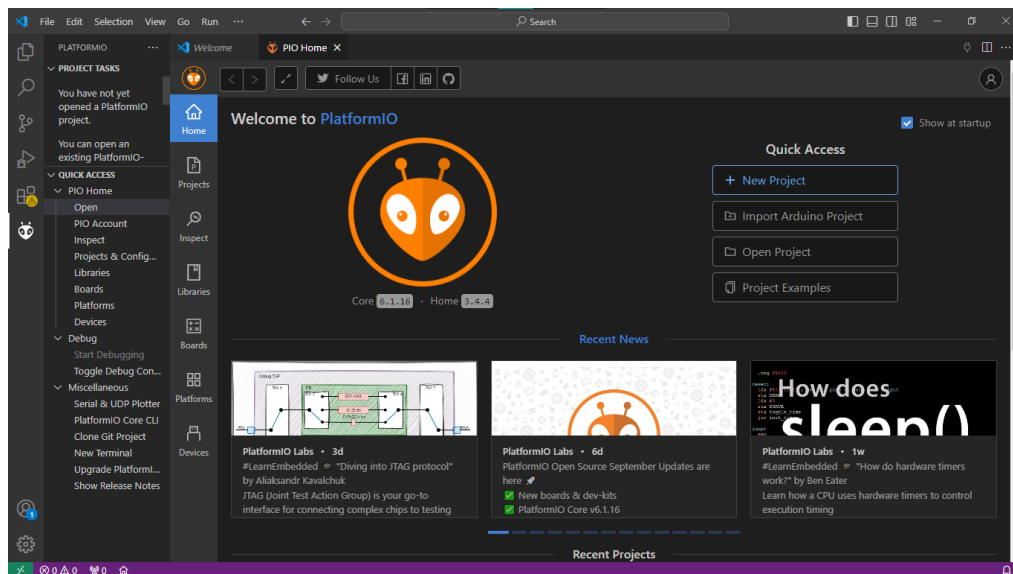
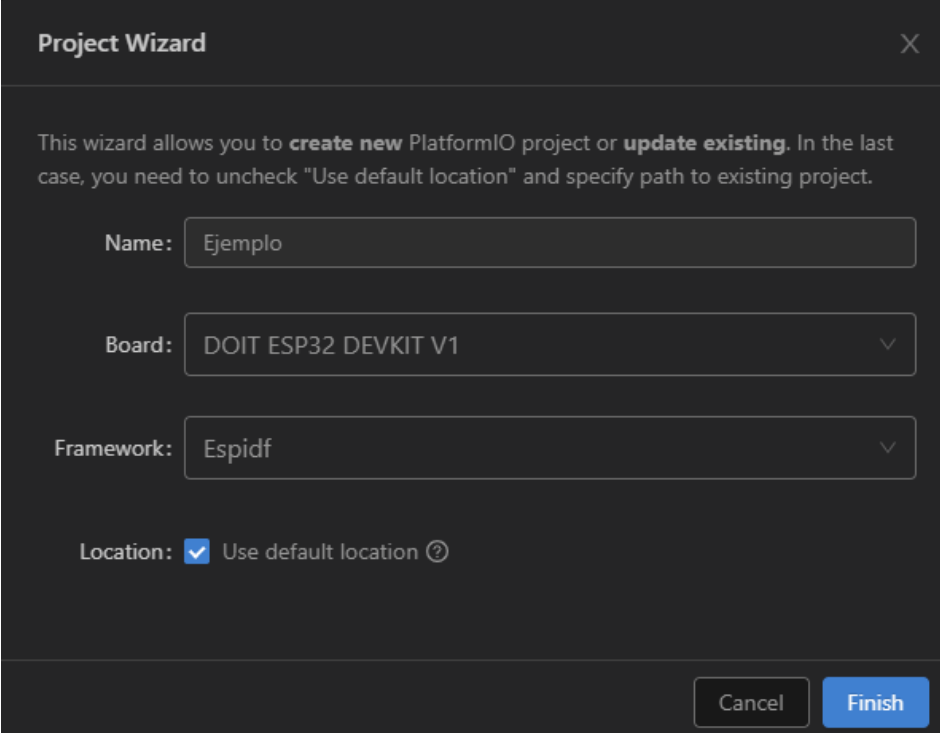


Figura 4 - Creación de proyecto

- 7) Ahora seleccionamos el nombre que deseamos, la placa que vamos a utilizar (en nuestro caso ESP32) y el framework.



Project Wizard [X]

This wizard allows you to **create new** PlatformIO project or **update existing**. In the last case, you need to uncheck "Use default location" and specify path to existing project.

Name:

Board:

Framework:

Location: ☒ Use default location ?

Figura 5 - Configuración del proyecto

Sistema operativo en tiempo real (RTOS)

Antes de hablar de un sistema operativo en tiempo real, debemos conocer el concepto de sistema operativo en general. Este es un conjunto de programas que gestiona recursos del hardware para que funcionen eficientemente las aplicaciones. Actúa como intermediario entre el hardware de la computadora y los programas de usuario, facilitando la interacción entre ambos. Tiene muchas funciones a cargo, entre ellas podemos mencionar la gestión de recursos, la administración de tareas, el control de archivos, la seguridad y protección y proporcionar una interfaz de usuario. Algunos ejemplos de sistemas operativos son: Windows, Linux, Android, iOS, etc.

RTOS y diferencias con un sistema de propósito general

Para explicar este concepto debemos saber que la mayoría de los sistemas operativos “parecen” permitir la ejecución de varios programas al mismo tiempo. Esta característica se denomina **multitarea**. Pero, la realidad, es que cada núcleo de procesador solo puede ejecutar un único subproceso de ejecución en un momento dado. El encargado de decidir qué programa ejecutar y cuando dentro del sistema operativo es el **programador de tareas** o **scheduler**. Este componente proporciona la ilusión de ejecución simultánea al cambiar rápidamente entre cada programa.

Por otro lado, un requisito de tiempo real es aquel que especifica que el sistema embebido debe responder a un determinado evento dentro de un tiempo estrictamente definido (fecha límite). Sólo se puede garantizar el cumplimiento de los requisitos en tiempo real si el comportamiento del scheduler del sistema operativo se puede predecir (es determinista).

Por lo tanto, un sistema operativo en tiempo real (RTOS) posee las mismas funcionalidades que es un sistema operativo de propósito general, pero con la diferencia que un RTOS debe considerar el tiempo límite que debe cumplir cada tarea.

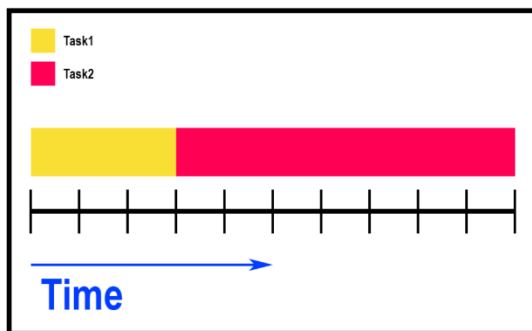


Figura 6 – Super Loop en GPOS

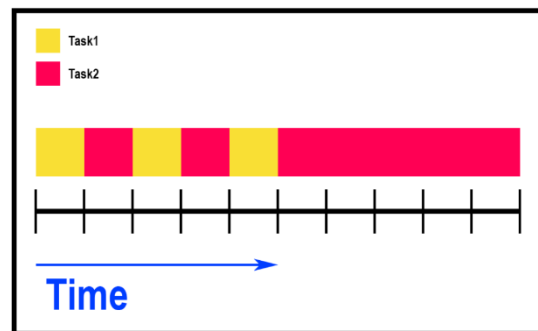


Figura 7 – Multitarea en RTOS

Algunos ejemplos de sistemas operativos son: FreeRTOS, VxWorks, algunas distribuciones de UNIX, etc.



¿Por qué usar RTOS?

El RTOS es un sistema pensado para microcontroladores y debe utilizarse para cumplir con compromisos temporales estrictos. Se encarga de absorber el manejo de temporizadores y esperas, de modo que hace más fácil al programador el manejo del tiempo.

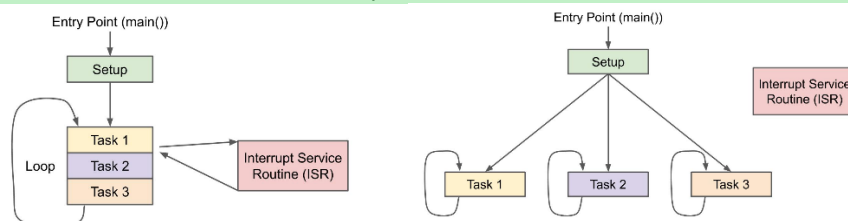
A continuación, nombraremos algunas características que son destacables en FreeRTOS:

- Permite abstraer información de tiempo. Por lo tanto, la estructura del código de la aplicación es más sencilla y el tamaño general del código es menor.
- La aplicación es menos susceptible a cambios en el hardware subyacente.
- Modularidad y reutilización del código.
- Posee un tiempo de inactividad (idle time) conformado por una tarea inactiva (idle) que se ejecuta automáticamente cuando no hay tareas de aplicación que requieran procesamiento. La tarea inactiva puede medir la capacidad de procesamiento sobrante, realizar verificaciones de antecedentes o colocar el procesador en un modo de bajo consumo.
- Bajo consumo de energía debido al tiempo de inactividad.
- Manejo flexible de interrupciones.
- Requisitos de procesamiento mixto.

Super Loop vs Multitarea

PARÁMETRO	SUPER LOOP	MULTITAREA
Estructura	Bucle infinito simple, ejecutando tareas en forma secuencial.	Las tareas se ejecutan dependiendo su prioridad.
Concurrencia	No permite la ejecución de tareas en paralelo.	Sí, lo permite.
Escalabilidad	Limitada, sobre todo si el número de tareas aumenta.	Alta. Permite modularizar el código fácilmente.
Complejidad	Baja. El código es fácil de entender y de depurar.	Alta. Requiere mecanismos de sincronización.
Uso de recursos	Bajo consumo.	Mayor consumo de memoria y CPU, ya que debe ejecutar tareas en segundo plano.

Diagrama funcional





Terminología en RTOS

- **Tarea:** es un conjunto de instrucciones cargadas en la memoria que están listas para ser ejecutadas por el CPU. También puede significar alguna unidad de trabajo u objetivo que debe lograrse.
- **Subproceso o hilo:** es la unidad más pequeña de ejecución que puede ser gestionada de manera independiente por el programador (scheduler) del sistema operativo. Posee su propio contador de programa y memoria de pila.
- **Proceso:** instancia de un programa de computadora. Un proceso puede tener varios subprocesos.
- **Tick:** es una señal periódica generada por un temporizador de hardware que interrumpe el CPU a intervalos regulares de tiempo.

Estos términos pueden volverse más confusos dependiendo el RTOS que se haya elegido. A continuación, **adoptaremos la terminología de FreeRTOS porque es el sistema operativo que se ha seleccionado para trabajar**. Debemos recordar que FreeRTOS llama "tareas" a los "subprocesos".

Dentro del sistema operativo se pueden configurar rutinas de servicio de interrupción (ISR) que pueden adelantarse a cualquiera de las tareas que se están ejecutando. Generalmente, se lo utiliza para controlar desbordamientos del temporizador de hardware, cambios de estado de pines o comunicaciones en un bus.

A continuación, observaremos una imagen de cómo funcionan en conjunto los conceptos mencionados anteriormente:

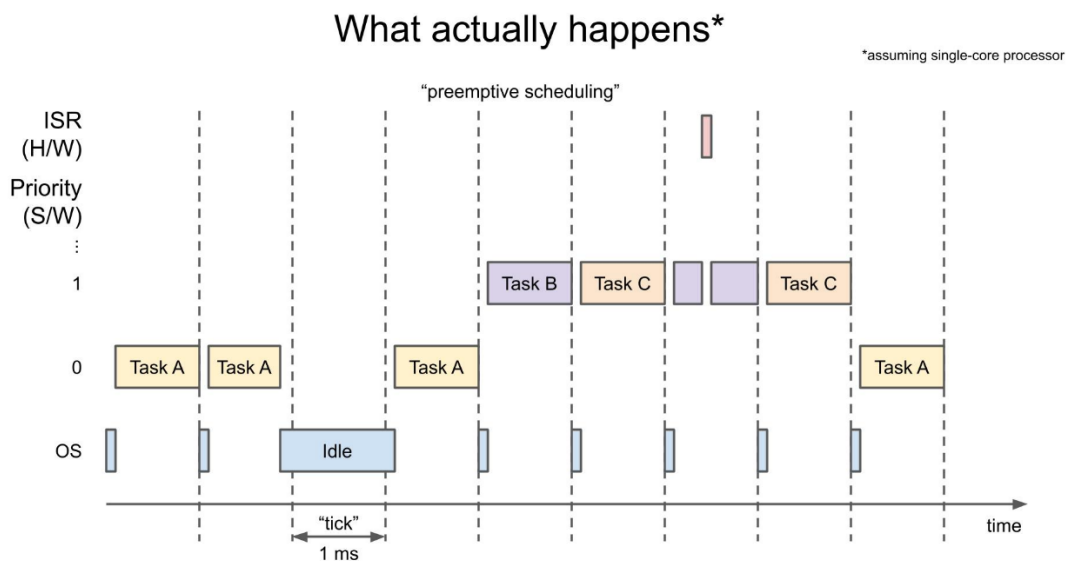


Figura 8 - Programador de tareas en un solo núcleo



Administración y Comunicación entre Tareas

Tarea

En FreeRTOS podemos dividir el término tarea en cuatro estados.

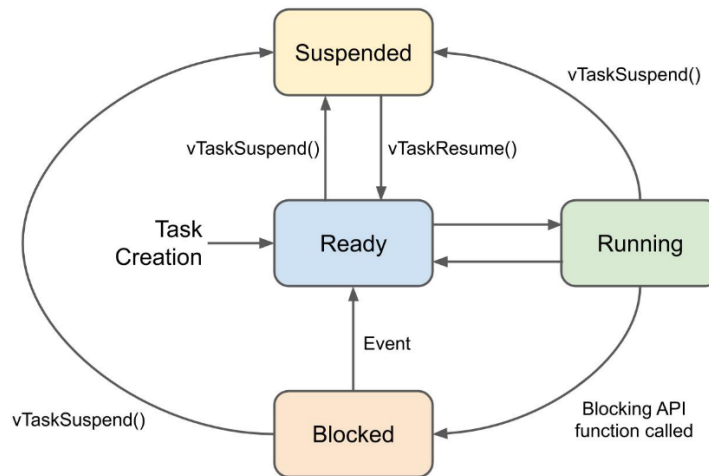


Figura 9 - Estados de una Tarea

Cuando se crea una tarea, se dice que está en el estado *Ready (Listo)*. Este estado le indica al programador que esa tarea está lista para ejecutarse. En cada tick, el programador elige una tarea para ejecutar que esté en estado listo (en un sistema multinúcleo, el programador puede elegir varias tareas).

Mientras se ejecuta una tarea se encuentra dentro del estado *Running (En ejecución)*. Luego, el programador puede devolverla al estado *Listo*.

Las funciones que hacen que la tarea espere, como `vTaskDelay()`, ponen la tarea en estado *Blocked (Bloqueado)*. Aquí, la tarea está esperando a que ocurra algún otro evento, como que expire el temporizador de `vTaskDelay()`. La tarea también puede estar esperando que otra tarea libere algún recurso, como un semáforo. Las tareas en estado *Bloqueado* permiten que se ejecuten otras tareas en su lugar.

Finalmente, una llamada explícita a `vTaskSuspend()` puede poner una tarea en modo (Suspendend) *Suspendido*. Cualquier tarea puede poner cualquier tarea (incluida ella misma) en modo *Suspendido*. Una tarea solo puede regresar al estado *Listo* mediante una llamada explícita a `vTaskResume()` por parte de otra tarea.



Cola

Es un sistema FIFO con lecturas y escrituras atómicas (operaciones que no pueden ser interrumpidas por otras tareas durante su ejecución).

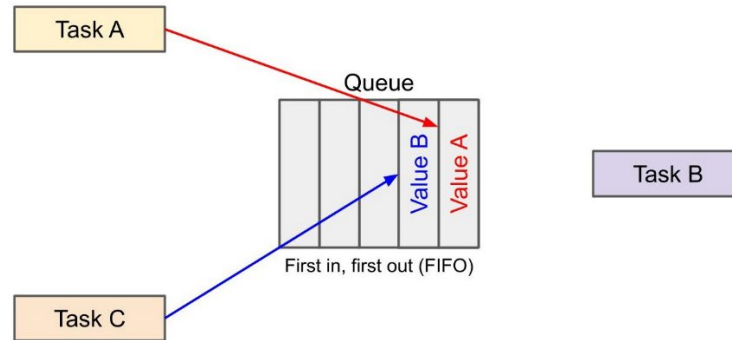


Figura 10 - Tareas en una cola

En FreeRTOS, la información se copia en una cola por valor y no por referencia y para ello se utiliza la función `xQueueSend()`. Esto puede resultar útil, pero puede requerir un tiempo de copia prolongado si se trata de un fragmento de datos grande, como una cadena.



Sincronización y control de acceso a recursos

Mutex

Es una bandera o bloqueo que se utiliza para permitir que solo un subproceso acceda a una sección de código a la vez. Se utiliza para garantizar seguridad a los subprocesos que se encuentren en una sección crítica del programa.

Para poder entender este concepto veremos un ejemplo donde dos tareas incrementan la misma variable global, por lo que cada tarea debe leer la variable global de la memoria, incrementarla y escribirla nuevamente en la memoria (ciclos de instrucciones separados).

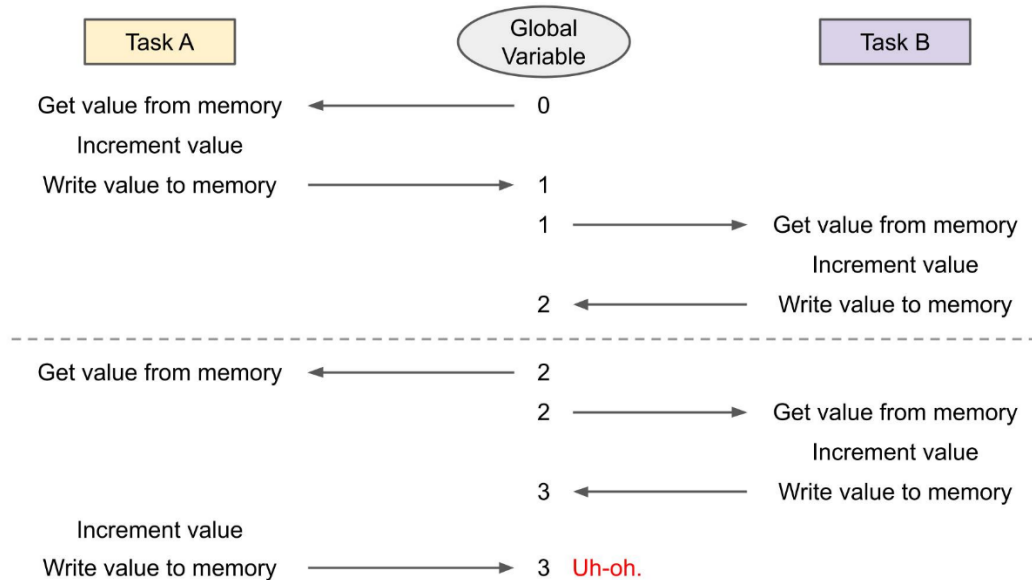


Figura 11 - Tareas incrementan una variable global

Podemos terminar con una situación en la que la *tarea A* lee el valor, la *tarea B* interrumpe para leer el mismo valor, incrementarlo y volverlo a escribir. Cuando la ejecución regresa a la *tarea A*, todavía tiene el valor original en la memoria local de trabajo. Incrementa ese valor y lo vuelve a escribir en la memoria, sobrescribiendo el trabajo de la *tarea B*. Esto da como resultado que se escriba el mismo valor en la memoria global, aunque se hayan ejecutado dos comandos de incremento. Esto se conoce como una “condición de carrera”, ya que el momento exacto de la secuencia de eventos para realizar una acción puede cambiar el resultado. Para eliminar esta condición de carrera vamos a utilizar un **mutex**.

En RTOS un mutex es simplemente un valor binario global al que se puede acceder de forma atómica. Esto significa que, si un subproceso toma el mutex, puede leer y decrementar el valor sin ser interrumpido por otros subprocesos. Si un subproceso ve que todavía no tiene el mutex, entra en estado de bloqueo o puede realizar algún otro trabajo antes de verificar el mutex nuevamente.



A continuación, se mantendrá el ejemplo anterior, pero se protegerá la variable global con un mutex.

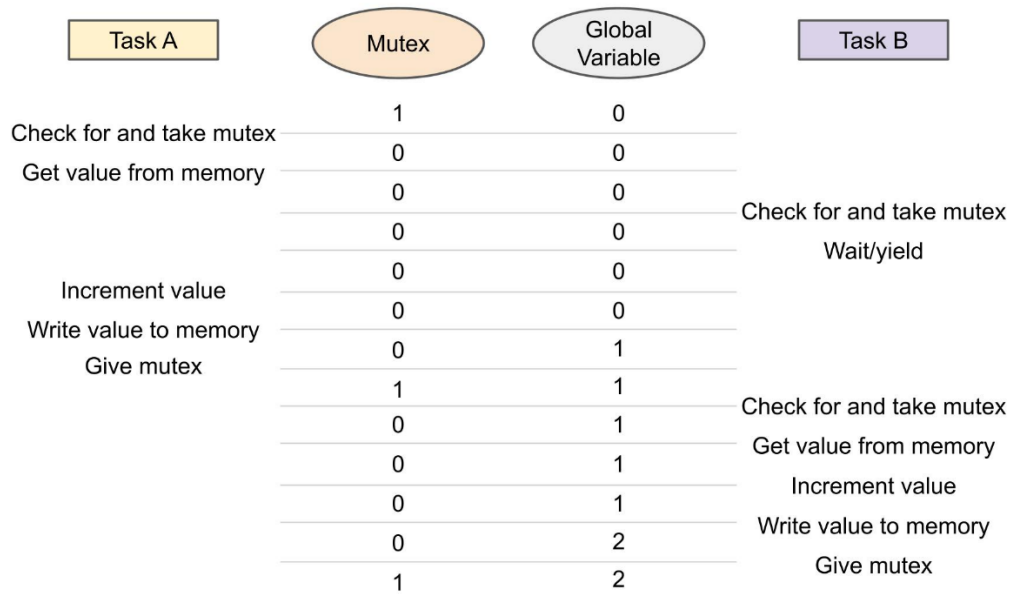


Figura 12 - Tareas incrementan una variable global con mutex

Cada línea denota una (o un grupo) de instrucciones ejecutadas en el procesador. En la imagen se puede ver que la *tarea A* solo puede incrementar la variable global mientras tiene el mutex. La *tarea B* puede interrumpirla, pero no puede ingresar a la sección crítica (ya que no hay ningún mutex disponible). Solo cuando la *tarea A* finaliza y devuelve el mutex, la *tarea B* puede ingresar a la sección crítica para incrementar la variable global.

Semáforo

Es una variable que se utiliza para controlar el acceso a un recurso compartido común al que deben acceder varios subprocesos o procesos. Es similar a un mutex pero permite que varias tareas ingresen a una sección crítica.

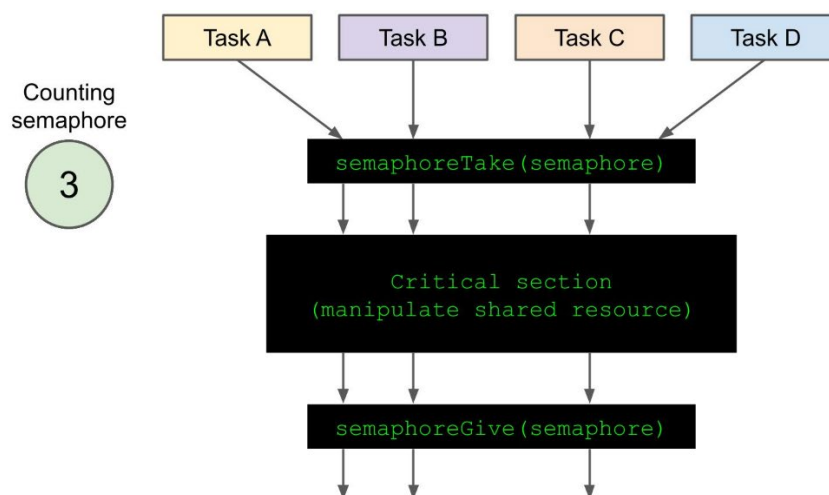


Figura 13 - Funcionamiento teórico del semáforo

En teoría, un **semáforo** es un contador compartido que se puede incrementar y decrementar de forma atómica. En la imagen anterior vemos un ejemplo donde las *tareas A, B* y *C* desean ingresar a la sección crítica. Para ello, cada una de ellas llama a `semaphoreTake()` (función que decrementa el contador del semáforo). En este punto, las 3 tareas están dentro de la sección crítica y el valor del semáforo es 0.

Si otra tarea, como la *Tarea D*, intenta ingresar a la sección crítica, primero debe llamar también a `semaphoreTake()`. Sin embargo, dado que el semáforo es 0, `semaphoreTake()` le indicará a la *Tarea D* que espere. Cuando cualquiera de las otras tareas abandone la sección crítica, debe llamar a `semaphoreGive()`, que incrementa atómicamente el semáforo. En este punto, la *Tarea D* puede llamar a `semaphoreTake()` nuevamente y entrar a la sección crítica.

Esto demuestra que los semáforos funcionan como un mutex generalizado capaz de contar hasta números mayores que 1. Sin embargo, en la práctica, rara vez se utilizan semáforos como este, ya que incluso dentro de la sección crítica, no hay forma de proteger recursos individuales con solo ese semáforo.

En la práctica, los semáforos se utilizan a menudo para indicar a otros subprocesos que hay algún recurso común disponible para su uso. Esto funciona bien en escenarios de productor/consumidor donde una o más tareas generan datos y otras tareas utilizan esos datos. A continuación, veremos un ejemplo.

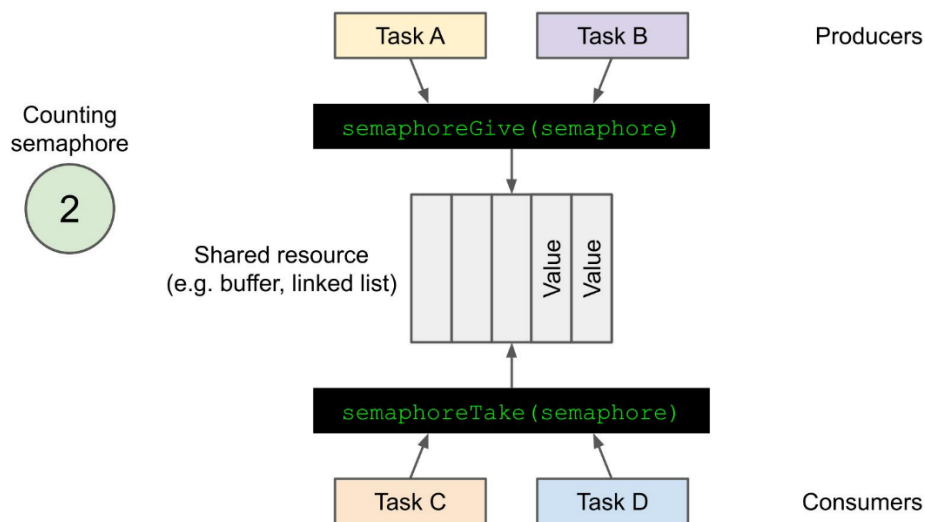


Figura 14 - Funcionamiento práctico del semáforo

Las *tareas A* y *B* (productores) crean datos que se colocarán en un recurso compartido, como un búfer o una lista enlazada. Cada vez que lo hacen, llaman a `semaphoreGive()` para incrementar el semáforo. Las *tareas C* y *D* (consumidores) pueden leer valores de ese recurso y eliminarlos. Cada vez que una tarea lee del recurso, llama a `semaphoreTake()`, que disminuye el valor del semáforo.

Luego, se utiliza un semáforo como señal adicional para las tareas del consumidor de que los valores están listos. Al limitar el valor máximo del semáforo, podemos controlar cuánta información se puede introducir o leer del recurso a la vez (es decir, el tamaño del búfer o la longitud máxima de la lista enlazada).

Diferencia entre un mutex y un semáforo

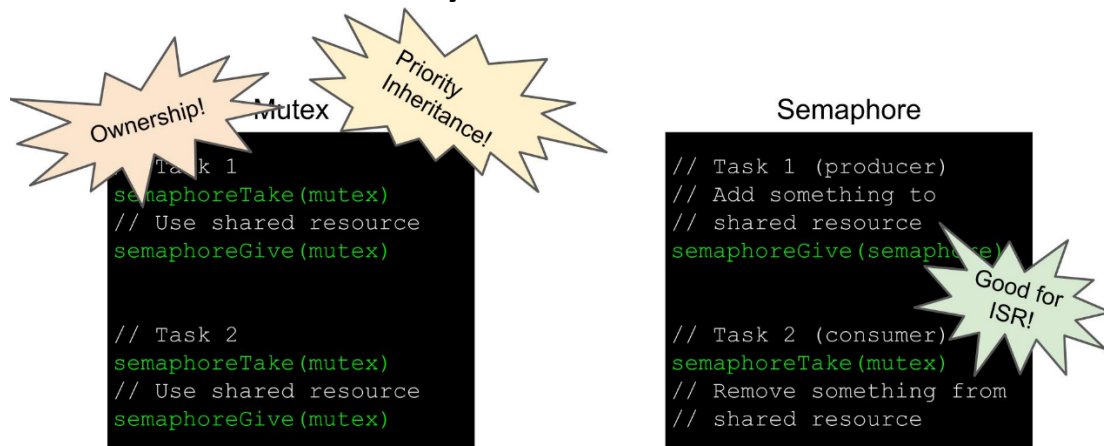


Figura 15 - Diferencia entre Mutex y Semáforo

Un **mutex** implica que el mismo subproceso debe "tomar" y "dar" el mutex durante la ejecución de la sección crítica. En cambio, un **semáforo** es dado y tomado por diferentes subprocesos, lo que significa que es mejor usarlo como un mecanismo de señalización para sincronizar subprocesos.

Generalmente, no se recomienda utilizar un mutex en una ISR dado que no es deseable que la rutina se bloquee. Sin embargo, podría utilizar un semáforo en una ISR para indicar a otros subprocesos que se ha ejecutado y que algunos datos están listos para su consumo.

Problemas de Concurrencia y Gestión de Prioridades

Deadlock y Starvation

Estos conceptos se pueden explicar con el problema de los filósofos cenando (Dining philosophers problem) planteado por Edsger Dijkstra en 1965. Este problema se basa en cinco filósofos sentados en una mesa redonda con un plato de fideos en el centro y un palillo entre cada filósofo. De esta manera, un filósofo solo puede comer si tiene dos palillos en la mano y cuando termina debe dejar los palillos para que coma otro filósofo.



Figura 16 - Problema de los filósofos cenando

La idea del algoritmo es permitir que todos los filósofos tengan la oportunidad de comer. En esta analogía, los filósofos son como tareas (o hilos) que intentan realizar algún trabajo con un recurso compartido (el plato de fideos). Los palillos son análogos a los bloqueos (semáforos o mutex) que se necesitan antes de acceder al recurso compartido. Si un par de filósofos tienen mayor prioridad que el resto y nunca ceden, entonces el resto de los filósofos (hilos) corren el riesgo de no comer nunca. Esto se conoce como **starvation**.



Figura 17 – Starvation en FreeRTOS



Existen algunas formas de combatir el starvation. La primera es simplemente asegurarse de que las tareas de alta prioridad siempre le den algo de tiempo al procesador (por ejemplo, esperando un mutex, un semáforo o una función de retardo). El segundo método es depender de una tarea de mayor prioridad para supervisar el tiempo que otras tareas han estado inactivas. Si una tarea de menor prioridad ha estado inactiva (bloqueada) durante algún tiempo, su nivel de prioridad se eleva gradualmente hasta que tiene la oportunidad de ejecutarse. Una vez que se ha ejecutado durante algún tiempo, su nivel de prioridad vuelve a su nivel original. Esto se conoce como "envejecimiento" y se puede ver en la siguiente figura:



Figura 18 - Envejecimiento en FreeRTOS

Puede existir otro problema. Supongamos que el algoritmo para cada filósofo es el siguiente:

- Tomar el palillo izquierdo.
- Tomar el palillo derecho.
- Comer un rato.
- Bajar el palillo izquierdo.
- Bajar el palillo derecho.

Si cada filósofo toma su palillo izquierdo y es interrumpido inmediatamente por otro filósofo que toma su palillo izquierdo, eventualmente llegaremos a un punto en el que todos los filósofos sostienen su palillo izquierdo. Pero, ninguno de los filósofos puede levantar su palillo derecho porque el filósofo a su derecha también está esperando. Esta situación se conoce como **deadlock o punto muerto** y el sistema se expresa deteniéndose mientras todas las tareas esperan circularmente que se liberen los bloqueos.

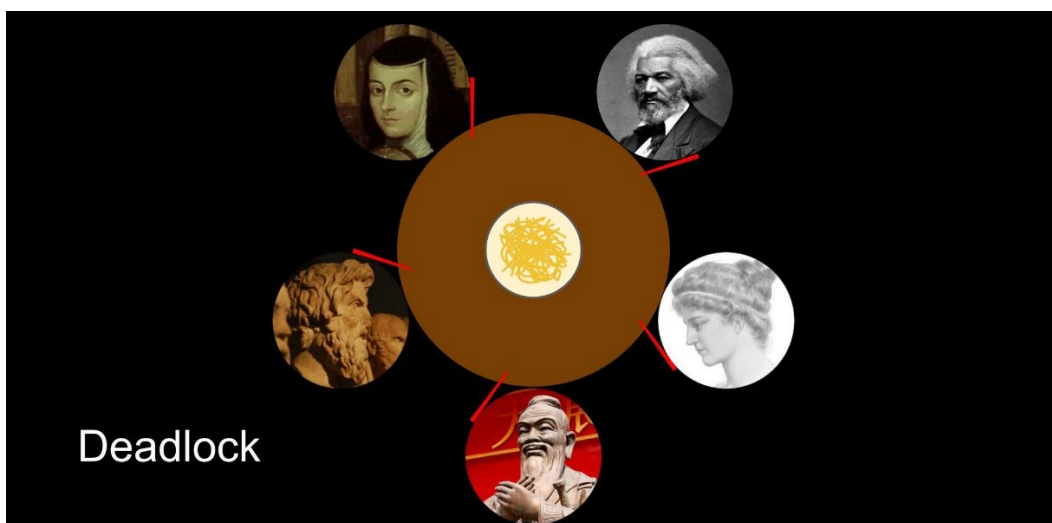


Figura 19 - Deadlock en FreeRTOS

Una forma de resolver este problema es dar un valor de prioridad o jerarquía a cada palillo. Esto evitará que se produzca un bloqueo (deadlock), ya que el último filósofo debe esperar a que regrese el palillo con el número más bajo antes de recoger otro palillo.

Otra solución implica un “camarero” o “árbitro”, que básicamente bloquea todos los palillos con un mutex global. De esta manera, cada vez que un filósofo desee tomar un palillo, primero debe solicitar permiso a este árbitro. El mutex (árbitro) disminuye de valor y permite al filósofo tomar ambos palillos para comer. A cualquier otro filósofo que desee tomar un palillo se le negará el permiso mientras el primer filósofo tenga el mutex.



Figura 20 y 21 - Soluciones al problema del filósofo

Esto funciona como una solución para evitar el deadlock, pero elimina cualquier ganancia de eficiencia que pudiéramos ver al permitir que varios filósofos coman al mismo tiempo (paralelismo).

Inversión de prioridades

La inversión de prioridad es un error que ocurre cuando una tarea de alta prioridad es reemplazada indirectamente por una tarea de baja prioridad. Por ejemplo, la tarea de baja prioridad contiene un mutex que la tarea de alta prioridad debe esperar para continuar ejecutándose. A continuación, se puede ver un caso simple de este problema:

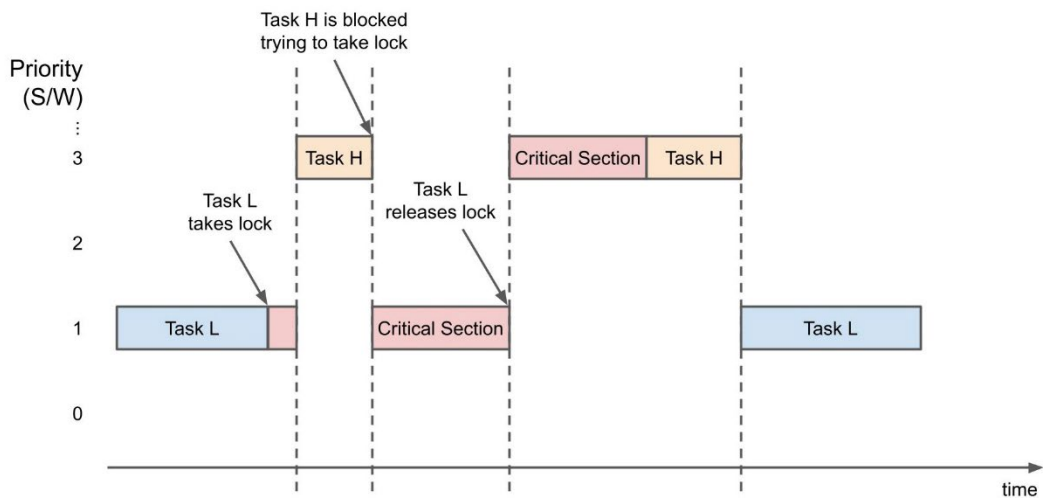


Figura 22 - Inversión de prioridad limitada



En la figura, la tarea de alta prioridad (*Tarea H*) está bloqueada mientras la tarea de baja prioridad (*Tarea L*) mantenga el bloqueo. Esto se conoce como **inversión de prioridad limitada**, ya que la duración de la inversión está limitada por el tiempo que la tarea de baja prioridad se encuentre en la sección crítica (manteniendo el bloqueo). La prioridad de las tareas se ha “invertido” indirectamente, ya que ahora la *tarea L* se ejecuta antes que la *tarea H*.

También existe la inversión de prioridad ilimitada que ocurre cuando una tarea de prioridad media (*Tarea M*) interrumpe a la *Tarea L* mientras mantiene el bloqueo. Se denomina **ilimitada** porque la *Tarea M* ahora puede bloquear efectivamente a la *Tarea H* por cualquier cantidad de tiempo, ya que la *Tarea M* está interrumpiendo a la *Tarea L* (que aún mantiene el bloqueo).

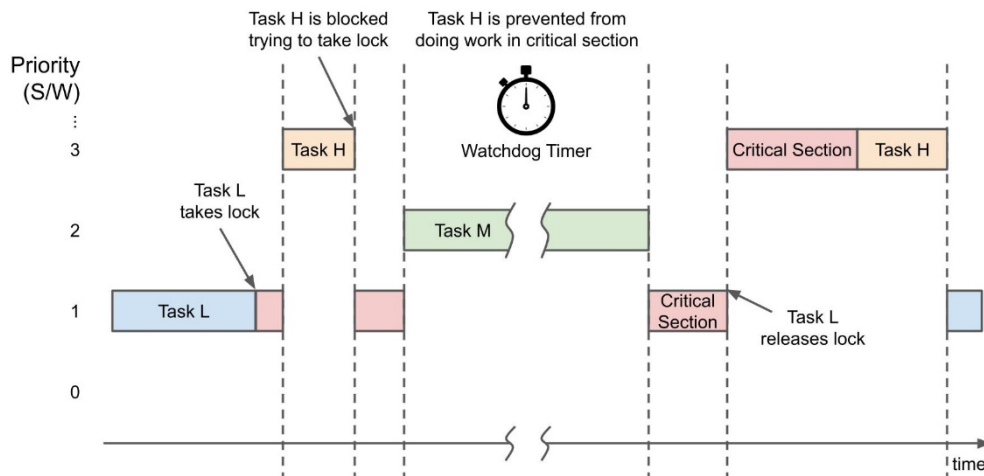


Figura 23 - Inversión de prioridad ilimitada

Un ejemplo real de inversión de prioridad se dio con la misión Mars Pathfinder en 1997.

Existen dos formas de combatir la inversión de prioridad ilimitada, a través del protocolo de límite de prioridad o de la herencia de prioridad.

El **protocolo de límite de prioridad** implica asignar un nivel de límite de prioridad a cada recurso o bloqueo. Siempre que una tarea trabaja con un recurso en particular o toma un bloqueo, el nivel de prioridad de la tarea aumenta automáticamente al límite de prioridad asociado con el bloqueo o recurso. El límite de prioridad está determinado por la prioridad máxima de cualquier tarea que necesite utilizar el recurso o bloqueo.

Priority Ceiling Protocol

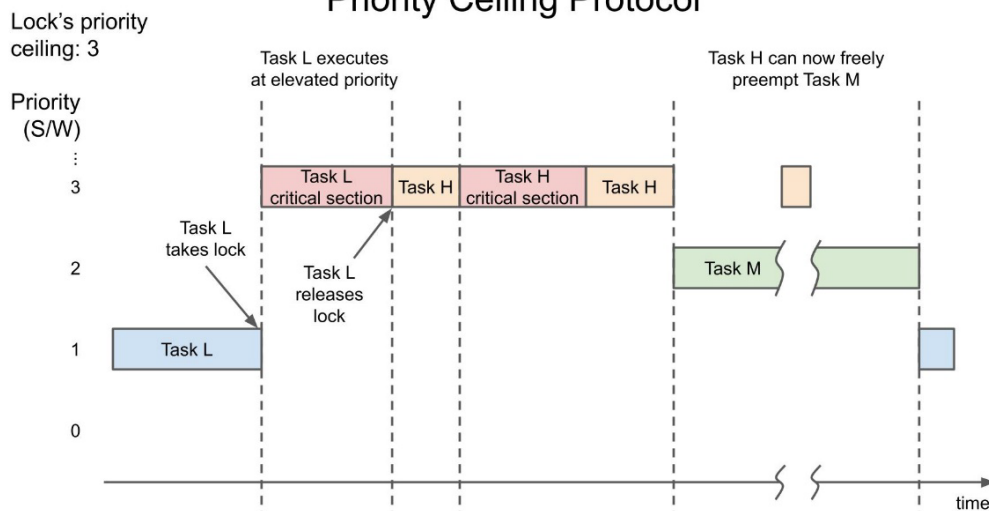


Figura 24 - Protocolo de límite de prioridad



Como el límite de prioridad del bloqueo es 3, cada vez que la *Tarea L* toma el bloqueo, su prioridad aumenta a 3 para que se ejecute con la misma prioridad que la *Tarea H*. Esto evita que la *Tarea M* (prioridad 2) se ejecute hasta que las *Tareas L* y *H* terminen con el bloqueo.

El otro método para combatir la inversión de prioridad ilimitada es utilizar la **herencia de prioridad**, la cual implica aumentar la prioridad de una tarea que tiene un bloqueo a la de cualquier otra tarea (de mayor prioridad) que intente tomar el bloqueo.

Priority Inheritance

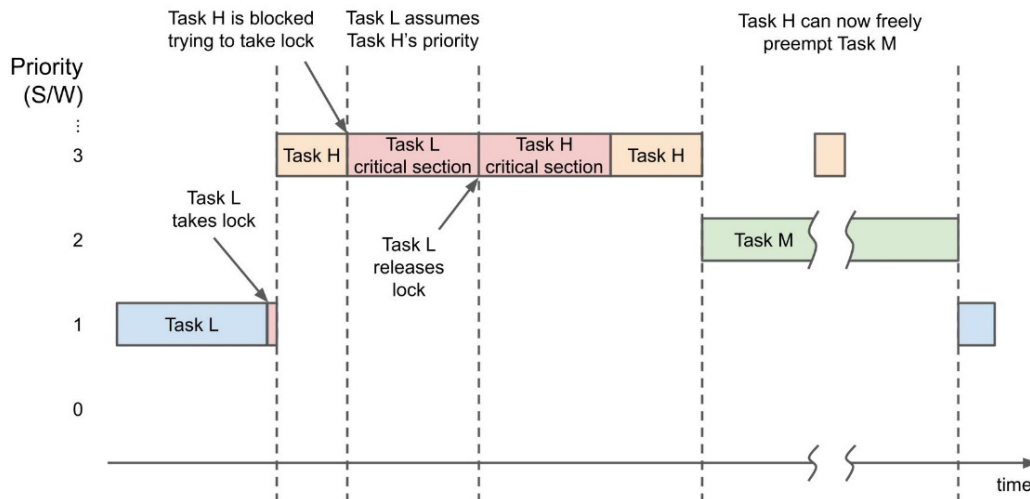


Figura 25 - Herencia de prioridad

En este caso la *tarea L* toma el candado. Solo cuando la *tarea H* intenta tomar el candado, la prioridad de la *tarea L* aumenta a la de la *tarea H*. Una vez más, la *tarea M* ya no puede interrumpir a la *tarea L* hasta que ambas tareas hayan finalizado en la sección crítica.

Se debe tener en cuenta que, en ambos métodos, la prioridad de la tarea *L* vuelve a su nivel original una vez que se libera el bloqueo y que la inversión de prioridad limitada puede ocurrir ocasionalmente. Dado que solo se puede mitigar este problema mediante buenas prácticas de programación como mantener las secciones críticas breves, evitar utilizar secciones críticas o mecanismos de bloqueo que puedan bloquear una tarea de alta prioridad, utilizar una tarea para controlar un recurso compartido para evitar la necesidad de crear bloqueos para protegerlo, entre otras.

Gestión de tiempo y eventos

Temporizadores de software

Los temporizadores en sistemas integrados permiten retrasar la ejecución de alguna función o ejecutar una función periódicamente. Pueden ser temporizadores de hardware, que son exclusivos de la arquitectura, o temporizadores de software que se basan en algún código en ejecución o en el temporizador de ticks del sistema operativo en tiempo real (RTOS).

En FreeRTOS, hay algunas formas de retrasar la ejecución de una función:

- `vTaskDelay()` permite bloquear la tarea que se está ejecutando actualmente por una cantidad de tiempo determinada (expresada en ticks).
- Se puede realizar un retraso sin bloqueo comparando `xTaskGetTickCount()` con alguna marca de tiempo conocida.
- Muchos microcontroladores (y microprocesadores) pueden configurar algunos registros para que cuenten hacia arriba o hacia abajo y activen una rutina de servicio de interrupción (ISR) cuando expiren (o alcancen un número determinado).
- Los temporizadores de software existen en el código y no dependen del hardware (excepto por el hecho de que el temporizador de ticks de RTOS generalmente depende de un temporizador de hardware).
- También, existe una API en FreeRTOS que facilita la gestión de temporizadores.

Cuando se incluye la biblioteca de temporizadores de FreeRTOS se ejecuta una tarea de servicio de temporizador ("timer Daemon") que se encarga de administrar todos los temporizadores y de llamar a las distintas funciones callbacks.

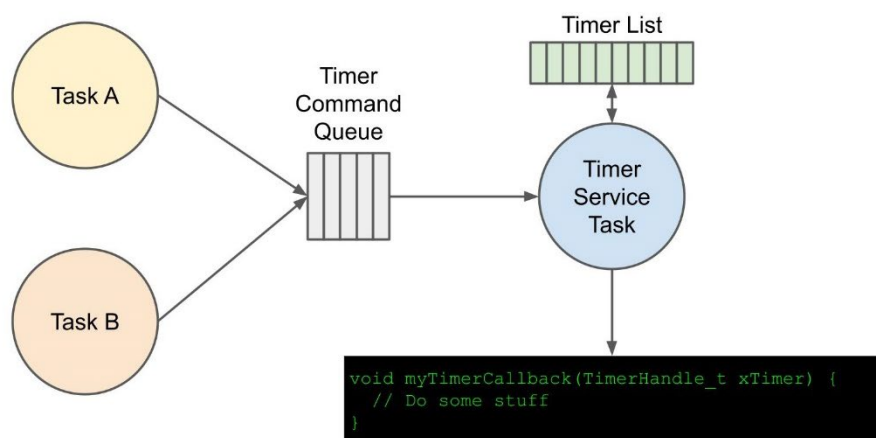


Figura 26 - Temporizador por software

Cuando se crea un temporizador, se le asigna una función (callback) que se llama cada vez que el temporizador expira. El temporizador de software puede ser de "una sola ejecución" (ejecuta el callback una vez) o de "recarga automática" (ejecuta el callback periódicamente). Se debe tener en cuenta que el tiempo de los temporizadores se basan en ticks (1 tick es aproximadamente igual a 1 ms).

Interrupciones de hardware

Las interrupciones de hardware son señales enviadas por dispositivos externos o periféricos que requieren atención inmediata del procesador. Permiten que los eventos se produzcan de forma asincrónica y notifican a la CPU que debe realizar alguna acción. Tienen la particularidad de detener el funcionamiento del CPU para que ejecute una rutina de servicio de interrupción.

Ejemplos de estas acciones pueden incluir cosas como presionar un botón (cambio de voltaje del pin de entrada), un temporizador de hardware que expira o un búfer de comunicación que se llena.

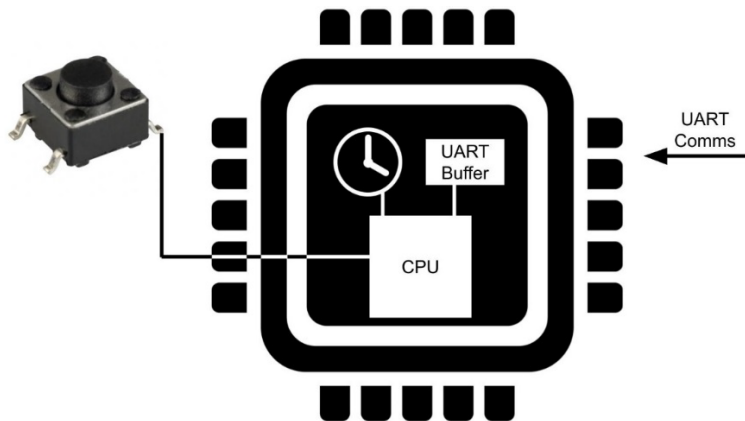


Figura 27 - Fuentes de Interrupción por hardware

En FreeRTOS, las interrupciones de hardware tienen una prioridad mayor que cualquier tarea. Cuando se trabaja con ellas, se deben tener en cuenta que nunca deben bloquearse a sí mismas (no pueden esperar colas, mutex o semáforos), su rutina debe ser lo más breve posible, si se requiere actualizar variables globales dentro de sus rutinas deben ser declaradas como "volatile".

Si se desea sincronizar una tarea con un ISR se puede utilizar una "interrupción diferida" desplazando el procesamiento de los datos capturados dentro del ISR a otra tarea. A continuación, veremos un ejemplo:

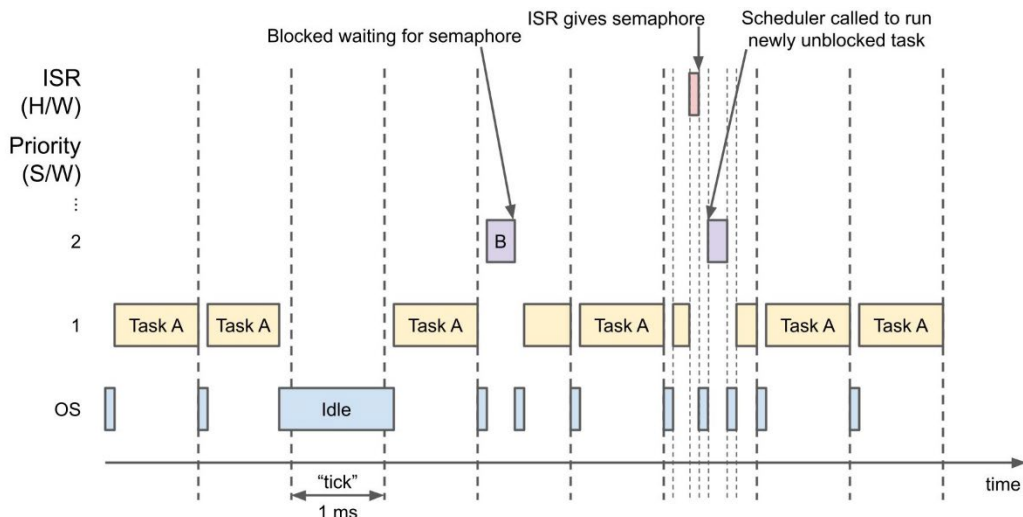


Figura 28 - Interrupción diferida

En este diagrama, la *tarea B* está bloqueada a la espera de un semáforo. Cuando el ISR se ejecuta (por ejemplo, para recopilar algunos datos de un sensor) proporciona el semáforo y cuando termina su ejecución, la *tarea B* se desbloquea inmediatamente y se ejecuta para procesar los datos recién recopilados.

Administración de memoria

Memoria

En la mayoría de los microcontroladores la memoria volátil (por ejemplo, RAM) se divide en tres secciones: *static* (estática), *stack* (pila) y *heap* (“montón”). Dependiendo el propósito, una variable puede ser asignada en una sección u otra. A continuación, vemos un ejemplo de cómo son asignadas las variables:

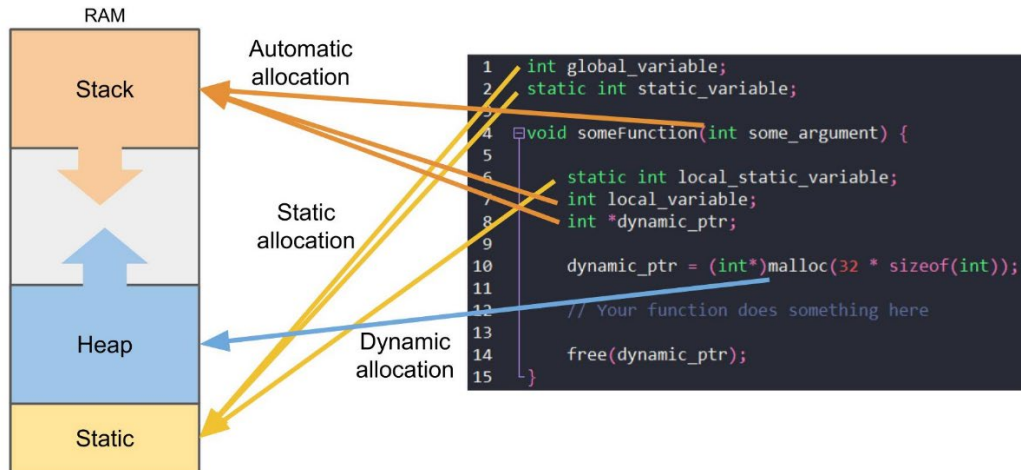


Figura 29 - Asignación de memoria convencional

En FreeRTOS, la asignación de memoria es distinta.

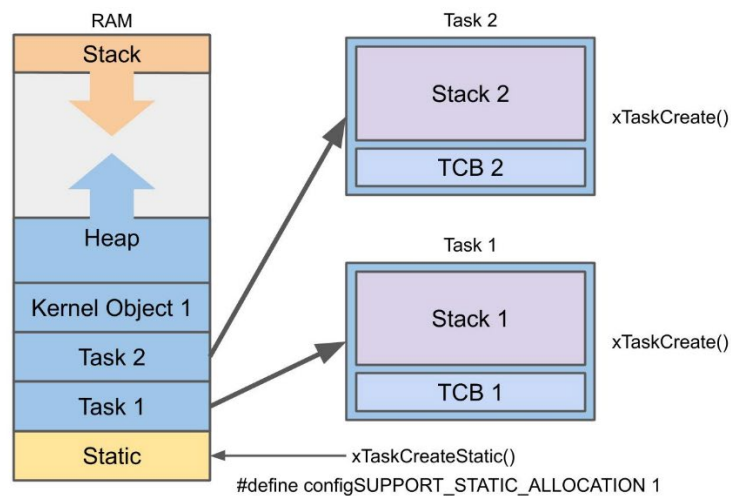


Figura 30 - Asignación de memoria en RTOS

En RTOS cuando se crea una tarea (por ejemplo, con `xTaskCreate()`), el sistema operativo asigna una sección de memoria del *heap* para esta tarea. La memoria asignada está formada por el *bloque de control de tareas (TCB)*, que se utiliza para almacenar información sobre la tarea (prioridad y el puntero de la pila local), y la otra sección está reservada como una *pila local* que funciona igual que la pila global (pero en una escala más pequeña solo para esa tarea).

Las variables locales creadas durante las llamadas a funciones dentro de una tarea se envían a la pila local de la tarea. Debido a esto, es importante calcular el uso de pila previsto de una tarea con anticipación e incluirlo como parámetro de tamaño de pila en `xTaskCreate()`.

Si se desea asignar variables al heap global del sistema se debe utilizar `pvPortMalloc()` y `vPortFree()`.



Bibliografía

[1] D. Sandler, "What is a Real-Time Operating System (RTOS)?", *Digi-Key Electronics*, 2022. [Online]. Disponible en: <https://www.digikey.com/en/maker/projects/what-is-a-realtime-operating-system-rtos/28d8087f53844decafa5000d89608016>. [Último acceso: Oct. 23, 2024].

[2] "About FreeRTOS", *FreeRTOS Documentation*. [Online]. Disponible en: <https://www.freertos.org/about-RTOS.html>. [Último acceso: Oct. 10, 2024].

[3] "FreeRTOS en ESP32/ESP8266: Multi-Tarea", *ElectrosoftCloud*, 2023. [Online]. Disponible en: <https://www.electrosoftcloud.com/freertos-en-esp32-esp8266-multi-tarea/>. [Último acceso: Sep. 24, 2024].

[4] U. Gaviria, "ESP32_FreeRtos", *GitHub Repository*, 2024. [Online]. Disponible en: https://github.com/uagaviria/ESP32_FreeRtos. [Último acceso: Sep. 19, 2024].