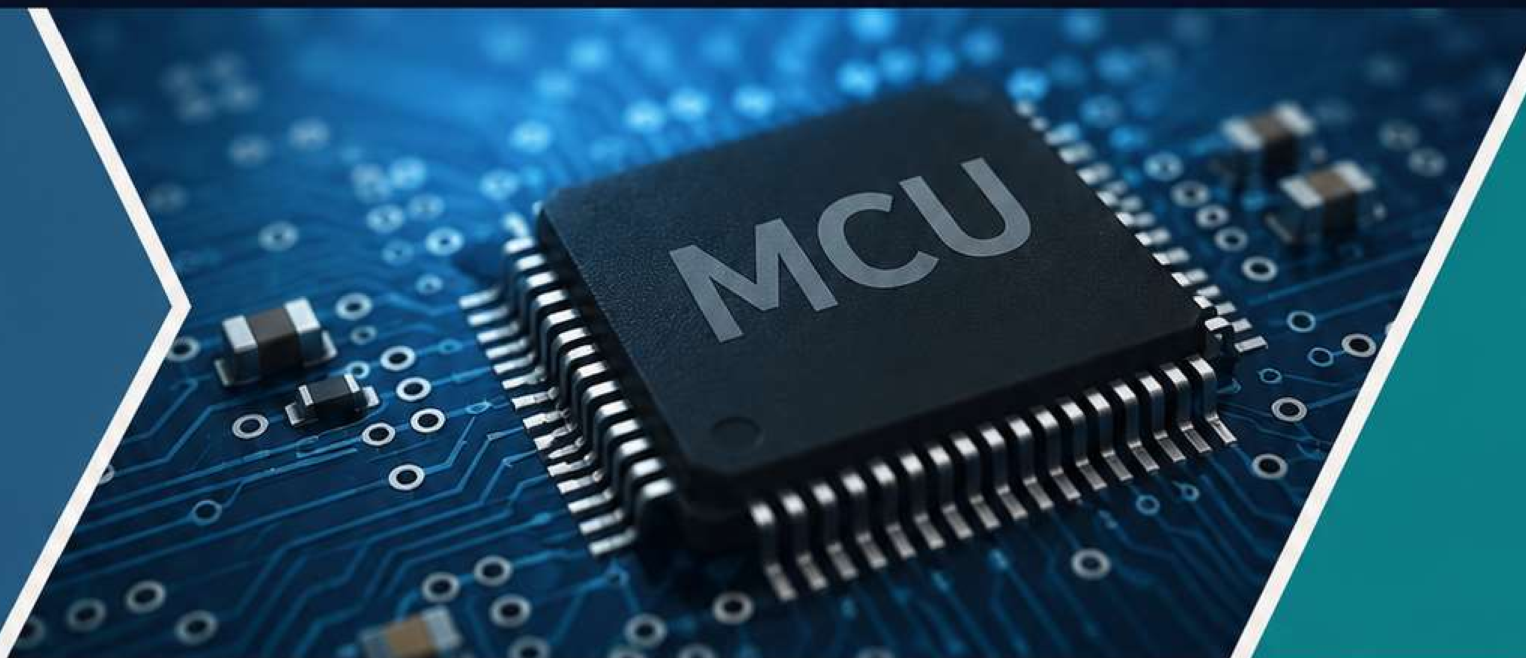
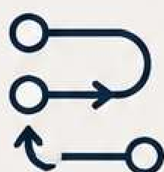
Firmware sin RTOS:

Multitarea cooperativa en sistemas embebidos





Conceptos clave, procesamiento cooperativo y ejemplos prácticos sin RTOS

FrameCode
Soluciones Tecnológicas

Fabián Romo

Firmware sin RTOS: Multitarea cooperativa en sistemas embebidos

Conceptos clave, multitarea cooperativa y máquinas de estado con ejemplos en C

Fabián Romo

Introducción

El presente libro es una guía práctica para escribir firmware de manera ordenada, permitiendo que un microcontrolador ejecute múltiples tareas de forma simultánea sin la necesidad de un RTOS.

Esto no significa que un RTOS no sea importante o inútil, todo lo contrario. Un RTOS es una herramienta poderosa y en muchos proyectos es la solución correcta. Sin embargo, no todos los proyectos lo requieren, y agregar un RTOS a un sistema de complejidad media o baja puede ser más problema que solución: mayor consumo de memoria, mayor complejidad de depuración, y una curva de aprendizaje considerable.

Este libro nació de la experiencia propia de haber aprendido estos conceptos de manera dispersa, sin que existiera una referencia clara que los reuniera en un solo lugar. Todo lo que aquí se explica fue aprendido a base de prueba, error e investigación.

¿Para quiénes es este libro?

En cierta manera, este libro fue escrito para mí mismo. Representa todo lo que me hubiera gustado tener cuando estaba aprendiendo estos conceptos a base de prueba y error, buscando información dispersa en documentaciones, foros y ejemplos que pocas veces explicaban el porqué de las cosas.

Si te encuentras en esa misma situación — ya sabes programar microcontroladores, pero sientes que tu firmware podría estar mejor estructurado — este libro es para ti."

¿Qué es lo que deberías conocer antes de leer este libro?

Se asume que el lector ya tiene experiencia básica con algún microcontrolador y conocimientos de programación en C. Los ejemplos están escritos en C, pero los conceptos son aplicables a cualquier lenguaje y cualquier plataforma."

¿Qué herramientas de software y hardware se utilizará?

Ninguna en particular, y esa es una decisión intencional.

La idea central de este libro es que los conceptos aquí explicados sean aplicables a cualquier microcontrolador, cualquier compilador y cualquier entorno de desarrollo. Haber elegido una plataforma específica habría convertido este libro en una guía más de un MCU particular, con el riesgo adicional de quedar desactualizado cada vez que el fabricante actualice sus herramientas — algo que ocurre con más frecuencia de lo deseable.

Por otro lado, intentar cubrir múltiples plataformas habría hecho el libro innecesariamente voluminoso sin resolver el problema de fondo: siempre quedarían plataformas fuera.

Los ejemplos de código están escritos en C estándar, sin depender de librerías ni herramientas específicas de ningún fabricante. El lector es libre de tomar esos conceptos y adaptarlos al microcontrolador, compilador y plataforma que ya conoce y utiliza en su trabajo.

Acerca del autor

Mi relación con los microcontroladores empezó en la universidad, y desde el primer momento supe que era lo mío. Hay algo fascinante en esas pequeñas máquinas capaces de controlar el mundo físico con unas pocas líneas de código, y esa fascinación no ha desaparecido con los años.

Empecé como muchos: con un PIC16F877 y lenguaje ensamblador, convencido de que programar en alto nivel era hacer trampa. Con el tiempo, la realidad de los proyectos me fue cambiando de opinión. Pasé por el dsPIC30F, luego por el C18 de Microchip, y eventualmente llegué a los PIC32 y los SAM de 32 bits, que son las plataformas con las que trabajo hasta hoy.

No soy un experto en todo. Hay temas en este libro que domino profundamente porque los he aplicado en proyectos reales, y hay otros donde mi conocimiento tiene límites. Lo que sí puedo garantizar es que todo lo que aquí se explica viene de la experiencia directa, no de repetir lo que dice la documentación oficial.

Si encuentras algún error o algo que podría explicarse mejor, será bienvenido.

Sugerencias y comentarios.

Cualquier sugerencia o comentario las pueden realizar al siguiente correo electrónico: f_romo_rivera@outlook.com

Fabián Rodrigo Romo.

Organización de cada capítulo

El libro está estructurado de manera progresiva: los primeros capítulos establecen los conceptos fundamentales, y los siguientes muestran cómo aplicarlos de forma práctica y ordenada.

Capítulo 1. Conceptos básicos de sistemas.

Se explica qué es un sistema embebido y se introducen los conceptos fundamentales que el lector necesita comprender antes de avanzar: concurrencia, determinismo, condiciones de carrera, recursos del microcontrolador, procesamiento cooperativo y apropiativo, entre otros.

Capítulo 2. Tareas síncronas y asíncronas.

Se explica la diferencia entre una tarea síncrona y una asíncrona, por qué las tareas síncronas pueden ser un problema en un sistema con múltiples procesos, y por qué el procesamiento asíncrono es la base del enfoque cooperativo que propone este libro.

Capítulo 3. Retardos asincrónicos

Los retardos son necesarios en casi cualquier sistema con microcontroladores, pero implementados de manera incorrecta bloquean el CPU e impiden que otras tareas se ejecuten. Este capítulo explica qué son los retardos asíncronos, cómo crearlos y cómo utilizarlos correctamente dentro de un sistema cooperativo.

Capítulo 4. Máquinas de estado.

Las máquinas de estado son la herramienta central de este libro. Se explica qué son, cómo se diseñan y por qué permiten estructurar el firmware de manera ordenada, predecible y fácil de mantener.

Capítulo 5. Arquitectura del firmware: el Super Bucle y organización de tareas.

Se explica cómo organizar el código de un proyecto real: cómo estructurar cada tarea en su propio módulo, cómo el Super Loop las coordina, y cómo esta arquitectura permite escalar el firmware sin que se vuelva inmanejable.

Capítulo 6. Bucles largos sin bloqueo.

Los bucles anidados de larga duración, como los que se usan al manejar displays o memorias, pueden bloquear el CPU durante milisegundos o incluso segundos. Este capítulo muestra cómo descomponer esos bucles en máquinas de estado para que el sistema siga respondiendo mientras se ejecutan. Los bucles anidados de larga duración, como los que se usan al manejar displays o memorias, pueden bloquear el CPU durante milisegundos o incluso segundos. Este capítulo muestra cómo descomponer esos bucles en máquinas de estado para que el sistema siga respondiendo mientras se ejecutan.

Capítulo 7. Interrupciones y periféricos independientes del CPU.

No toda la multitarea depende del procesador. Este capítulo explica cómo las interrupciones y los periféricos independientes del CPU complementan el

modelo cooperativo, y cómo integrarlos sin comprometer la estabilidad del sistema.

Capítulo 8. Depuración en sistemas cooperativos.

Encontrar errores en un sistema con múltiples tareas es diferente y más desafiante que en código secuencial. Este capítulo reúne técnicas y consejos prácticos para diagnosticar problemas

Capítulo 9. Ejemplos.

Con todos los conceptos establecidos, este capítulo los reúne en ejemplos concretos escritos en C estándar, diseñados para ilustrar cómo se aplica el modelo cooperativo en situaciones reales. Los ejemplos son independientes de cualquier plataforma y pueden adaptarse al microcontrolador que el lector ya utiliza.

Capítulo 1

Conceptos básicos de Sistemas Embebidos

1.1 ¿Qué es un sistema embebido?

Un sistema es difícil de definir con una sola frase, pero en términos prácticos podemos decir que es un conjunto de partes que trabajan juntas para resolver un problema concreto. Para entenderlo mejor, pensemos en un ejemplo cotidiano.

Imaginemos un horno microondas. Desde afuera parece simple: el usuario selecciona un programa, presiona un botón y el horno hace su trabajo. Pero por dentro, el microcontrolador está gestionando simultáneamente varios procesos. Figura 1.1.

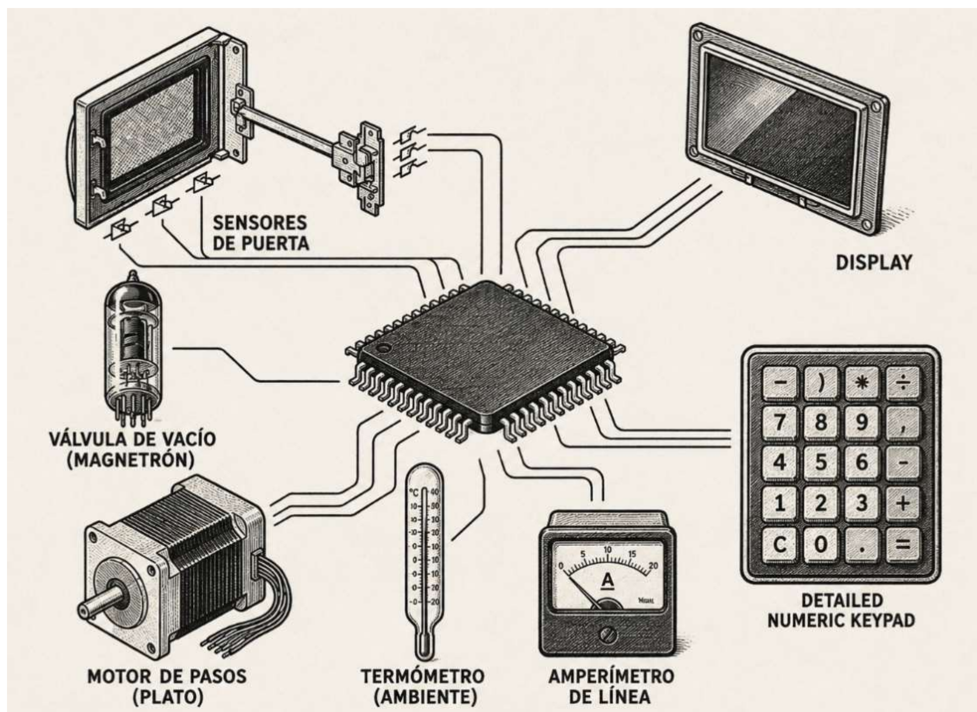


Figura 1.1 Un microcontrolador hace varios procesos a la vez en un horno microondas.

Cada uno de estos procesos debe ejecutarse de manera coordinada y sin interferir con los demás. A esto se le denomina multitarea, y es precisamente el tema central de este libro.

1.2 ¿Qué es una tarea?

Una tarea es un conjunto de instrucciones que el CPU del microcontrolador ejecuta para resolver una parte específica del problema. En el caso del horno microondas, el sistema requiere al menos las siguientes tareas:

- Monitoreo del sensor de la puerta: si la puerta se abre durante el funcionamiento, el sistema debe detener el calefactor de inmediato.
- Monitoreo de la temperatura interna.
- Monitoreo de la corriente del sistema de potencia: si supera un nivel crítico, debe desconectarse la salida para proteger el equipo.
- Control del sistema calefactor.
- Control del motor del plato giratorio.
- Control del teclado.
- Control de la pantalla.

Nótese que algunas de estas tareas son críticas: si el monitoreo de la puerta o de la corriente falla, aunque sea por unos pocos milisegundos, el sistema puede volverse peligroso. Esto ilustra perfectamente por qué importa cómo está estructurado el firmware — un código mal organizado que bloquee el CPU en el momento equivocado puede tener consecuencias reales.

El firmware de un microcontrolador debe ser legible, fácilmente modificable y dividido en módulos independientes. Idealmente, un programador puede encargarse del control de la pantalla mientras otro trabaja en el control del motor, sin que su código interfiera con el del otro.

Para lograr eso, se necesita una manera ordenada de organizar y coordinar todas esas tareas.

1.3 ¿Qué es una multitarea?

Imaginémonos conduciendo un automóvil mientras atendemos una llamada telefónica e intentamos comer algo. Es **irresponsable** y **peligroso**, pero ilustra perfectamente el concepto de multitarea: realizar varias cosas al mismo tiempo. Figura 1.2.

Figura 1.2. Multitarea al manejar, la ilusión de hacer todo a la vez.

Sin embargo, hay algo importante en ese ejemplo que vale la pena notar: en realidad nuestro cerebro no hace todo simultáneamente. Lo que hace es cambiar su atención de una tarea a otra tan rápidamente que da la ilusión de simultaneidad. Conduce, luego atiende la llamada por un instante, luego

da un mordisco, y vuelve a conducir. Todo tan rápido que parece en paralelo.

Un microcontrolador funciona de manera similar. En el ejemplo del horno microondas, el MCU debe vigilar el sensor de la puerta, monitorear la temperatura, controlar el motor del plato giratorio, gestionar el sistema de potencia, detectar si el usuario presionó algún botón y actualizar la información en la pantalla. Aparentemente todo al mismo tiempo, pero en realidad atendiendo cada tarea de acuerdo a un esquema predeterminado. Ese esquema es lo que se conoce como planificador, o ***Scheduler*** en inglés.

1.4 ¿Qué es *Deadline*?

Retomando el ejemplo del microondas: imaginemos que la tarea encargada de medir el tiempo de cocción detecta que se ha alcanzado el tiempo programado por el usuario. En ese instante, el proceso de calentamiento debe detenerse. Si no lo hace a tiempo, el alimento se quema. Ese límite de tiempo en el que una tarea debe haber cumplido su función se denomina ***deadline***. Figura 1.3

No todas las tareas tienen el mismo nivel de urgencia. Dependiendo de las consecuencias de no cumplir el ***deadline***, se distinguen dos tipos:

***Deadline* absoluto (*Hard Real-Time*)** Es aquel que no puede superarse bajo ninguna circunstancia. La tarea que monitorea la corriente del sistema de potencia es un ejemplo claro: si detecta una sobrecorriente, debe actuar en cuestión de milisegundos. Un retraso podría significar un daño en el equipo o incluso un incendio. En estos casos no hay margen de tolerancia.

Figura 1.3. Fallo de *deadline*, correr rápido no sirve de nada, te has atrasado a tu trabajo.

Deadline flexible (*Soft Real-Time*) Es aquel donde un pequeño retraso es

EDICIÓN DE MUESTRA

prioridad adecuada. Una tarea crítica que no cumple su *deadline* puede tener consecuencias graves. Una tarea no crítica que se retrasa levemente, en cambio, generalmente no compromete el funcionamiento del sistema.

Este concepto será clave cuando se explique cómo organizar y priorizar las tareas en los capítulos siguientes.

1.5 ¿Qué es Prioridad?

No todas las tareas son igual de importantes. La prioridad es el mecanismo que determina qué tan urgente es que una tarea se ejecute y cumpla con su *deadline* en relación con las demás.

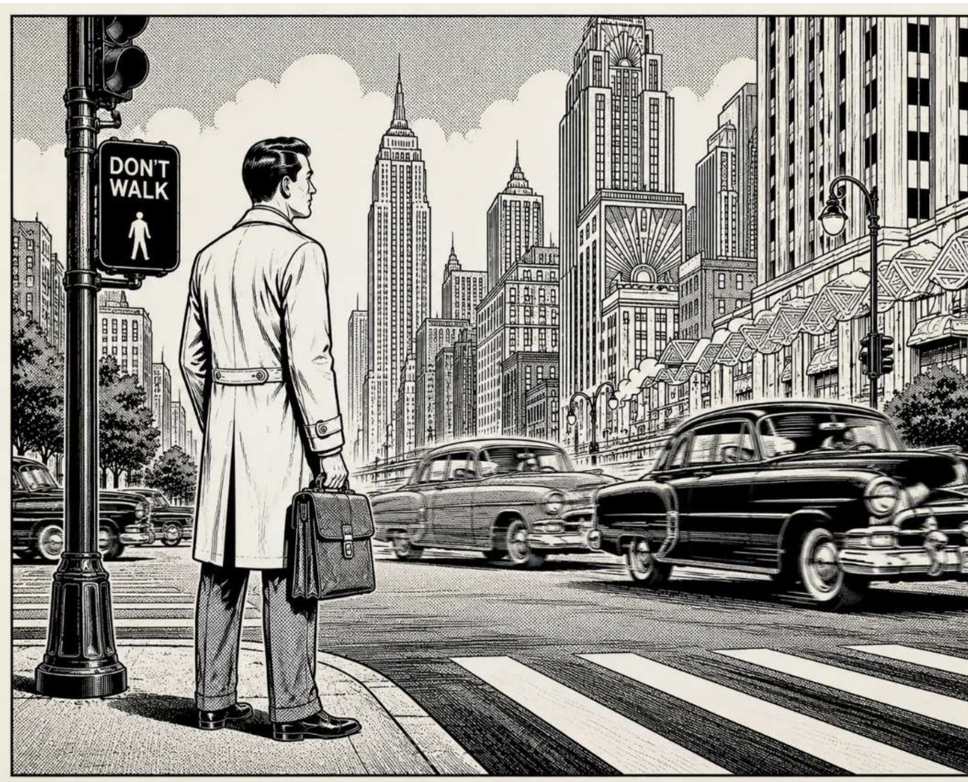


Figura 1.4. Preeminencia, el tráfico tiene prioridad, el peatón debe esperar.

En el ejemplo del horno microondas, la tarea que monitorea la corriente

La Figura 1.5 muestra un diagrama donde el eje horizontal representa el tiempo y el eje vertical el nivel de prioridad de cada tarea.

Figura 1.5. Una tarea que sea de alta prioridad respecto a otra puede interrumpirla.

Al inicio, se está ejecutando una tarea de muy baja prioridad (verde). En un momento dado, surge una tarea de baja prioridad (amarillo) que interrumpe a la verde y toma el control del CPU. Poco después, aparece una tarea de alta prioridad (rojo) que a su vez interrumpe a la amarilla.

Una vez que la tarea roja finaliza su proceso, cede el control a la tarea amarilla, que retoma su ejecución desde donde fue interrumpida. Finalmente, cuando la tarea amarilla concluye, la tarea verde reanuda su proceso.

1.6 ¿Qué es un Planificador?

1.7 ¿Qué es un RTOS?

Un sistema operativo en tiempo real, o RTOS, gestiona la planificación de múltiples tareas para todos los procesos que requiere el sistema embebido. Mantiene los *deadlines* y la funcionalidad dentro de las restricciones de tiempo definidas por quien escribió el código. Figura 1.7.

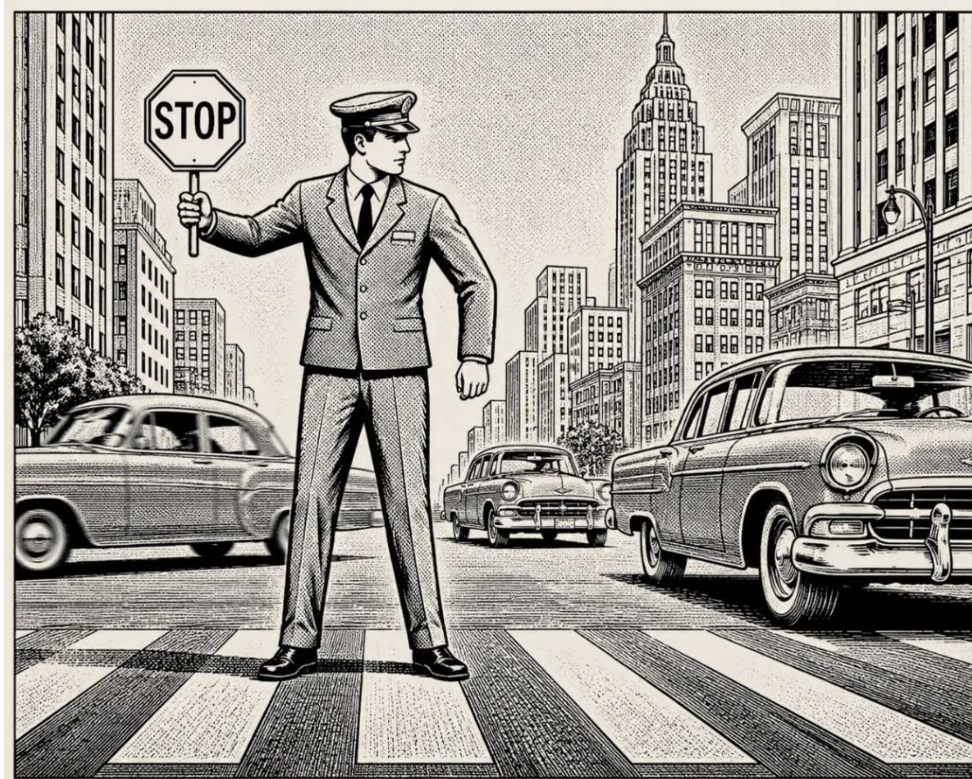


Figura 1.6. Planificador de tareas: decide qué proceso avanza y cuál debe esperar

El RTOS abstrae la posible complejidad del desarrollo: el programador escribe las tareas que requiere la aplicación y las integra en el RTOS, que se encarga de gestionar los requisitos de hardware necesarios para cada una de ellas.

Mediante este diseño, es más sencillo desarrollar sistemas de gran complejidad en los que el programador puede no tener un conocimiento profundo de los controladores (drivers) o las *API*s utilizadas.

El RTOS determina la mejor manera de distribuir el uso de los recursos del microcontrolador — CPU, memoria, periféricos, entre otros — y gestiona todos los datos sensibles al contexto, así como las interacciones entre tareas.

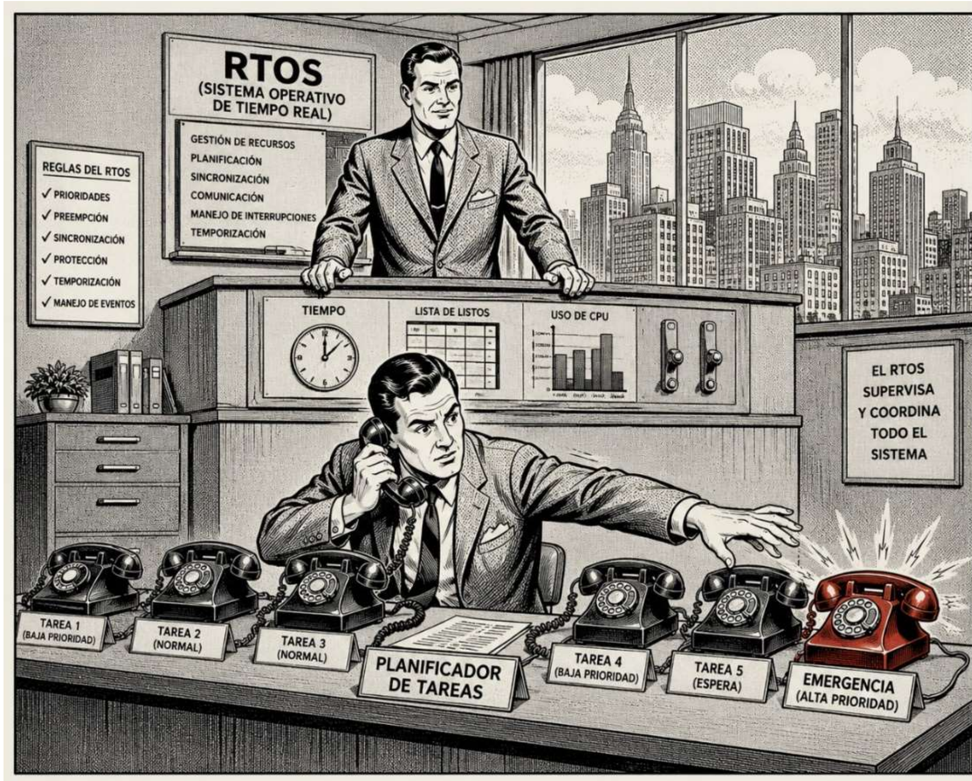


Figura 1.7. RTOS administra todo el hardware de un dispositivo para garantizar que las respuestas ocurran en un tiempo **exacto y predecible**.

En otras palabras, todo lo que se ha comentado anteriormente —

EDICIÓN DE MUESTRA

para el sistema en cuestión (***scaled up***).

La principal ventaja de una arquitectura basada en RTOS es que se encarga de la administración de recursos y memoria sin que el desarrollador de firmware tenga que preocuparse por ello. Gestiona la planificación y el funcionamiento de las tareas, abstrayendo estas complejidades para el desarrollador.

EDICIÓN DE MUESTRA

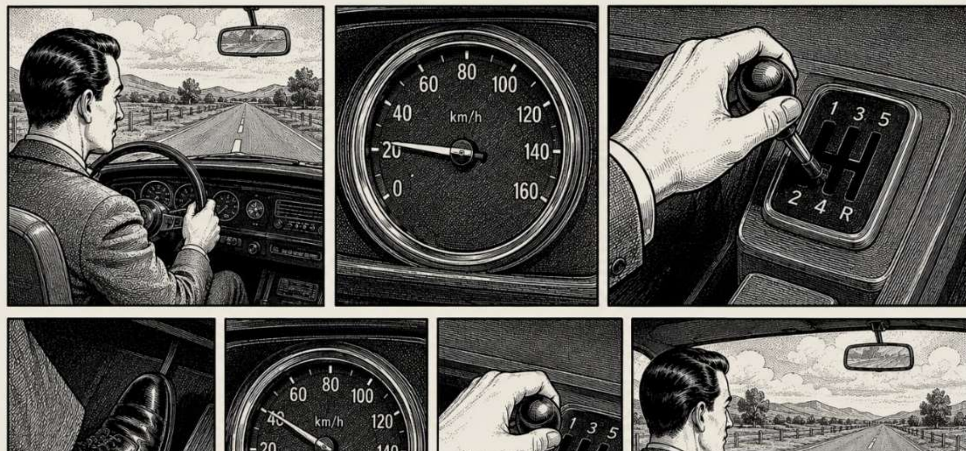
Por último, la configuración del RTOS sigue dependiendo del conocimiento que tenga el desarrollador sobre sus comandos, su uso y la arquitectura del sistema operativo en cuestión. En otras palabras, el RTOS reduce la complejidad de la planificación, pero introduce su propia curva de aprendizaje.

1.8 ¿Qué es Concurrencia?

La concurrencia se define como un proceso que se ejecuta de forma determinista desde un inicio definido hasta un final con resultado esperado. En otras palabras, es el orden en que las cosas van de A a Z dentro de una misma tarea. Puede tratarse de una única acción independiente o de un conjunto de acciones que deben ejecutarse en un orden determinado — de A a B, de B a C, y así sucesivamente — hasta llegar a una conclusión.

Pongamos un ejemplo. Supongamos que estoy conduciendo un automóvil en una carretera solitaria a 20 km/h y noto que voy demasiado lento, así que decido aumentar la velocidad.

Lo primero que hago es mirar el velocímetro para confirmar que la velocidad es baja. Luego realizo un cambio de marcha para dar más potencia al motor y comienzo a presionar el acelerador. Mientras lo hago, alterno la mirada entre el frente de la carretera y el velocímetro para verificar que la velocidad vaya aumentando hasta llegar a 40 km/h. Finalmente, al alcanzar la velocidad deseada, es posible que realice otro cambio de marcha para reducir la potencia del motor y mantener una velocidad constante. Figura 1.8



EDICIÓN DE MUESTRA

Sin embargo, la situación se complica cuando deben gestionarse múltiples

tareas simultáneamente. Esto nos lleva al ejemplo del conductor que, además de conducir y aumentar la velocidad, intenta atender una llamada telefónica y comer algo al mismo tiempo.

En resumen, la concurrencia es determinista en su proceso — cada paso depende del anterior —, produce el resultado esperado si se ejecuta sin variaciones ni interrupciones, y se vuelve compleja cuando varias tareas concurrentes deben ejecutarse al mismo tiempo, interactuando o no entre sí.

Analicemos otro ejemplo. Un proceso concurrente puede ser el de conducir desde el hogar hasta el lugar de trabajo. Es una acción que muchas personas realizan habitualmente y los pasos que siguen son bastante similares entre sí. Por lo tanto, este proceso es relativamente determinista. Figura 1.9

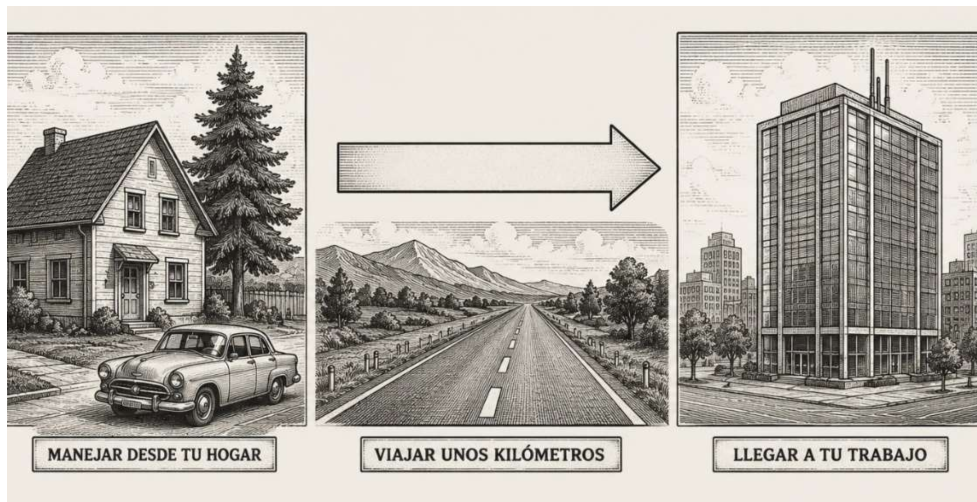


Figura 1.9. Conducir a tu trabajo puede considerarse un proceso determinista.

Los pasos son conocidos: salir del hogar, subir al automóvil, encender el motor y conducir por la ruta habitual hasta llegar al destino. Pueden existir variaciones externas que afecten el proceso, pero de eso se hablará más adelante.

En este ejemplo, la acción concurrente es conducir hasta el trabajo. Los actores son el conductor — que puede ser cualquier persona, como una variable — y el automóvil, que es el medio para ejecutar el proceso, como una función. El resultado final es llegar al trabajo.

Es evidente que la variable 'conductor' puede cambiar, pero la acción concurrente en sí no cambia, y tampoco su resultado. Para usted, el trayecto puede tomar 40 minutos. Para un compañero que vive cerca de la oficina, puede tomar 10 minutos. Sin embargo, esos tiempos serán consistentes para cada conductor bajo condiciones similares.

Esto se debe a que el proceso concurrente de conducir es determinista: los pasos que debe seguir cada persona no cambian, y por lo tanto los resultados son predecibles y esperados.

1.8 ¿Qué son Recursos Compartidos?

¿Qué ocurre cuando tu compañero de trabajo y tú deben compartir un recurso como la carretera? Ambos están ejecutando simultáneamente el mismo proceso: conducir hasta el trabajo.

Los dos desean llegar lo más rápido posible, pero al compartir la carretera y encontrarse en un cruce, si ninguno cede el paso, habrá un problema. Esto es exactamente lo que ocurre en un microcontrolador cuando dos tareas necesitan el mismo recurso al mismo tiempo.

En el caso del microcontrolador, el recurso compartido es el CPU, y este solo puede atender una tarea a la vez. Por lo tanto, es necesario implementar cierta lógica en el código que permita tomar decisiones inteligentes sobre qué tarea accede al recurso y en qué momento, evitando conflictos y garantizando que el sistema funcione de manera ordenada.".

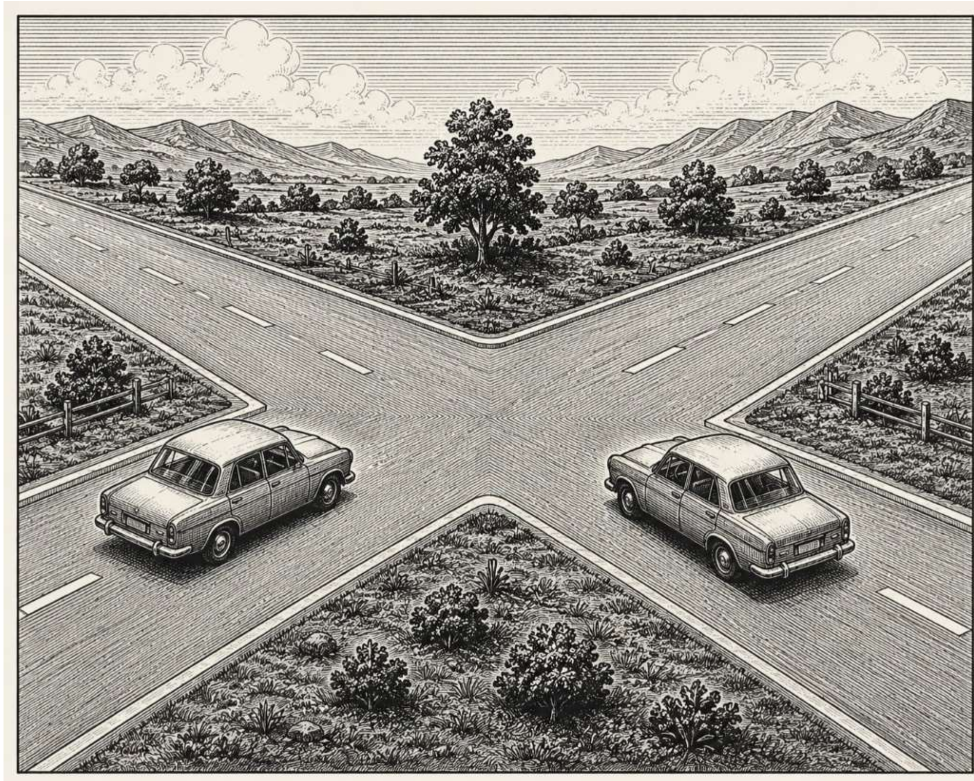


Figura 1.10. Ambos automóviles comparten un recurso compartido que es la carretera.

1.9 ¿Qué es Sincronización?

Retomando el ejemplo del cruce en la carretera: un semáforo se encarga de regular el tráfico proporcionando una regla de acceso. Básicamente,

EDICIÓN DE MUESTRA

aumenta y las decisiones se vuelven más complejas. Este es un ejemplo de **semáforo**: se pueden realizar varias acciones en función de variables condicionales.

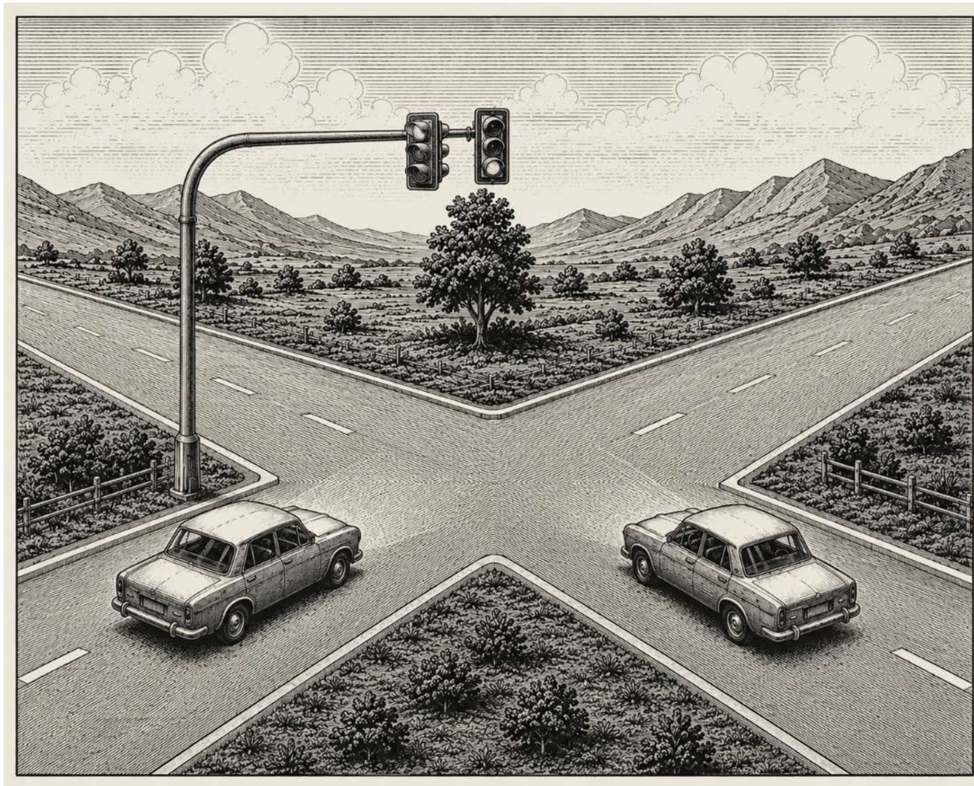


Figura 1.11. El semáforo es un ejemplo de sincronización para regular el tráfico en un cruce de carreteras.

La gestión del semáforo puede planificarse de diferentes maneras. Dos ejemplos comunes son:

Un sistema periódico, donde el acceso a la carretera se alterna entre el tráfico en dirección norte-sur y el tráfico en dirección este-oeste en intervalos de tiempo fijos.

Un sistema basado en eventos, donde la carretera cuenta con sensores que detectan los vehículos en espera y toma decisiones lógicas sobre cuándo cambiar de estado en función de los datos disponibles.

1.10 ¿Qué es Inanición?

Independientemente del tipo de planificación que se utilice, algunas condiciones deben estar claras.

En primer lugar, el acceso al cruce debe ser alternado. Si no es así, una de las direcciones permanecerá bloqueada indefinidamente, lo cual no puede

1.11 ¿Qué es Punto Muerto?

De camino al trabajo, el compañero de oficina decidió no respetar las normas de acceso e intentó apropiarse de un tramo de la carretera que en ese momento estaba siendo usado por otro vehículo. Como resultado, se

produjo una interrupción que obligó al conductor a modificar su ruta para llegar al trabajo.

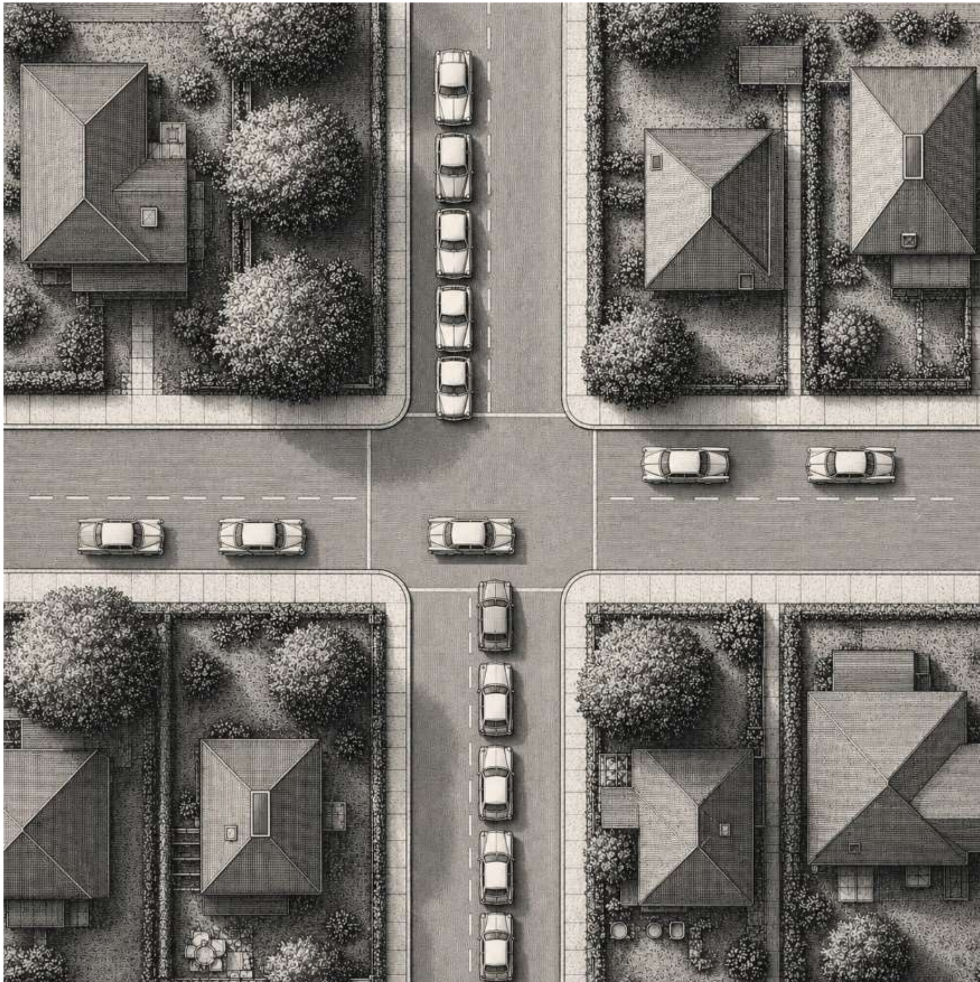


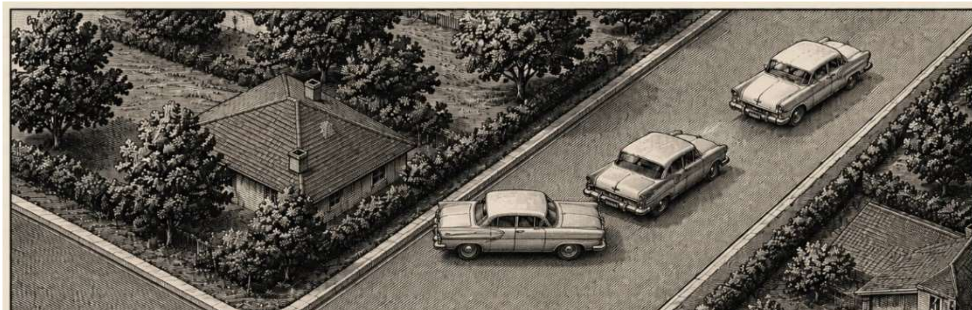
Figura 1.12. El problema de la inanición ilustrado por un semáforo mal calibrado.

Por suerte, el conductor sigue las reglas de acceso y es capaz de reaccionar con la rapidez suficiente para evitar el incidente sin mayores consecuencias. Sin embargo, debido al desvío, debe tomar una ruta diferente a la habitual. Al final llegará al trabajo, pero probablemente con algo de retraso.

El compañero, en cambio, al no haber respetado las reglas de acceso, quedó bloqueado. En el mejor de los casos, el incidente provocará un retraso breve. En el peor, puede producirse un **punto muerto** — o ***Deadlock*** en

inglés — donde la carretera queda completamente cerrada e inaccesible para todos. Figura 1.13

Por fortuna existe una ruta secundaria por la que se puede circular. Fue necesario modificar algunas variables del trayecto — las calles de conexión utilizadas para llegar al destino — lo que introdujo fluctuaciones en el tiempo total de viaje. Algo inesperado generó un tiempo de procesamiento superior al ideal, debido a la mayor latencia que el conductor tuvo que asumir.



EDICIÓN DE MUESTRA

1.12 ¿Qué es Mutex?

Es importante recordar que, en los ejemplos anteriores, los autos son los

EDICIÓN DE MUESTRA

1.13 ¿Qué es un Semáforo?

Un **semáforo** es un mecanismo de sincronización más avanzado que el **mutex**, ya que permite gestionar el acceso a recursos compartidos considerando múltiples condiciones simultáneamente.

Retomando el ejemplo del tráfico: imaginemos que el conductor debe girar a la izquierda en un cruce. En ese momento, hay un peatón cruzando por el paso peatonal frente a él, y además hay un vehículo que viene de frente con

intención de seguir recto. Según las normas de acceso, el conductor debe primero esperar a que el peatón despeje el paso, y luego ceder el paso al vehículo de frente antes de poder girar. Figura 1.15.

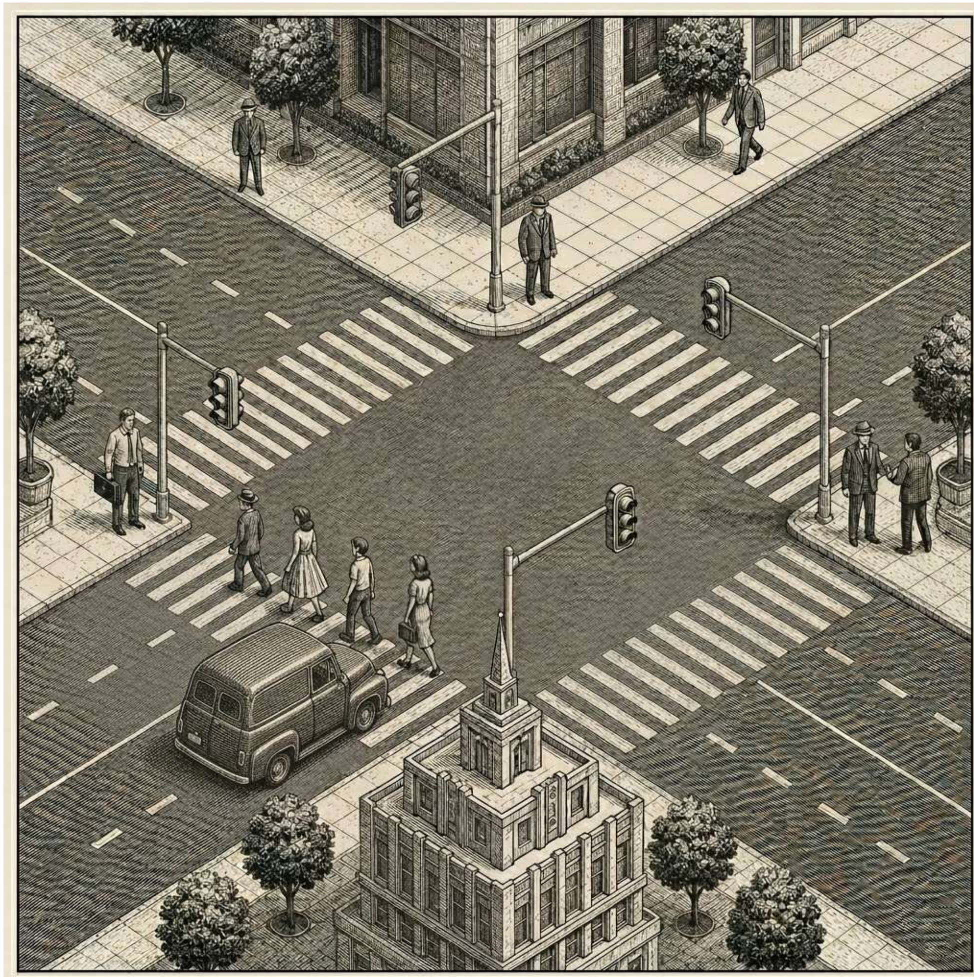


Figura 1.14. Cuando un peatón pisa la calle, todos los autos deben permitir su paso.

A diferencia del **mutex**, donde la decisión es binaria — el recurso está libre o está ocupado — el **semáforo** evalúa múltiples variables condicionales antes de tomar una decisión. En este caso: ¿hay un peatón cruzando? ¿Hay un vehículo con prioridad? ¿Ambas condiciones se han resuelto? Solo cuando todas las condiciones son favorables, el conductor puede continuar.

Figura 1.15. Un semáforo es un método de sincronización más complejo que mutex.

Estas variables pueden cambiar de estado durante el funcionamiento normal del sistema, y el semáforo se encarga de evaluar esos cambios y coordinar el acceso de manera ordenada.

1.14 ¿Qué es un proceso concurrente?

Se ha utilizado el ejemplo del trayecto al trabajo para ilustrar distintos conceptos. Es momento de reunirlos en una definición concreta.

Un proceso concurrente es una secuencia de acciones que se ejecuta de forma determinista para alcanzar un resultado esperado, incluso cuando coexiste con otros procesos que comparten los mismos recursos.

En el ejemplo, el resultado final siempre es llegar al trabajo. El tiempo de viaje puede variar debido al tráfico, un accidente o un desvío, pero el

EDICIÓN DE MUESTRA

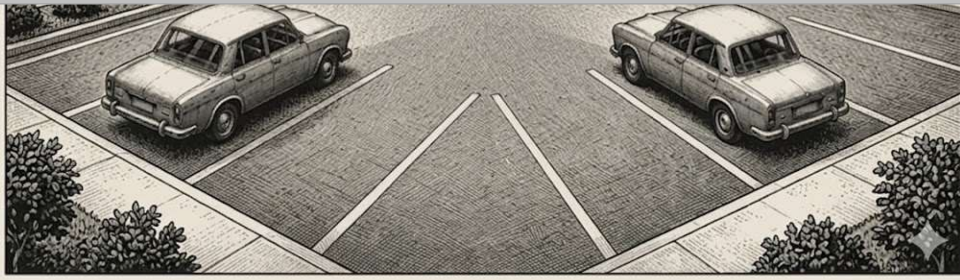


Figura 1.16. El viaje del hogar al trabajo es un proceso determinista porque produce un resultado predecible.

En este ejemplo, el recurso principal eran las carreteras y calles, compartidas por autos y peatones. Las normas de acceso — semáforos,

EDICIÓN DE MUESTRA

deadlocks, latencia excesiva y pérdida de predictibilidad

1.15 ¿Qué es un Jitter y Latencia?

Siguiendo con el ejemplo del trayecto al trabajo, supongamos que el tiempo ideal para llegar, sin ningún inconveniente en el camino, es de 20 minutos. A ese tiempo ideal se le denomina latencia: es el tiempo que transcurre desde que el proceso comienza hasta que produce su resultado.

EDICIÓN DE MUESTRA

Figura 1.17. La latencia es el tiempo que toma un proceso en completarse. La fluctuación o variación impredecible de dicho tiempo entre un evento y otro se denomina ***Jitter***.

En resumen: la latencia es el tiempo de respuesta del sistema, y el *jitter* es la variación de ese tiempo. Un sistema bien diseñado busca minimizar ambos: una latencia baja garantiza respuestas rápidas, y un **jitter** bajo garantiza que esas respuestas sean predecibles y consistentes.

1.16 ¿Qué es Concurrencia Verdadera?

Regresando al ejemplo del horno microondas, tomemos tres tareas: la que

EDICIÓN DE MUESTRA

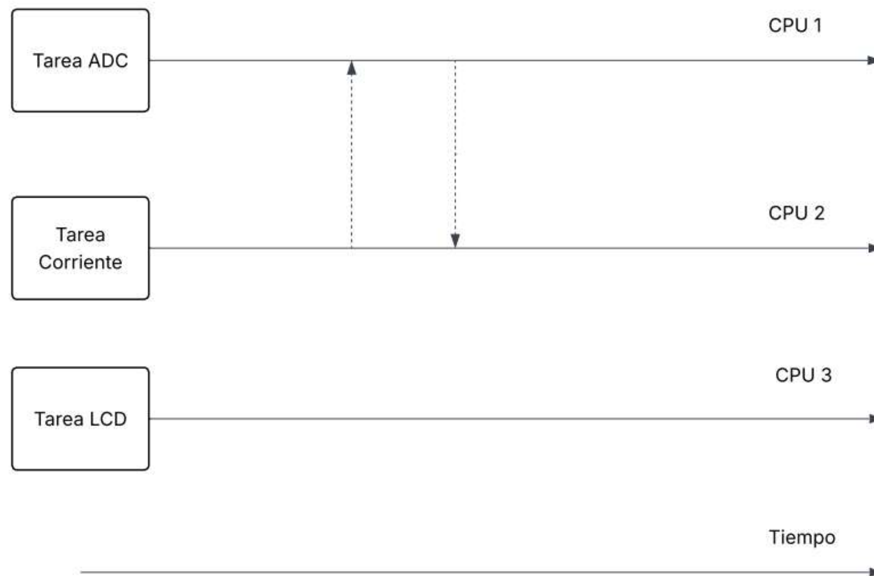


Figura 1.18. Concurrencia verdadera con 3 procesadores.

La tarea que administra la pantalla opera de manera completamente independiente. Muchas pantallas LCD solo reciben datos y nunca envían una respuesta, por lo que esta tarea no depende de ninguna otra para funcionar.

La tarea que monitorea la temperatura interna del horno, en cambio, sí depende de otra: necesita los datos que provee la tarea que lee el conversor ADC. No puede funcionar sin ellos.

La tarea del ADC opera de manera independiente — está constantemente leyendo el conversor y no hace nada más — pero es indispensable para que la tarea de monitoreo de temperatura pueda cumplir su función.

Para que estas tres tareas funcionen verdaderamente al mismo tiempo, se necesitarían tres procesadores independientes, uno para cada tarea. Eso es la concurrencia verdadera. Sin embargo, la mayoría de los

microcontroladores cuentan con un único CPU, lo que hace imposible ejecutar varias tareas en paralelo real.

1.17 ¿Qué es Concurrencia Virtual?

Para que un solo procesador se encargue de ejecutar múltiples tareas concurrentes, se asigna a cada una pequeña porción de tiempo de forma ordenada.

La clave está en que el CPU atiende cada tarea por turnos, avanzando un poco en cada una de ellas. Si una tarea depende del resultado de otra, el

EDICIÓN DE MUESTRA

y la tarea de monitoreo lo analiza nuevamente. Figura 1.19.

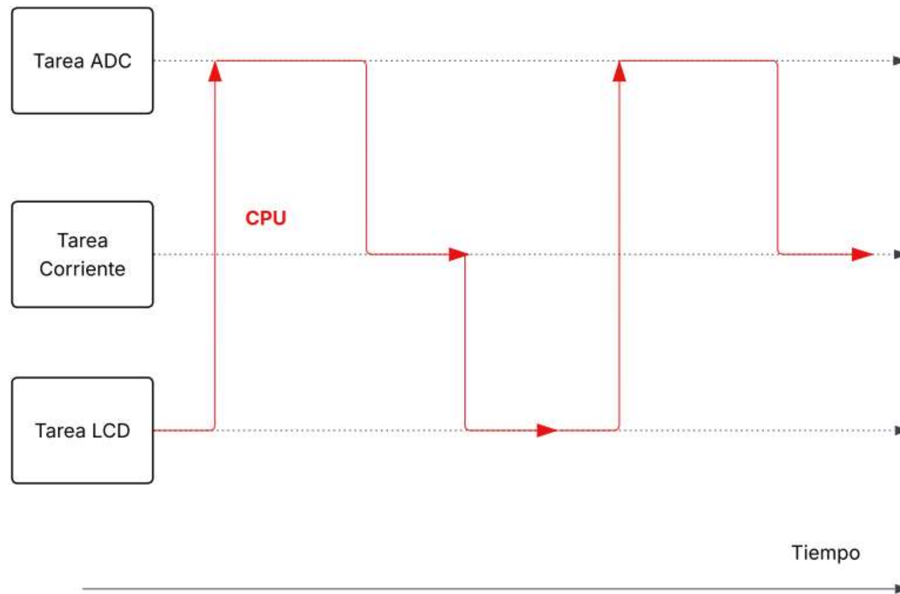


Figura 1.19. Concurrencia virtual consiste en que un único CPU haga varios procesos a la vez para crear la ilusión de simultaneidad.

1.18 ¿Qué es Tarea Cooperativa?

Este ciclo continuo crea la ilusión de que las tareas se ejecutan al mismo tiempo, cuando en realidad el CPU las atiende de forma secuencial, pero tan rápidamente que el resultado es indistinguible de la concurrencia verdadera.

A través de esta concurrencia virtual, las tareas operan de forma cooperativa: cada una cede voluntariamente el control del CPU una vez que ha completado su porción de trabajo, permitiendo que las demás tengan su turno. No hay ninguna autoridad externa que las interrumpa — la coordinación depende de que cada tarea haga su parte y libere el recurso a tiempo.

tarea se bloquea, el sistema entero se detiene con ella.

1.19 Analogía del tráfico con un sistema embebido.

A lo largo de este capítulo se ha utilizado el tráfico vehicular como hilo conductor para explicar los conceptos fundamentales de los sistemas embebidos. Es momento de hacer explícita la relación entre ambos mundos.

Los actores del sistema en el ejemplo del tráfico son los automóviles y los peatones — cada uno con sus propias reglas, prioridades y objetivos. En un sistema embebido, los actores equivalentes son las tareas, hilos, funciones o procesos que el microcontrolador debe ejecutar.

Los recursos compartidos en el tráfico son las calles y carreteras, que solo pueden ser utilizadas por un vehículo a la vez en cada tramo. En un sistema embebido, los recursos compartidos son el CPU, la memoria y los registros del microcontrolador — igualmente limitados e igualmente disputados por múltiples tareas.

Las reglas de acceso en el tráfico incluyen ceder el paso, respetar los semáforos, incorporarse con cuidado a un carril o frenar ante un cruce. En un sistema embebido, estas reglas se traducen en mecanismos de sincronización como mutex y semáforos, esquemas de prioridad y planificadores que determinan qué tarea accede al CPU y en qué momento.

El resultado determinista en el ejemplo del tráfico era siempre el mismo: llegar al trabajo. En un sistema embebido, el resultado esperado es que cada tarea complete su función correctamente y dentro de su deadline, contribuyendo al funcionamiento coherente del sistema completo.

La analogía no es perfecta — ninguna lo es — pero ilustra con claridad que los problemas de concurrencia, sincronización y gestión de recursos no son exclusivos del software. Son problemas universales que aparecen en cualquier sistema donde múltiples actores compiten por recursos limitados

1.20 ¿Qué es Condición de Carrera?

Uno de los problemas más importantes y difíciles de detectar en un sistema concurrente se denomina **condición de carrera**, o *Race Condition* en inglés.

Se define como el comportamiento inesperado de un sistema cuya salida depende del orden o del momento exacto en que ocurren ciertos eventos, los cuales no siempre pueden controlarse ni predecirse.

Retomando el ejemplo del tráfico: imaginemos una vía de dos carriles que se estrecha en un solo carril. Dos vehículos que circulan en paralelo se aproximan simultáneamente a ese punto de reducción. Ninguno cede el paso al otro y ambos ingresan al carril único al mismo tiempo. El resultado es que ninguno puede avanzar y todo el tráfico detrás de ellos queda bloqueado. Nadie planeó ese resultado — ocurrió porque dos procesos intentaron acceder al mismo recurso limitado en el mismo momento, sin ninguna regla que determinara quién tenía prioridad. Figura 1.21

En un sistema embebido ocurre algo similar. Si dos tareas acceden o modifican el mismo recurso compartido — una variable, un registro, un periférico — sin ningún mecanismo de sincronización, el resultado final dependerá de cuál de las dos lo hizo primero. Y ese orden puede variar cada vez que el sistema se ejecuta, produciendo resultados distintos e impredecibles.

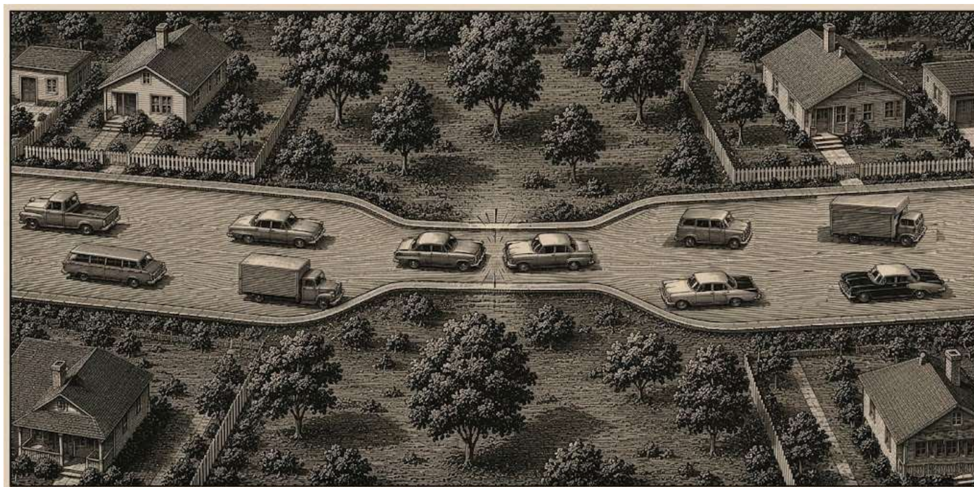


Figura 1.21. Condición de carrera: dos procesos intentan usar el mismo recurso al mismo tiempo, provocando un bloqueo impredecible.

El problema más grave de las condiciones de carrera es que el sistema puede funcionar correctamente la mayor parte del tiempo, y fallar solo en circunstancias muy específicas que son difíciles de reproducir. Esto las convierte en uno de los errores más complicados de identificar y corregir en el desarrollo de firmware.

La manera de evitarlas es garantizar que el acceso a los recursos compartidos esté controlado mediante los mecanismos adecuados de sincronización, como los que se explicaron en los subcapítulos anteriores.

1.21 ¿Qué es un *Heinsenbug*?

Las condiciones de carrera son difíciles de reproducir y, por lo tanto, difíciles de encontrar y corregir. Este comportamiento esquivo tiene un nombre propio en el mundo del desarrollo de software: *Heisenbug*.

EDICIÓN DE MUESTRA

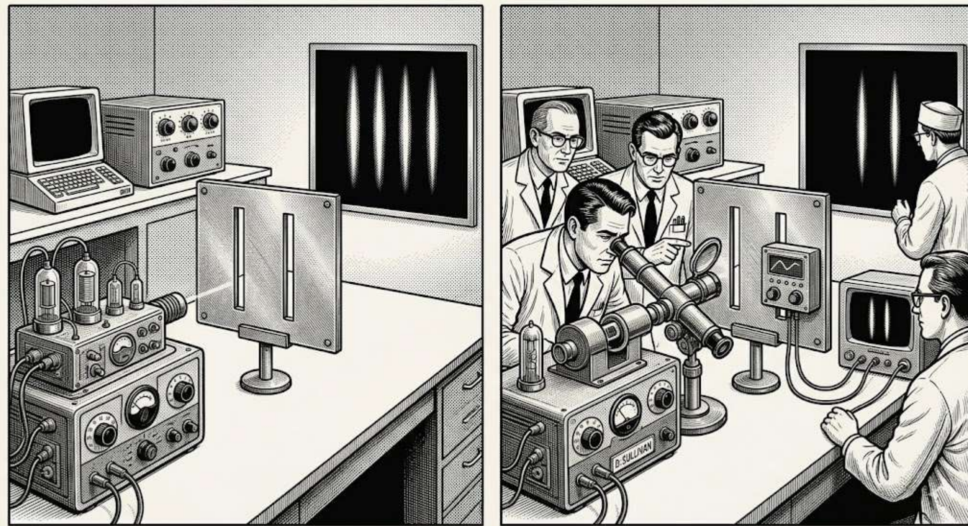


Figura 1.22. El sistema embebido se muestra inestable cuando no está observado (izquierda), pero el error desaparece al conectar las herramientas de depuración (derecha).

En realidad, el error sigue ahí — simplemente las condiciones exactas que lo provocan ya no se están dando. En cuanto el sistema vuelve a ejecutarse sin el depurador, el problema regresa.

EDICIÓN DE MUESTRA

3 unidades del mismo producto en la misma página. Figura 1.23

Ambos realizan la compra simultáneamente. El sistema procesa su pedido primero, por una diferencia de apenas unos milisegundos. Usted ve en pantalla que su compra fue exitosa y que el pedido será despachado — todo parece correcto.

El otro comprador, sin embargo, queda confundido: se realizó el cobro por 3 unidades, pero la página le indica que el producto está agotado. ¿Qué ocurrió con su dinero?

Y quien administra el sistema se encuentra con algo aún más extraño: el inventario muestra menos 3 unidades — un valor negativo que no tiene ningún sentido.

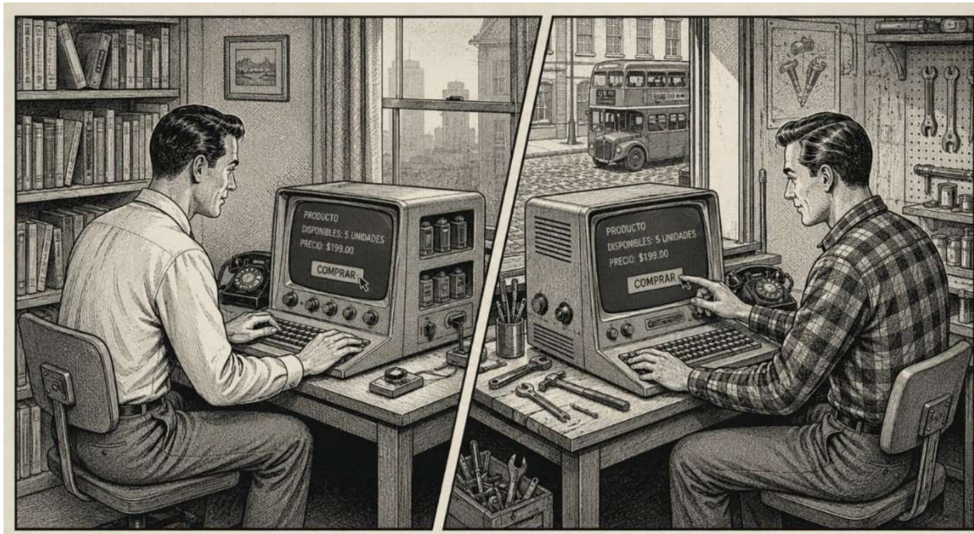


Figura 1.23. Dos usuarios leen simultáneamente el mismo dato de inventario (5 unidades) en paralelo. Ambos procesarán la compra al mismo instante, corrompiendo el stock final del sistema

Lo que ocurrió es que ambas solicitudes leyeron el inventario al mismo tiempo, cuando aún había 5 unidades disponibles. Cada una tomó esa información como válida y procedió con la compra sin saber que la otra estaba haciendo exactamente lo mismo en paralelo. El sistema no tenía

ningún mecanismo que impidiera el acceso simultáneo a ese recurso compartido.

EDICIÓN DE MUESTRA

genera una interrupción que incrementa la variable auxiliar. En cualquier momento, al combinar ambos valores se obtiene un temporizador virtual de 32 bits. Aparentemente es una solución elegante y correcta. Figura 1.24

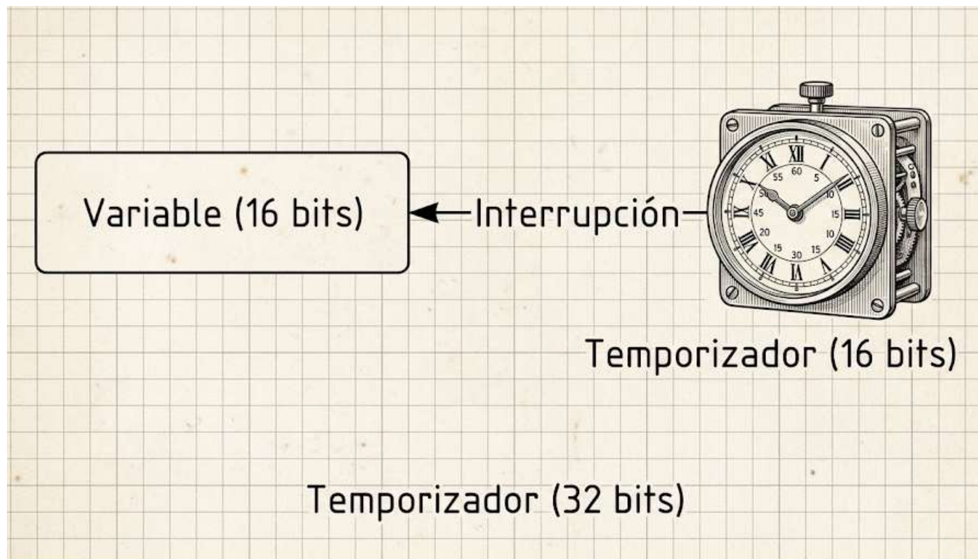


Figura 1.24. Con un temporizador de 16 bits y una variable también de 16 bits, se puede conseguir un “temporizador” de 32 bits.

Sin embargo, aquí se esconde una condición de carrera.

Supongamos que el temporizador está a punto de desbordarse, es decir, su valor actual es 0xFFFF, y la variable auxiliar vale 0x0003. Se invoca una función que debe leer ambos valores y combinarlos para formar la variable de 32 bits. Figura 1.25.

```
uint32_t get32BitsTimerValue (void)
{
    return ((uint32_t)variable16bits << 16) | (uint32_t)timer16BitsValue;
}
```

Figura 1.25. Función que permite unir la variable y el temporizador de 16 bits para crear una variable de 32 bits

Caso 1: se lee primero la variable auxiliar. Su valor 0x0003 es copiado a registros internos del CPU, desplazada 16 bits y convertida a una variable de 32 bits, operaciones que consumen algunos ciclos de máquina. Durante esos ciclos, el temporizador se desborda, su valor pasa a 0x0000. Cuando a continuación se lee el temporizador, su valor ya es 0x0000. El resultado combinado es 0x00030000, lo cual es incorrecto. Figura 1.26

Caso 2: se lee primero el temporizador y luego la variable auxiliar, Figura 1.27. Su valor 0xFFFF es copiado a registros internos. Durante esa operación, el temporizador se desborda y la interrupción incrementa la variable auxiliar de 0x0003 a 0x0004. Cuando se lee la variable auxiliar, su valor ya es 0x0004. El resultado combinado es 0x0004FFFF, también incorrecto. Figura 1.28.

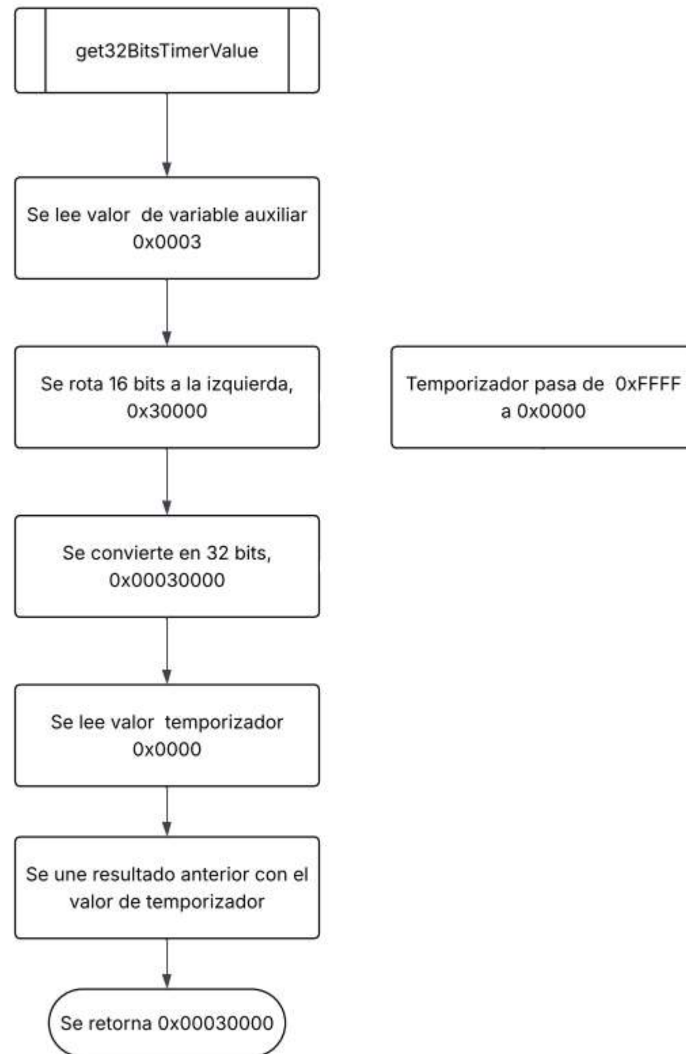


Figura 1.26. Primero se lee la variable auxiliar y luego al temporizador, pero a la final se obtiene un valor erróneo.

```
uint32_t get32BitsTimerValue (void)
{
    return (uint32_t)timer16BitsValue | (uint32_t)(variable16bits << 16));
}
```

Figura 1.27. Función que permite unir la variable y el temporizador de 16 bits para crear una variable de 32 bits, primero se lee al temporizador.

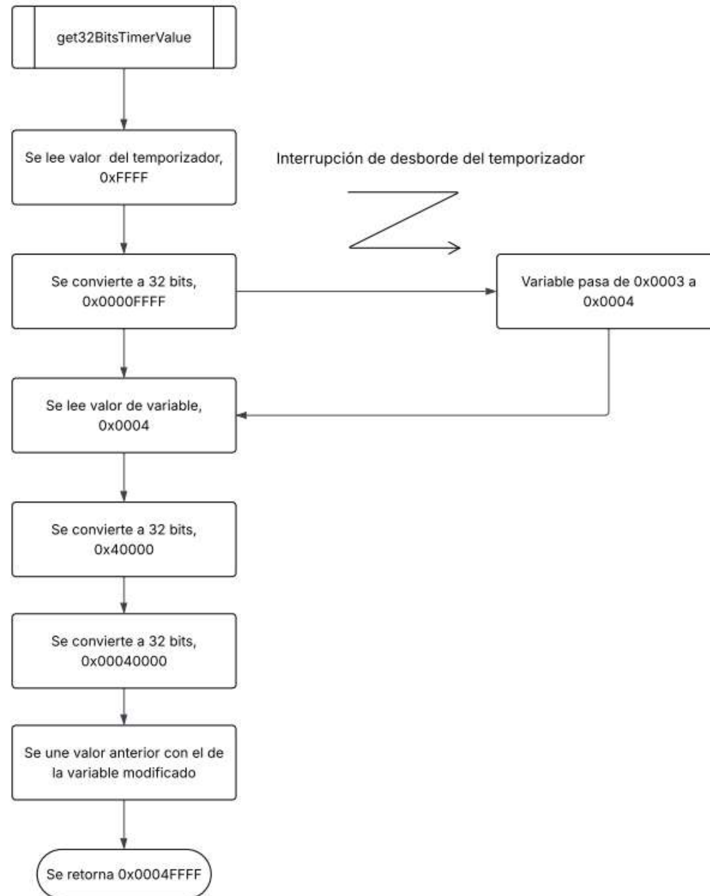


Figura 1.28. Primero se lee el temporizador y luego la variable auxiliar, pero a la final también se obtiene un valor erróneo.

En ambos casos el resultado es erróneo. El valor correcto debería haber sido 0x0003FFFF — si el desbordamiento aún no había ocurrido — o 0x00040000 si ya había ocurrido. Pero la condición de carrera entre la lectura de los dos valores y la interrupción de desbordamiento produce un resultado corrupto que puede ser muy difícil de detectar, precisamente porque no ocurre siempre, solo cuando la lectura coincide con el momento exacto del desbordamiento.

1.22 ¿Qué es una Operación Atómica?

En el ejemplo del temporizador de 32 bits, una solución al problema de la

ningún momento.

El flujo típico es el siguiente: se invoca la función o proceso que requiere protección, se deshabilitan todas las interrupciones del sistema, se ejecuta el proceso hasta completarse, y finalmente se vuelven a habilitar las interrupciones, devolviendo el sistema a su funcionamiento normal.

Sin embargo, las operaciones atómicas tienen un costo: mientras el sistema está bloqueado, ninguna otra tarea puede ejecutarse ni ninguna interrupción puede atenderse. Por esa razón deben ser lo más breves

posible y usarse solo cuando sean estrictamente necesarias. Abusar de ellas es contrario al objetivo de crear verdadera multitarea cooperativa. Figura 1.30

Figura 1.30. Al igual que una operación atómica, la balsa solo permite que un automóvil use el recurso a la vez, evitando interferencias y accesos simultáneos

Las operaciones atómicas verdaderas no toleran fluctuaciones de tiempo — su duración debe ser predecible y constante, ya que cualquier variación puede comprometer el funcionamiento del sistema.

Para la mayoría de los casos donde se necesita proteger el acceso a un recurso compartido, existe una alternativa menos invasiva: el bloqueo mutex, que se explica a continuación

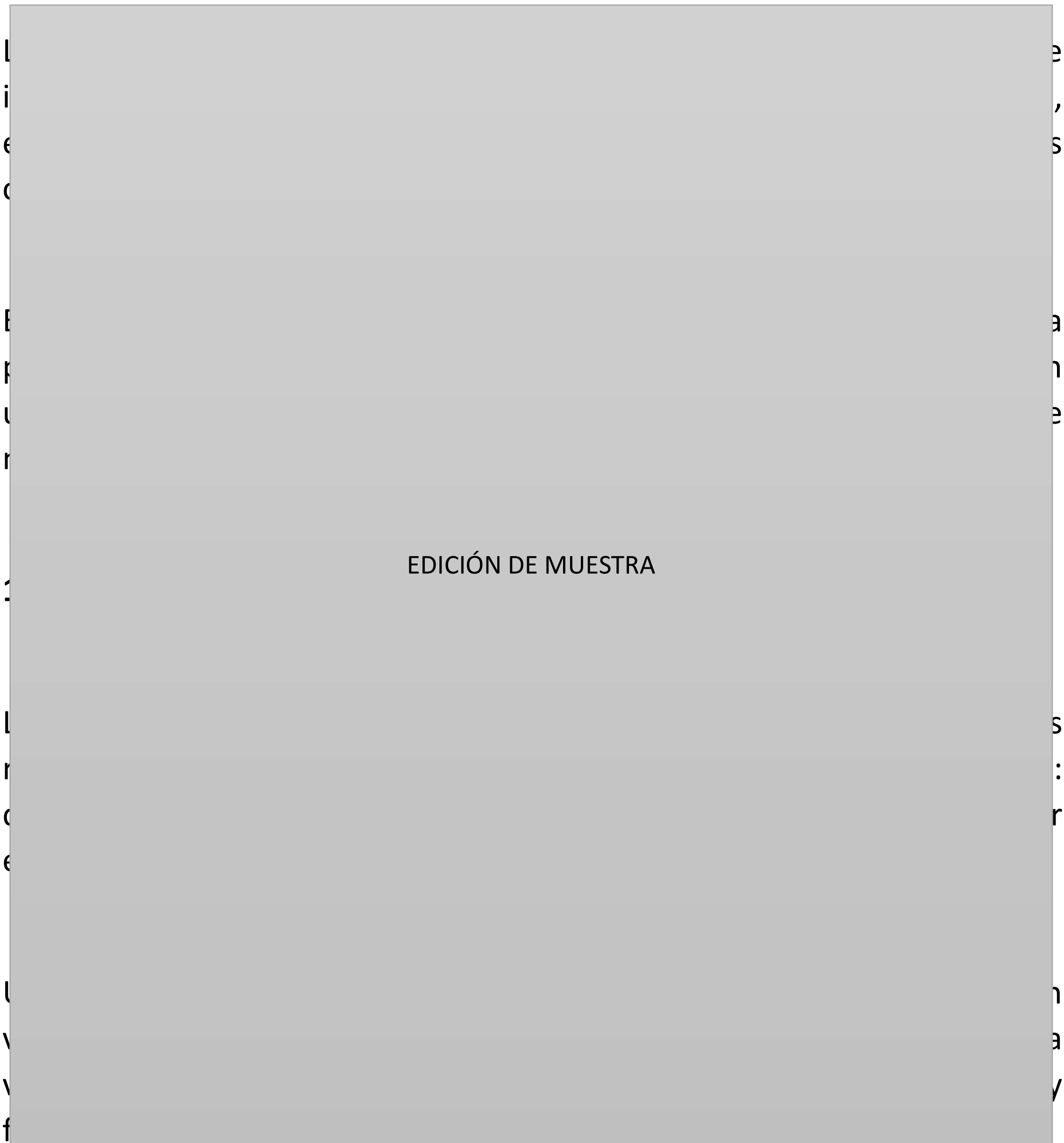
1.23 ¿Qué es un Bloqueo Mutex?

Un bloqueo mutex es una variable que garantiza que solo una tarea pueda acceder a un recurso compartido en un momento dado. A diferencia de la operación atómica, no deshabilita las interrupciones del sistema — simplemente señaliza que el recurso está en uso, para que otras tareas lo sepan y esperen su turno. Figura 1.31

Figura 1.31. El estacionamiento es un recurso compartido protegido por un mutex. Cuando no hay plazas disponibles, los vehículos no pueden entrar y se retiran, no pueden quedarse creando tráfico.

En un microcontrolador, un ejemplo claro es el conversor ADC. Un dispositivo puede tener múltiples sensores conectados, pero generalmente solo hay un único ADC disponible. Si dos tareas intentan usarlo simultáneamente sin ningún mecanismo de control, una de ellas podría leer el valor correspondiente al sensor de la otra, produciendo un error difícil de detectar.

Con un bloqueo mutex, antes de usar el ADC la tarea verifica si el recurso está libre. Si lo está, lo marca como ocupado, realiza su lectura y al finalizar lo libera. Si no está libre, espera hasta que la otra tarea lo libere.



Si ambas lecturas del registro alto son idénticas, significa que no hubo desbordamiento durante la lectura y el valor combinado es válido. Si difieren, significa que el temporizador se desbordó entre las dos lecturas y el proceso debe repetirse. Figura 1.32.

```
uint32_t get32BitsTimerValue (void)
{
    do
    {
        uint16_t highPart = variable16BitsValue;
        uint16_t lowPart = timer16BitsValue;
    } while (highPart != variable16BitsValue);  //Si la parte alta ha cambiado, repito el proceso
    return ((uint32_t)highPart << 16) | (uint32_t)highLow;
}
```

Figura 1.32. Verificar que la variable auxiliar no haya cambiado tras leer el temporizador permite evitar la condición de carrera sin detenerlo.

Esta técnica requiere más operaciones y comparaciones que una operación atómica simple, lo que la hace levemente más costosa en ciclos de CPU. Sin embargo, no bloquea el sistema ni deshabilita interrupciones, lo que la hace más compatible con un modelo de multitarea cooperativa. Figura 1.33

1.25 ¿Qué es el método Productor-Consumidor?

El método Productor-Consumidor es un patrón de sincronización que

carrera.

Figura 1.33. El automóvil se aparta a la bahía de acceso para verificar disponibilidad sin interrumpir el tráfico, igual que una lectura atómica con verificación evita detener el proceso.

Esta dependencia aumenta la complejidad del sistema y debe tenerse en cuenta desde el diseño. Identificar correctamente qué tareas producen datos y cuáles los consumen, y en qué orden deben ejecutarse, es fundamental para planificar un sistema cooperativo que funcione de manera predecible y sin errores. Figura 1.34.

1.26 Principios fundamentales de la concurrencia

Al diseñar un sistema que opera múltiples procesos y tareas concurrentes, hay tres conceptos fundamentales que deben cumplirse: desempeño, eficiencia y escalabilidad.

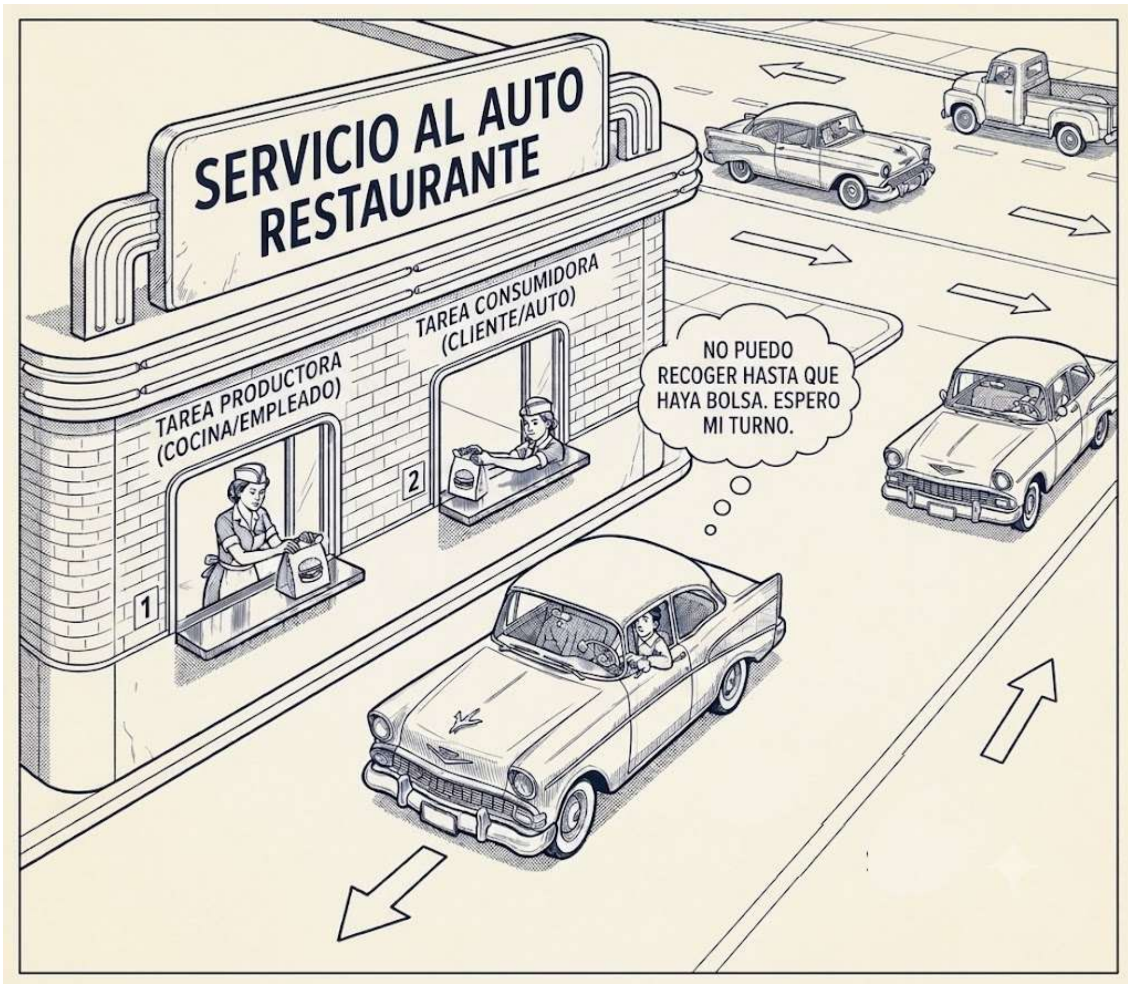


Figura 1.34. El auto servicio de comida es un claro ejemplo de método Productor-Consumidor.

1.26.1 Desempeño.

Un buen desempeño se basa en tres elementos relacionados: el rendimiento, la latencia y el jitter.

El *rendimiento*, o ***throughput***, es la duración de la ejecución de una tarea desde que inicia hasta que finaliza. Puede medirse en tiempo o en ciclos de reloj del CPU. El objetivo es que ese tiempo sea lo más pequeño posible,

asegurando que cuando una tarea se ejecuta, no desperdicie tiempo esperando. Si una tarea necesita datos que aún no están disponibles, el CPU debería atender otra tarea y regresar cuando los datos estén listos.

EDICIÓN DE MUESTRA

latencia y el jitter, manteniéndose estable incluso cuando las condiciones no son las ideales.

1.26.3 Escalabilidad.

La escalabilidad es la capacidad del sistema para adaptarse a cambios — ya sea en complejidad, en número de tareas o en requisitos — sin que eso afecte negativamente su funcionamiento.

Una tarea es escalable si las modificaciones realizadas en una tarea relacionada no afectan su capacidad de procesamiento. Por ejemplo, si soy una tarea consumidora y la tarea productora fue modificada o ampliada, esos cambios no deberían afectar mi tiempo de procesamiento ni mi funcionamiento. El impacto debe limitarse al rendimiento interno del productor.

EDICIÓN DE MUESTRA

Escalado horizontal (**Scale out**) Consiste en diseñar una tarea o proceso con un conjunto de reglas más amplio, capaz de incorporar una variedad de funciones. Es más flexible y adaptable, pero requiere más recursos para funcionar correctamente.

EDICIÓN DE MUESTRA