

Módulo 2

Introducción y conceptos básicos

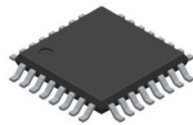
¿Qué es un sistema embebido?

- Sistema de computación con un fin específico.
- Forma parte de un sistema completo con otras partes físicas y mecánicas.
- Reacciona al entorno físico.



¿Qué es un sistema embebido?

- Generalmente poseen un microcontrolador.
- Memoria y capacidad de procesamiento limitada.
- En un sistema embebido se busca:
 - Reducir su tamaño, su costo, y su consumo.
 - Optimizar la performance y confiabilidad.
 - Asegurar un tiempo de respuesta determinado.



Microcontroladores

- B / KB de memoria

Microchip (PIC)

Atmel (AVR, ATmega)

Espressif ESP32



Microprocesadores / SOM

- MB / GB de memoria

ARM

RISC-V

ASICs

Características

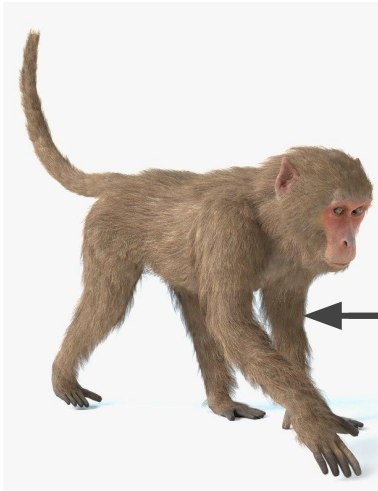
I/O *

Pueden o no tener:

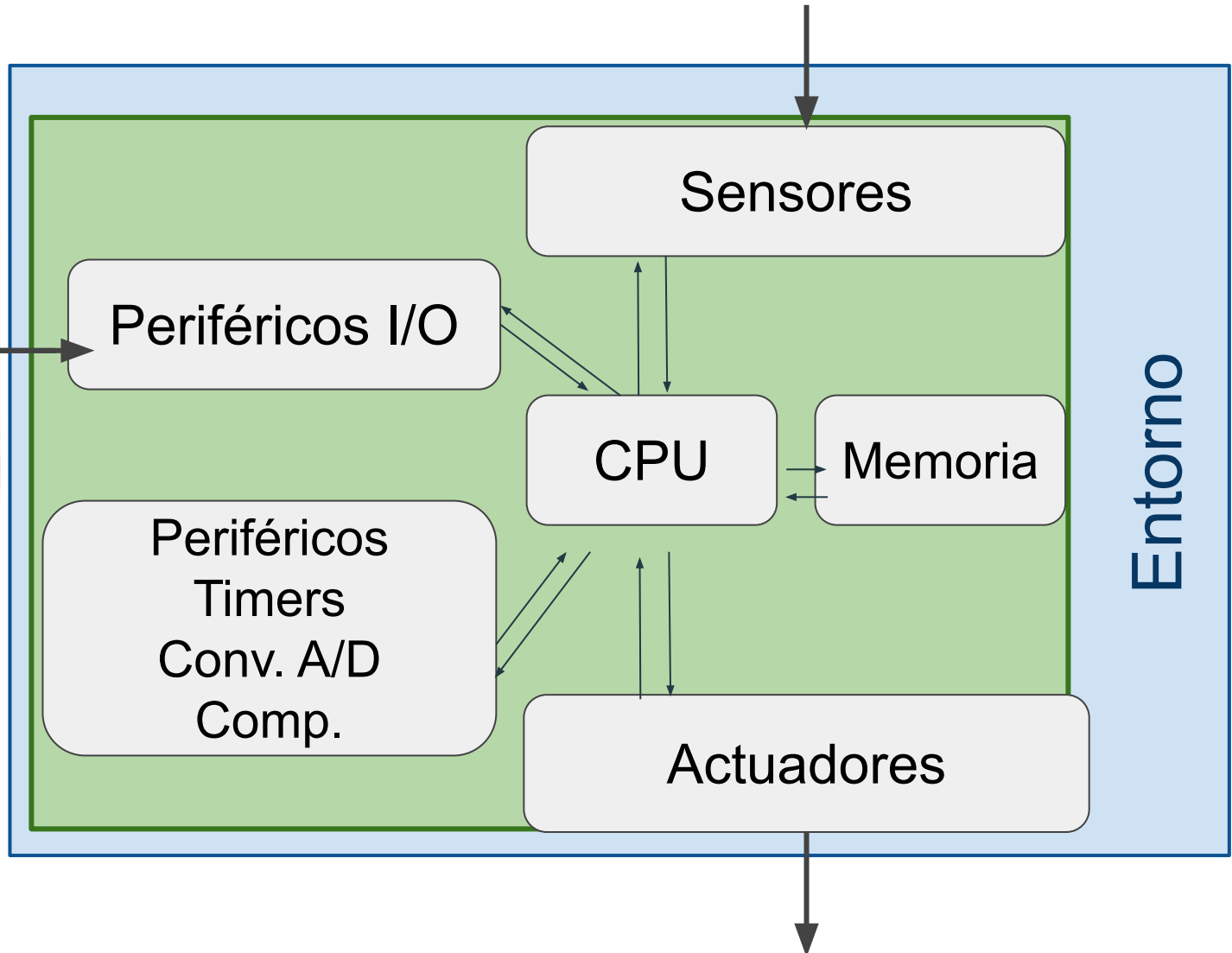
- Botones o teclados pequeños (keypads)
- Displays (TFT, LCD, OLED, E-INK, etc.)
- Conectividad (RS232, USB, Ethernet, WiFi, Bluetooth, etc.)
- Memoria (RAM, EEPROM, FLASH, FRAM, etc.)
- Sonido
- Vibración
- Leds

*Excepto capaz la memoria, ponele que ahí depende

Hardware



Usuario



Criterios de diseño

- Rendimiento
- Confiabilidad
 - No se puede resetear
 - Pérdida de información importante
 - Riesgo de vida
 - No hay un usuario manipulando
- Disponibilidad
 - 24 - 7 - 365
- Seguridad
- Bajo consumo

Lenguajes de programación

- C es el lenguaje por excelencia para SSEE
 - Compromiso entre facilidad para entender y mantener (comparado con Assembly) y eficiencia (comparado con lenguajes de más alto nivel)
- También se usan muchos otros lenguajes, dependiendo el caso, por ejemplo C++, Python, Java, Lua, Rust, entre otros.
- Los lenguajes de más alto nivel se suelen usar en micros más potentes en combinación con algún sistema operativo.

¿Cómo se programan?

- Bare Metal
- RTOS – Real Time Operating System
- Sistema Operativo Embebido

La decisión depende de varios factores, como la potencia del micro o la complejidad y necesidad de control del proyecto

Bare Metal

- No hay sistema operativo, ni ninguna capa de abstracción.
- El único código que corre es el que se programa.
- Positivo: no existe **overhead** y el manejo de recursos se hace de forma más eficiente.
- Negativo: se torna muy complejo para proyectos grandes, con varias tareas, que necesitan escalabilidad.
- Lenguajes de programación: Assembly, C

NOTA: OVERHEAD/SOBRECOSTE es el exceso de tiempo de computación, memoria, ancho de banda u otros recursos, que son necesarios para realizar una tarea específica. Este puede ser un factor decisivo en el diseño de un software en lo que se refiere a su estructura, corrección de errores e inclusión de nuevas características. Algunos ejemplos incluyen la llamada a una función, la transferencia de información o el procesamiento de las estructuras de datos.

RTOS - Real Time Operating System

- Funcionalidades core de un sistema operativo – scheduler, manejo de memoria, sincronización y comunicación entre tareas (mutex, semáforos, queues...)
- Clave: tiempo real
- Lenguajes de programación: generalmente C o C++
- Ejemplos: FreeRTOS, Zephyr, embOS

Sistema Operativo Embebido

- Versión más compacta y eficiente (recortada) de un sistema operativo tradicional
- Mayor cantidad de servicios en comparación con opciones anteriores
- Lenguajes de programación: Cualquiera...
- Ejemplos: OpenWRT, Ubuntu Core.
- Generadores de imágenes: Yocto, Buildroot

Firmware vs Software Embebido

- Frontera difusa
- Firmware es un tipo de software
 - Asociado históricamente a software grabado en ROM durante fabricación
 - El software que hace que el hardware funcione
 - Puede ser una parte (o todo) el software en un sistema embebido
 - No necesariamente presente en un sistema embebido.
Por ejemplo: BIOS
- Software embebido es simplemente software que corre sobre un sistema embebido

Arquitecturas de software

Algunas definiciones:

- Evento: situación que debe ser atendida (ej: cambio en una entrada, recepción de un dato, etc.)
- Handler: código que se encarga de gestionar el evento

En términos generales, la arquitectura define cómo se detectan los eventos y de qué forma se llama al handler.

Arquitecturas de software

- De donde surgen?

A medida que **aumenta el tamaño y la complejidad** de los sistemas de software, el problema de diseño **va más allá de los algoritmos y estructuras** de datos de la computación: **diseñar y especificar la estructura general del sistema surge como un nuevo tipo de problema.**

Las cuestiones estructurales incluyen la organización general y la estructura de control global; protocolos de comunicación, sincronización y acceso a datos; asignación de funcionalidad para diseñar elementos; distribución física; composición del diseño elementos; escalamiento y rendimiento; y selección entre alternativas de diseño

Este es el nivel de diseño de la arquitectura de software.

“An Introduction to Software Architecture”
David Garlan and Mary Shaw

La **arquitectura del software** juega un papel **esencial al iniciar** cualquier proyecto de desarrollo de software. Esta fase nos brinda **una guía detallada sobre cómo estructurar el producto**, identificando sus componentes, su organización y la manera en que interactúan entre sí. Es como un mapa que **nos orienta** en el camino hacia la creación del software, estableciendo las bases para su construcción.

Arquitecturas de software

¿Cómo seleccionar una arquitectura de software?

- ¿Qué tiene que hacer el sistema?
- Tiempos de respuesta.
- ¿Cuántos eventos diferentes hay que atender?
- Tiempos críticos.

Estas son con las que vamos a trabajar:

- Round Robin (RR)
- RR con interrupciones
- Planificación por encolado de funciones
- Máquinas de estado
- Sistemas operativos en tiempo real

Round Robin (RR)

1. Se utiliza un bucle infinito
2. Se chequean los eventos
3. A partir del chequeo se ejecuta el control

Prioridades:

- Ninguna: las acciones son todas iguales
- Cada handler debe esperar su turno.
- Tiempo de respuesta (peor caso):
 - Una vuelta al bucle (gestionando todos los eventos restantes primero)
 - Malo para todos los eventos si uno solo requiere procesamiento pesado

Desventajas:

- Solución frágil: agregar un nuevo handler modifica tiempos de respuesta restantes y puede provocar pérdida de deadlines.

Ventajas:

- Sencillez: única tarea, sin interrupciones.

Round Robin (RR): Ejemplo

```
void main(void)
{
    ...
    while(1)
    {
        if(evento_1)
        {
            control_1();
        }

        if(evento_2)
        {
            control_2();
        }
        ...

        if(evento_N)
        {
            control_N();
        }
    }
}
```

1. Se utiliza un bucle infinito
2. Se chequean los eventos
3. A partir del chequeo se ejecuta el control

RR con interrupciones

¿Qué es una interrupción?

Evento en el que se interrumpe el hilo de ejecución, para ejecutar *otra parte del código*.

Conceptos asociados:

- **IRQ** - *Interrupt Request*.
Señal que le indica al microprocesador que tiene que interrumpir su ejecución. Puede ser interna o externa al microcontrolador.
- **ISR** - *interrupt service routine*.
Rutina de Atención a la Interrupción.

Interrupciones

¿Qué ocurre cuando hay un interrupción?

1. Se corta el hilo de ejecución
2. ¿Puede una interrupción cortar una instrucción? **SI**
3. El micro tiene que guardar próxima instrucción a ejecutar en Program Counter (PC) y cargar el inicio de la ISR en el PC.
4. Se ejecuta la ISR, se “limpia” el flag de la interrupción, y se retorna
5. Se carga el valor guardado del PC y se sigue ejecutando el hilo principal.

¿Qué pasó con los registros del CPU y variables?

Se tiene que guardar y restaurar el contexto.

Round Robin (RR) con Interrupciones

1. Se utiliza un bucle infinito
2. Se chequean los eventos a partir de flags (banderas)
3. A partir del chequeo se ejecuta el control.

Prioridades:

- Prioridad de las interrupciones.
- Todos los handlers tienen igual prioridad

Características:

- Al agregar interrupciones:
 - ISR realiza la respuesta inicial
 - Resto realizado por funciones llamadas en el bucle
 - ISR indica con banderas la necesidad de procesamiento
- Presenta mayor flexibilidad:
 - Respuesta de tiempo crítico disparadas por interrupciones y realizadas en ISR.
 - Código con tiempo de procesamiento “largo” ubicado en handlers.
- Problema de datos compartidos:
 - Surge en la comunicación entre la interrupción y otras partes del código.
 - Generalmente una o más variables que se comparte.

Desventajas:

- ISRs y handlers comparten datos
- Tiempo de respuesta de handler no estable cuando el código cambia.

Ventajas:

- Trabajo realizado en ISR tiene mayor prioridad.
- Tiempo de respuesta de ISR estable si el código cambia.

Tiempo de respuesta:

- ISR: tiempo de ejecución de las ISR de mayor prioridad.
- Handler: suma de tiempos del resto de los handlers + interrupc.

Ejercicios RR y RR con Int

1. Round Robin:

Implemente una sistema de round robin que imprima en consola la hora configurada del dispositivo (no tiene porque estar en hora) y que identifique en consola si se esta presionando algun boton de la placa de audio.

2. Interrupciones:

Agregue al código anterior una interrupción. si se presiona alguno de los botones deberá imprimir en pantalla que ocurrió una interrupción.

3. Round robin con interrupciones:

Implemente un sistema de round robin con interrupción. El dispositivo deberá desplegar en consola un temporizador:

- Si hay tiempo cargado, debe mostrar en pantalla el tiempo restante.
- Si se mantiene presionado otro botón **PLAY** durante 3 segundos, deberá reiniciar el timer.
- Cada vez que se presiona **VOL+** deberá generar una interrupción e incrementar el timer.

Recuerde revisar los esquemáticos de la placa base y de la placa de audio. Cómo se conectan los botones? Es importante verificar que no existe problemas (o advertencias) al utilizar la placa base con la de audio.

Encolado de funciones

1. Se utiliza un bucle infinito
2. Se chequea en una cola/Queue si hay funciones a ejecutarse
3. Las interrupciones agregan las funciones a ejecutarse

Características:

- Similar a Round Robin con interrupciones
 - Trabajo dividido en ISR y handlers
 - ISR encola su propio handler
- Características de la cola de funciones:
 - FIFO: First Input First Output (cola clásica)
 - Alternativamente, un ISR de mayor prioridad podría encolar al principio para tener prioridad.
- Las interrupciones agregan **punteros a funciones** a una cola.
- El bucle principal va ejecutando esas funciones “apuntadas” a medida que van apareciendo en la cola

Encolado de funciones

1. Se utiliza un bucle infinito
2. Se chequea en una cola/Queue si hay funciones a ejecutarse
3. Las interrupciones agregan las funciones a ejecutarse

```
// Definición de tipo para puntero a función que toma y retorna un int
typedef int (*func_ptr)(int);

// Nodo de lista enlazada que almacena un puntero a función
typedef struct Node {
    func_ptr function;
    struct Node *next;
} Node;
```

```
int main() {
    // Cola de donde se guardan y levantan funciones
    Node *funcList = NULL;
    // Creo un puntero a una funcion
    int (*function)(int);

    int tries = 0;
    int resultado = 3;
    printf("Resultado: %d\n", resultado);

    // Agrego funciones al azar / simulo interrupciones
    appendFunc(&funcList, cuadrado);
    srand(time(NULL));
    for (int index = 0; index < (MAX_TRIES-1); index++)
    {
        switch(rand()%3) ...
    }

    // Verifico si hay algo para ejecutar
    // Esta limitada la cantidad de iteraciones por cuestiones
    // de compilacion, pero deberia ser 'while(1)'
    while(tries < MAX_TRIES)
    {
        if ( ( function = getFunction(funcList) ) != NULL )
        {
            printf("Pre-Resultado: %d |", resultado);
            resultado = function(resultado);
            printf("Resultado: %d\n", resultado);
        }
        nextInList(&funcList);
        tries++;
    }
    printf("Resultado: %d\n", resultado);

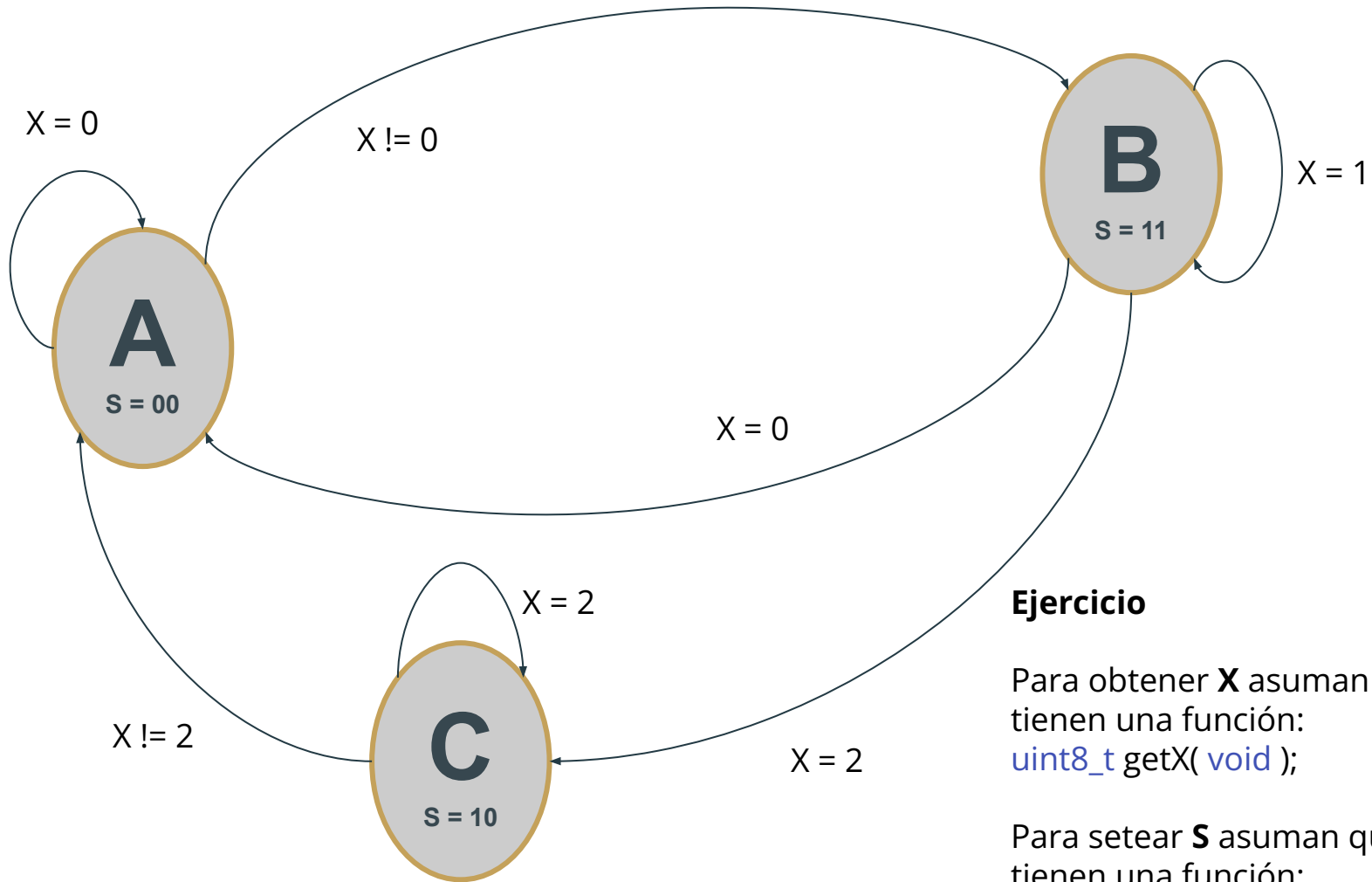
    return 0;
}
```

Multitasking

No bloqueante

- “Multitarea”: Aprovechar los “tiempos muertos” de las tareas
 - Cooperativo
 - Preemptivo
- Cooperativo: “Ceder el procesador”
- Preemptivo: “Tiempos de ejecución determinados por un scheduler” - “Slices”

Máquinas de estado



Ejercicio

Para obtener **X** asuman que tienen una función:
`uint8_t getX(void);`

Para setear **S** asuman que tienen una función:
`void setS(uint8_t p_output);`

Maquinas de estado

- Las máquinas de estado pueden usarse en conjunto con cualquier arquitectura de software, y permiten dividir una función para procesar en partes más chicas, o combinar con la arquitectura para esperar sin bloquear.
- Sirve para implementar un sistema de multitasking cooperativo.

```
#include <stdio.h>
#include <unistd.h> // para la función sleep()

// Definir los estados posibles del semáforo
typedef enum {
    ROJO,
    AMARILLO,
    VERDE
} EstadoSemaforo;

// Función para imprimir el estado actual del semáforo
void mostrarEstado(EstadoSemaforo estado) {
    switch (estado) {
        case ROJO:
            printf("Semaforo ROJO\n");
            break;
        case AMARILLO:
            printf("Semaforo AMARILLO\n");
            break;
        case VERDE:
            printf("Semaforo VERDE\n");
            break;
    }
}
```

```
int main() {
    // Estado inicial del semáforo
    EstadoSemaforo estado = ROJO;

    while (1) { // Ciclo infinito
        mostrarEstado(estado);
        sleep(1); // Esperar un segundo

        // Cambiar al siguiente estado
        switch (estado) {
            case ROJO:
                estado = VERDE;
                break;
            case AMARILLO:
                estado = ROJO;
                break;
            case VERDE:
                estado = AMARILLO;
                break;
        }
    }

    return 0;
}
```

Implementacion:

1. Se define un enum para los estados.
2. Se utiliza un switch para ejecutar el estado actual.
3. Se utiliza un ciclo infinito para cambiar el estado del semáforo secuencialmente.

¿Qué es un sistema embebido de tiempo real?

- Interacción con el entorno físico respondiendo a los estímulos en un plazo de tiempo determinado.
- Deben ejecutarse las acciones correctas en el instante de tiempo correcto.
- El tiempo en que se ejecutan las acciones del sistema es significativo.
- Riesgo de consecuencias graves, incluyendo el fallo.
- Tiempo real \neq Rápido.

Clasificación...

- Tiempo real **blando** (Soft Real-Time System)
 - Perder un deadline produce la degradación del sistema. Ej: telemetría.
 - Plazo de respuesta flexible
 - Comportamiento temporal determinado por el CPU
 - Comportamiento en sobrecargas degradado
 - Requisitos de seguridad acrícos
 - Recuperación de fallos
- Tiempo real **duro** (Hard Real-Time System)
 - Perder un deadline puede producir la falla del sistema. Ej: control de frenado.
 - Plazo de respuesta estricto
 - Comportamiento temporal determinado por el entorno
 - Comportamiento en sobrecargas predecible
 - Requisitos de seguridad críticos
 - Redundancia activa
- Tiempo real **firme** (Firm Real-Time System)
 - Se pueden perder plazos ocasionalmente. Una respuesta tardía no tiene valor.
Ejemplo: sistemas multimedia.

Multitarea

- Varias tareas ejecutándose “en paralelo” o “simultáneo”.
- **Concurrencia:** Simultaneidad aparente de tareas.
- Arquitecturas que lo implementan:
 - *Máquinas de estado* (cooperativo)
 - *freeRTOS* (preemptivo)

La multitarea **cooperativa**, también llamada **multitarea no-preemptiva**, es una forma de multitarea en la que el sistema operativo **no detiene ni cambia de contexto** un proceso activo de manera automática a otro. En cambio, **los procesos dan voluntariamente el control** en intervalos regulares o cuando están inactivos o bloqueados, permitiendo que diferentes aplicaciones se ejecuten simultáneamente. Este enfoque se llama "cooperativa" porque **todos los programas deben cooperar** y ceder el control para que el esquema funcione.

En la multitarea **preemptiva**, el sistema operativo **asigna tiempos** de CPU a cada proceso que se encuentra en ejecución. **En un núcleo de la CPU, solo puede ejecutarse un proceso a la vez.** El proceso activo recibe un intervalo de tiempo asignado, denominado *slice*, para ejecutarse. **Cuando este tiempo se agota, el proceso se detiene** y el sistema operativo **transfiere el intervalo de ejecución al siguiente proceso.** Eventualmente, el primer proceso recibirá más intervalos para continuar su ejecución hasta completarla, y así sucesivamente.

Para decidir **cuál proceso obtendrá el siguiente intervalo**, el sistema operativo cuenta con un componente llamado **scheduler**. Este puede aplicar diferentes criterios para tomar la decisión.

Conceptos: No preemptivo vs preemptivo

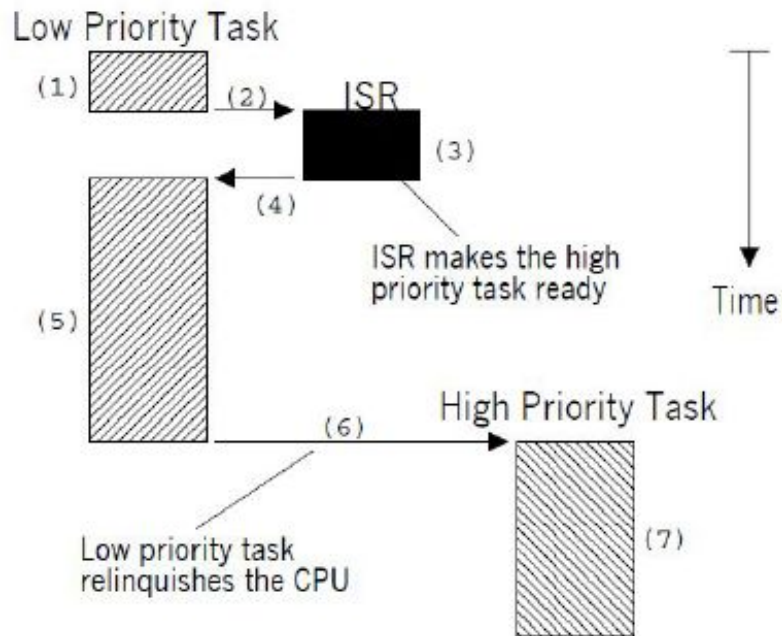


Figure 2-4, Non-preemptive kernel

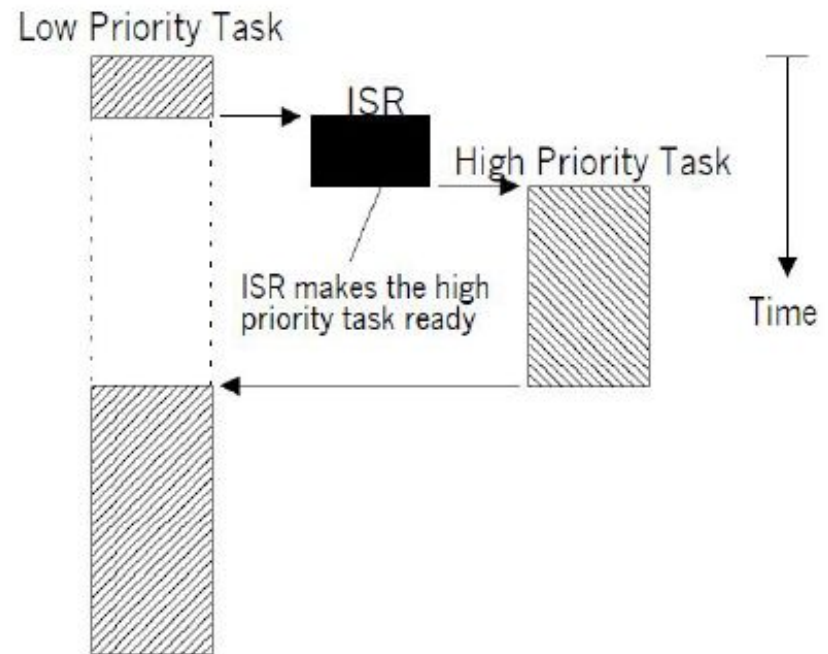


Figure 2-5, Preemptive kernel

Sistema Operativo en Tiempo Real (RTOS)

- Multitasking preemptivo
- Tareas priorizadas y ejecutadas por scheduler
- Prioridades
- Tareas se bloquean si esperan eventos o recursos
- Tiempo de respuesta: suma de los tiempos de ISR + tareas de mayor prioridad

RTOS: Conceptos

RTOS Suave

Las tareas se ejecutan lo más rápido posible pero no tienen un deadline estricto.

RTOS Duro

Las tareas deben ejecutarse rápido, bien, y en tiempo.

Tarea

Es un programa que opera bajo el supuesto de que es lo único ejecutándose en la CPU. Cada tarea tiene una prioridad, su juego de registros de CPU y su stack.

Recurso

Es una entidad utilizada por una tarea. Por ejemplo: cualquier dispositivo de I/O (teclado, display, etc) o una variable.

Recurso compartido

Es un recurso que puede ser utilizado por más de una tarea. Se debe utilizar algún mecanismo para garantizar que un recurso compartido no se utilice por más de una tarea al mismo tiempo. (Exclusión mutua)

RTOS: Conceptos

Sección crítica

Secciones de código que son indivisibles. Una sección crítica no puede ser interrumpida. ¿Cómo? Deshabilitando interrupciones antes y habilitando después (recurso compartido)

Context switch (task switch)

Acción que ocurre cuando el kernel decide correr otra tarea. Se guarda el estado actual de la tarea en ejecución (task's context, CPU registers) en el propio stack de la tarea. Cuantos más registros tenga el CPU, más demora.

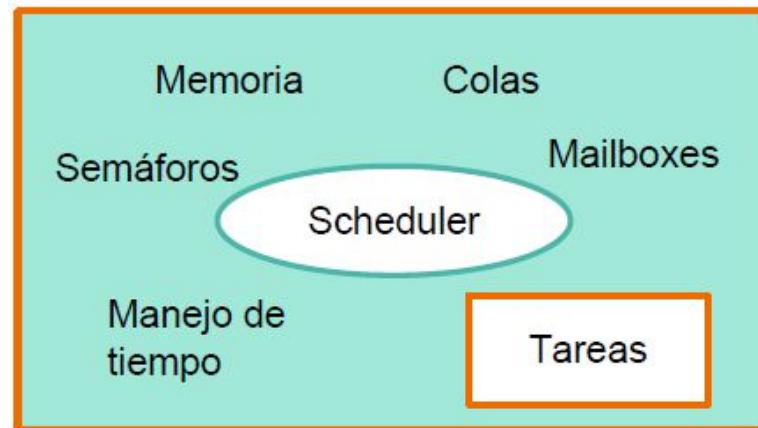
Kernel

La parte del sistema operativo responsable de administrar las tareas y la comunicación entre ellas. Tener en cuenta que ocupa memoria RAM, y ROM, así como recursos de CPU. Permite el uso de semáforos, mailboxes, colas, memoria dinámica, tiempo, etc.

RTOS: Conceptos

Scheduler

Es la parte del kernel responsable de determinar cuál tarea debe ejecutarse. En un kernel basado en prioridades, el scheduler siempre le ejecuta la tarea con mayor prioridad cuyo estado sea READY.



RTOS: Conceptos

Semáforos

Son un mecanismo para permitir el acceso de forma segura a un **recurso compartido**.

Hay dos tipos de semáforos:

- Binary
- Counting.

Un semáforo binario (mutex) es un semáforo contador inicializado en 1.

Las operaciones que se pueden con un semáforo son:

- WAIT / PEND / **TAKE**
- SIGNAL / POST / **GIVE**

RTOS: Conceptos

Comunicación entre tareas

se pueden comunicar a través de variables globales o enviando mensajes. El envío de mensajes es un servicio del kernel:

- Queues (colas)
- Mailboxes
- Memoria

Exclusión mutua

Cuando dos o más tareas comparten variables globales, se debe utilizar algún método para acceder a los datos de forma exclusiva. Las formas de hacer esto son:

1. Deshabilitar interrupciones: **taskENTER_CRITICAL()** y **taskEXIT_CRITICAL()**
2. Deshabilitar el scheduler: **vTaskSuspendAll()** y **xTaskResumeAll()**
3. **Utilizar semáforos (MUTual EXclusive)**

RTOS: Conceptos

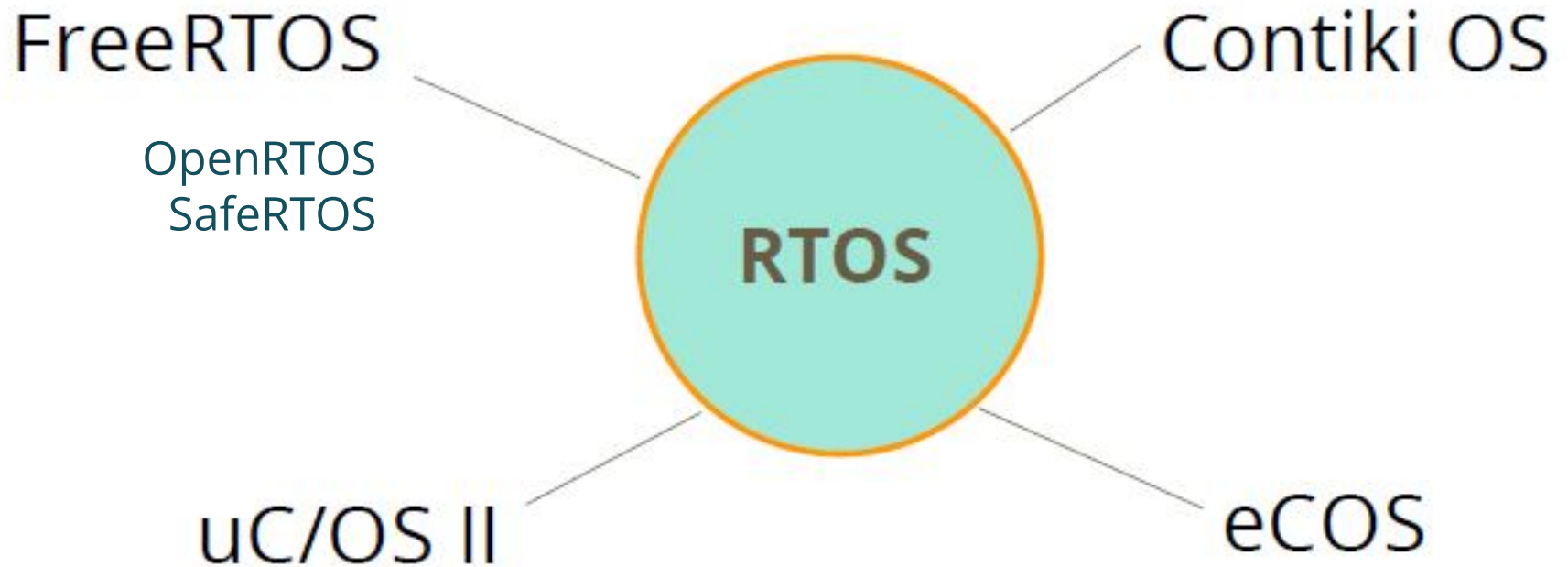
Ventajas:

- **Estabilidad** cuando el código cambia: agregar una tarea no afecta el tiempo de respuesta de las tareas de mayor prioridad.
- **Modularidad:** Agregar nuevas tareas es sencillo, permite también reutilizar código y facilita el testing.
- El **mantenimiento** del código es más simple, esto se debe a permite una mejor estructuración del mismo.
- **Eficiente:** por definición busca cumplir con los deadlines para las distintas tareas y hacerlo de forma satisfactoria.
- **Power management:** El tiempo “Muerto” (IDLE) se utiliza para mejorar el rendimiento energético del dispositivo (sleep).
- Muchas opciones disponibles: comerciales y GNU.

Desventajas:

- **Complejidad:** mucho dentro de RTOS, algo para usarlo.
- **Requerimientos mínimos y overhead del RTOS.**

Sistema Operativo en tiempo real



freeRTOS:

Características:

- Operación preemptiva o cooperativa
- Asignación flexible de prioridades
- Colas (Queues)
- Semáforos
- Timers por software
- Chequeo de stack overflow
- Trace recording
- Recolección de estadísticas en run-time
- Bajo consumo
- GRATIROLA!



freeRTOS: Types

BaseType_t

Está siempre definido como el tipo de datos más eficiente para la arquitectura dada. Típicamente, es 32-bit en una arquitectura de 32-bit, 16-bit en una arquitectura de 16-bit, etc.

BaseType_t se usa generalmente para *returns* de funciones que pueden tomar solo un número muy limitado de valores.

TickType_t

FreeRTOS configura una interrupción periódica llamada “Interrupción de Tick”.

La cantidad de interrupciones de tick que ocurrieron desde que arrancó a correr FreeRTOS se llama *tick count*, y se usa como medida de tiempo.

El tiempo entre dos interrupciones de tick se llama “período de tick”. Los tiempos se especifican como múltiplos de este período.

TickType_t es el tipo de dato usado para la variable que almacena el *tick count*, y para especificar tiempos.

freeRTOS: Tasks

- Son funciones de C con un prototipo especial:

```
void ATaskFunction( void *pvParameters );
```

- Tienen un punto de entrada y generalmente van a estar ejecutándose en un loop infinito.
- No tienen **return**
- Una tarea puede crear a otras múltiples tareas.
- Cada tarea tiene reservado su stack y su copia de variables locales

```
void ATaskFunction( void *pvParameters ) {  
    int32_t lVariableExample = 0;  
    while(1) {  
        // El código que ejecuta la tarea  
    }  
    vTaskDelete(NULL);  
}
```

freeRTOS: Tasks

```
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. (The prefixes added to variable names are described in
    section 1.5, Data Types and Coding Style Guide.) */
    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. The convention used to name API functions is
        described in section 0, Projects that use a FreeRTOS version older than V9.0.0
        must build one of the heap_n.c files. From FreeRTOS V9.0.0 a heap_n.c file is only
        required if configSUPPORT_DYNAMIC_ALLOCATION is set to 1 in FreeRTOSConfig.h or if
        configSUPPORT_DYNAMIC_ALLOCATION is left undefined. Refer to Chapter 2, Heap Memory
        Management, for more information.
        Data Types and Coding Style Guide. */
        vTaskDelete( NULL );
    }
}
```

Listing 12. The structure of a typical task function

freeRTOS: Estados de una tarea

- **SUSPENDED:** la tarea está en memoria, pero no está disponible para el kernel.
- **READY:** cuando puede ser ejecutada pero su prioridad es menor que la de la tarea actualmente en ejecución.
- **RUNNING:** cuando tiene el control de la CPU.
- **BLOCKED (WAITING FOR EVENT):** cuando requiere que suceda un evento para continuar (ej: que se complete una operación de I/O, pase un cierto tiempo, se libere un recurso, etc.).
- **INTERRUPTED:** cuando ocurre una interrupción y la CPU está procesando la rutina de interrupción.

freeRTOS: Estados de una tarea

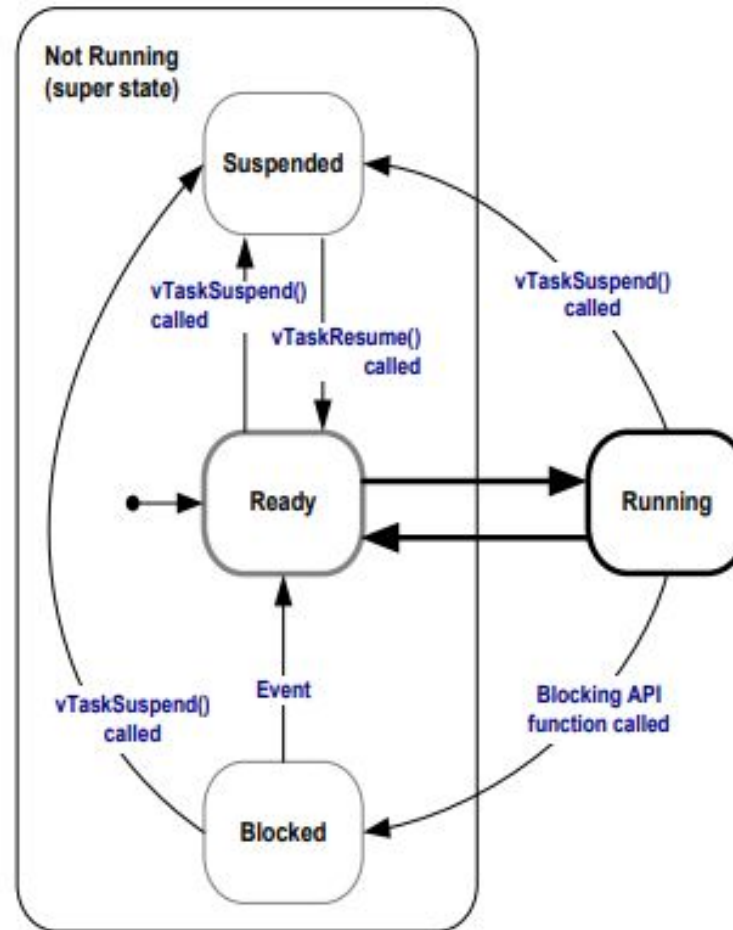


Figure 15. Full task state machine

freeRTOS: Idle Task

- Está corriendo cuando ninguna otra tarea está READY.
- Cumple el rol de “garbage collector”
 - Se encarga de liberar la memoria asignada por el RTOS a las tareas que han sido eliminadas. Por lo tanto, es importante en aplicaciones que utilizan la función *vTaskDelete()* asegurarse de que esta tarea no sea privada de tiempo de procesamiento. La tarea de inactividad no tiene otras funciones activas, por lo que puede legítimamente ser privada de tiempo del microcontrolador en todas las demás condiciones.
- Se le puede asociar una tarea a realizar con la **Idle Task Hook**.
 - El tamaño de stack reservado para esta tarea está definido en el archivo *FreeRTOSConfig.h* por **configMINIMAL_STACK_SIZE**
 - El tamaño de stack se define como la constante * 4 bytes (tamaño de palabra)

freeRTOS: Prioridades de las tareas

- Van de 0 a (`configMAX_PRIORITIES - 1`). Siendo 0 la menor prioridad.
- `configMAX_PRIORITIES` es configurable y no puede ser mayor a 32.
- La Idle Task tiene prioridad definida por `tskIDLE_PRIORITY` y vale 0.
- Varias tareas pueden tener misma prioridad.
 - Si `configUSE_TIME_SLICING` no está definida, se define en 1, y las tareas de igual prioridad comparten procesamiento usando time slice.
- Se puede cambiar la prioridad con `vTaskPrioritySet()`
- Se puede consultar la prioridad de una tarea con `uxTaskPriorityGet()`.

freeRTOS: Ejemplo - Creación de tareas

```
void vTask1( void *pvParameters ) {  
    const char *pcTaskName = "Task 1 is running\n";  
    volatile uint32_t ul;  
    while(1) {  
        printf("%s",pcTaskName );  
        for( ul = 0 ; ul < mainDELAY_LOOP_COUNT ; ul++ );  
    }  
    vTaskDelete(NULL);  
}
```

```
void vTask2( void *pvParameters ) {  
    const char *pcTaskName = "Task 2 is running\n";  
    volatile uint32_t ul;  
    for(;;) {  
        printf("%s",pcTaskName );  
        for( ul = 0 ; ul < mainDELAY_LOOP_COUNT ; ul++ );  
    }  
    vTaskDelete(NULL);  
}
```

freeRTOS: Ejemplo - Creación de tareas

```
void main( void ) {  
  
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 1, NULL);  
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL);  
  
    for(;;);  
}
```


freeRTOS: Ejemplo - Creación de tareas

```
void vTask1( void *pvParameters )
{
    const char *pcTaskName = "Task 1 is running\r\n";
    volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation. There is
             nothing to do in here. Later examples will replace this crude
             loop with a proper delay/sleep function. */
        }
    }
}
```

Listing 14. Implementation of the first task used in Example 1

freeRTOS: Ejemplo - Creación de tareas

```
void vTask2( void *pvParameters )
{
    const char *pcTaskName = "Task 2 is running\r\n";
    volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation. There is
             nothing to do in here. Later examples will replace this crude
             loop with a proper delay/sleep function. */
        }
    }
}
```

Listing 15. Implementation of the second task used in Example 1

freeRTOS: Ejemplo - Creación de tareas

```
int main( void )
{
    /* Create one of the two tasks. Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(    vTask1, /* Pointer to the function that implements the task. */
                  "Task 1", /* Text name for the task. This is to facilitate
                           debugging only. */
                  1000,    /* Stack depth - small microcontrollers will use much
                           less stack than this. */
                  NULL,    /* This example does not use the task parameter. */
                  1,       /* This task will run at priority 1. */
                  NULL ); /* This example does not use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks. If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 2 provides more information on heap memory management. */
    for( ;; );
}
```

Listing 16. Starting the Example 1 tasks

freeRTOS: Ejemplo - Creación de tareas

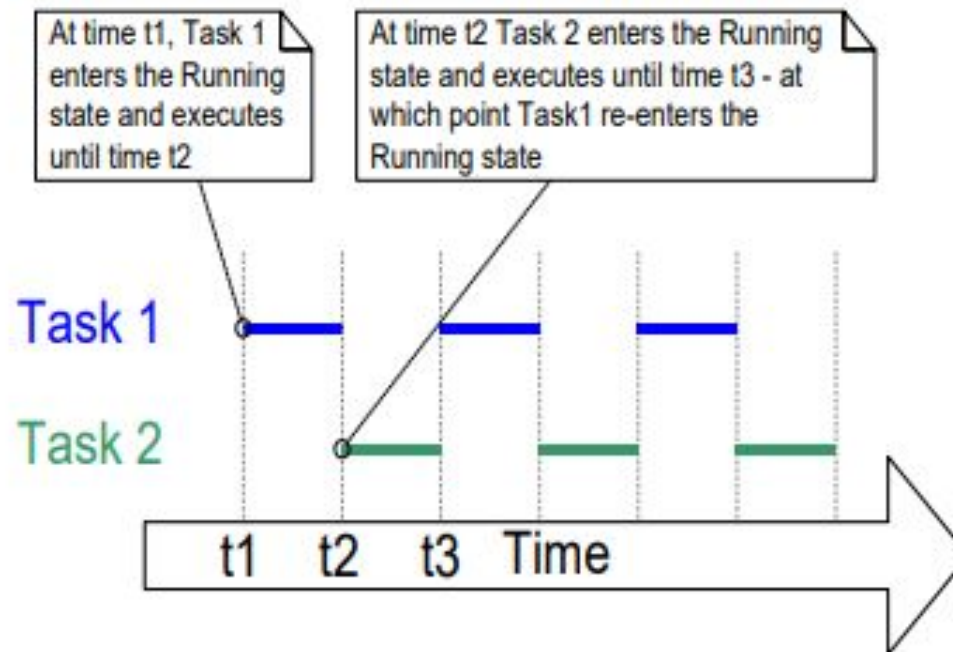


Figure 11. The actual execution pattern of the two Example 1 tasks

freeRTOS: Tareas que crean tareas

```
void vTask1( void *pvParameters ) {  
  
    const char *pcTaskName = "Task 1 is running\n";  
    volatile uint32_t ul;  
  
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL);  
  
    for(;;) {  
        printf("%s",pcTaskName );  
        for( ul = 0 ; ul < mainDELAY_LOOP_COUNT ; ul++ );  
    }  
  
    vTaskDelete(NULL);  
}
```

freeRTOS: Tareas que crean tareas

```
void vTask1( void *pvParameters )
{
    const char *pcTaskName = "Task 1 is running\r\n";
    volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* If this task code is executing then the scheduler must already have
    been started. Create the other task before entering the infinite loop. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation. There is
            nothing to do in here. Later examples will replace this crude
            loop with a proper delay/sleep function. */

        }
    }
}
```

Listing 17. Creating a task from within another task after the scheduler has started

freeRTOS: Ejemplo - Usando parámetros

```
void vTaskFunction( void *pvParameters ) {  
  
    char *pcTaskName;  
    volatile uint32_t ul;  
  
    pcTaskName = (char*)pvParameters;  
  
    for(;;) {  
        printf(“%S”,pcTaskName );  
        for( ul = 0 ; ul < mainDELAY_LOOP_COUNT ; ul++ );  
    }  
  
    vTaskDelete(NULL);  
}
```


freeRTOS: Ejemplo - Usando parámetros

```
static const char *pcTextForTask1 = "Task 1 is running\n";
static const char *pcTextForTask2 = "Task 2 is running\n";

void main( void ) {

    xTaskCreate( vTask1, "Task 1", 100, (void*)pcTextForTask1, 1, NULL);
    xTaskCreate( vTask2, "Task 2", 100, (void*)pcTextForTask2, 1, NULL);

    for(;;);
}
```


freeRTOS: Ejemplo - Usando parámetros

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* The string to print out is passed in via the parameter. Cast this to a
    character pointer. */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation. There is
            nothing to do in here. Later exercises will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Listing 18. The single task function used to create two tasks in Example 2

freeRTOS: Ejemplo - Usando parámetros

```
/* Define the strings that will be passed in as the task parameters. These are defined const and not on the stack to ensure they remain valid when the tasks are executing. */  
static const char *pcTextForTask1 = "Task 1 is running\r\n";  
static const char *pcTextForTask2 = "Task 2 is running\r\n";  
  
int main( void )  
{  
    /* Create one of the two tasks. */  
    xTaskCreate(      vTaskFunction,          /* Pointer to the function that implements the task. */  
                  "Task 1",                 /* Text name for the task. This is to facilitate debugging only. */  
                  1000,                     /* Stack depth - small microcontrollers will use much less stack than this. */  
                  (void*)pcTextForTask1,     /* Pass the text to be printed into the task using the task parameter. */  
                  1,                         /* This task will run at priority 1. */  
                  NULL );                   /* The task handle is not used in this example. */
```

freeRTOS: Ejemplo - Usando parámetros

```
/* Create the other task in exactly the same way. Note this time that multiple
tasks are being created from the SAME task implementation (vTaskFunction). Only
the value passed in the parameter is different. Two instances of the same
task are being created. */
xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 1, NULL );

/* Start the scheduler so the tasks start executing. */
vTaskStartScheduler();

/* If all is well then main() will never reach here as the scheduler will
now be running the tasks. If main() does reach here then it is likely that
there was insufficient heap memory available for the idle task to be created.
Chapter 2 provides more information on heap memory management. */
for( ;; );
}
```

Listing 19. The main() function for Example 2.

freeRTOS: Delays

- Forma que tiene el scheduler de “darse cuenta” que una tarea espera por un tiempo y puede “bloquearse”
Un delay en un loop, como en el ejemplo, no lo hace.

```
void vTaskDelay( TaskType_t xTicksToDelay );
```

- `vTaskDelay(pdMS_TO_TICKS(100))`
Para hacer un delay de 100 ms

freeRTOS: Eliminación de tareas

- Debe estar INCLUDE_vTaskDelete definido en 1 en FreeRTOSConfig.h.
- vTaskDelete()
- Las tareas eliminadas ya no existen y no pueden entrar a un estado “running”
- La Idle Task es la que se encarga de liberar la memoria de cualquier tarea que haya sido eliminada

```
void vTaskDelete( TaskHandle_t pxTaskToDelete );
```

freeRTOS: Ejemplo - Eliminación tareas

```
TaskHandle_t xTask2Handle = NULL;

void vTask1( void *pvParameters ) {
    const TickType_t xDelay100ms = pdMS_TO_TICKS(100UL);

    for(;;) {
        printf( "Task 1 is running\n" );
        xTaskCreate( vTask2, "Task 2", 100, NULL, 2, &xTask2Handle);
        vTaskDelay(xDelay100ms);
    }

    vTaskDelete(NULL);
}
```

freeRTOS: Ejemplo - Eliminación tareas

```
void vTask2( void *pvParameters ) {  
    printf( "Task 2 is running\n" );  
    vTaskDelete(NULL);  
}
```

```
void main( void ) {  
    xTaskCreate( vTask1, "Task 1", 100, NULL, 1, NULL);  
    for(;;);  
}
```


freeRTOS: Ejemplo - Eliminación tareas

```
int main( void )
{
    /* Create the first task at priority 1. The task parameter is not used
    so is set to NULL. The task handle is also not used so likewise is set
    to NULL. */
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 1, NULL );
    /* The task is created at priority 1 _____. */

    /* Start the scheduler so the task starts executing. */
    vTaskStartScheduler();

    /* main() should never reach here as the scheduler has been started. */
    for( ;; );
}
```

Listing 37. The implementation of main() for Example 9

freeRTOS: Ejemplo - Eliminación tareas

```
TaskHandle_t xTask2Handle = NULL;

void vTask1( void *pvParameters )
{
    const TickType_t xDelay100ms = pdMS_TO_TICKS( 100UL );

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task 1 is running\r\n" );

        /* Create task 2 at a higher priority.  Again the task parameter is not
        used so is set to NULL - BUT this time the task handle is required so
        the address of xTask2Handle is passed as the last parameter. */
        xTaskCreate( vTask2, "Task 2", 1000, NULL, 2, &xTask2Handle );
        /* The task handle is the last parameter _____ ^^^^^^^^^^^^^^^^^ */

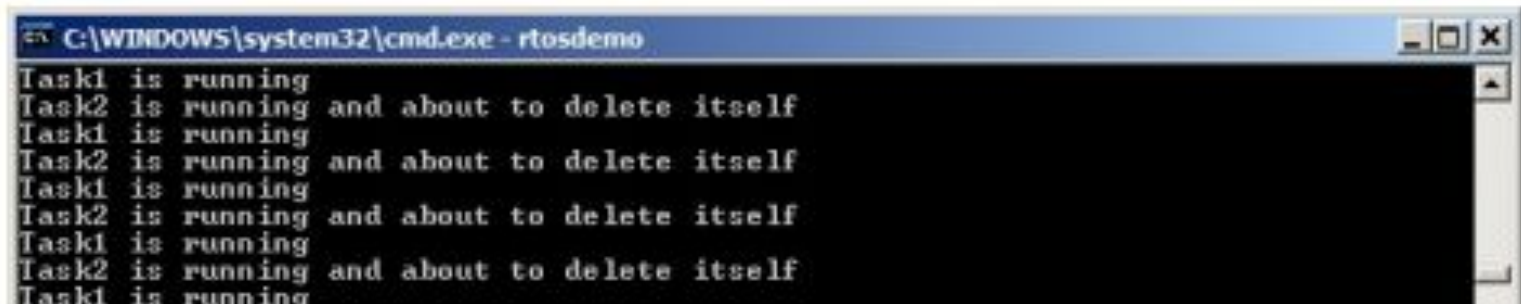
        /* Task 2 has/had the higher priority, so for Task 1 to reach here Task 2
        must have already executed and deleted itself.  Delay for 100
        milliseconds. */
        vTaskDelay( xDelay100ms );
    }
}
```

Listing 38. The implementation of Task 1 for Example 9

freeRTOS: Ejemplo - Eliminación tareas

```
void vTask2( void *pvParameters )
{
    /* Task 2 does nothing but delete itself. To do this it could call vTaskDelete()
    using NULL as the parameter, but instead, and purely for demonstration purposes,
    it calls vTaskDelete() passing its own task handle. */
    vPrintString( "Task 2 is running and about to delete itself\r\n" );
    vTaskDelete( xTask2Handle );
}
```

Listing 39. The implementation of Task 2 for Example 9



```
C:\WINDOWS\system32\cmd.exe - rtosdemo
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
Task2 is running and about to delete itself
Task1 is running
```

freeRTOS: Time slices - Tick

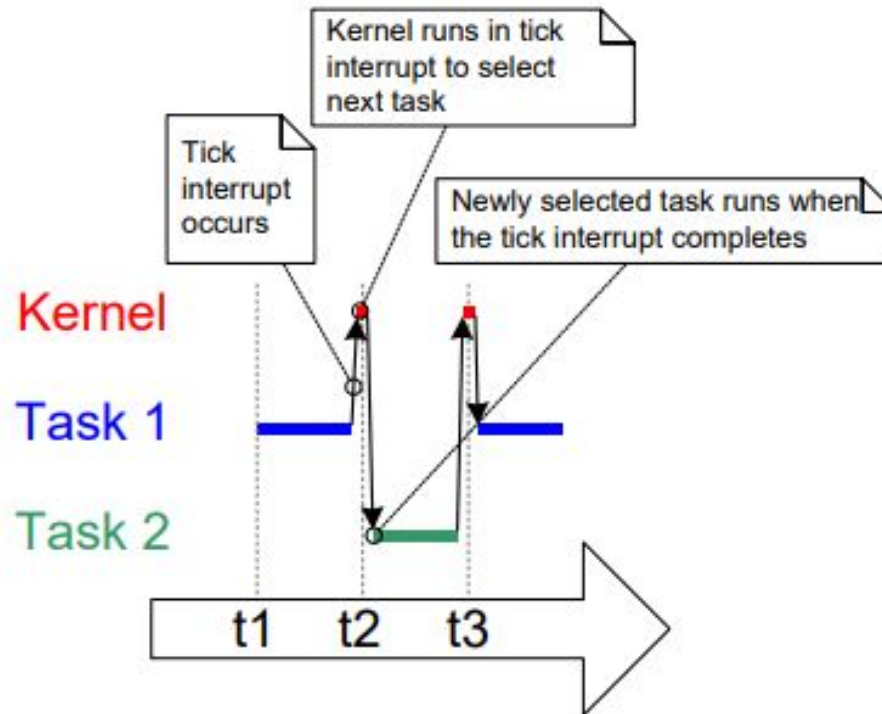


Figure 12. The execution sequence expanded to show the tick interrupt executing

Conceptos: Func. Reentrante

Función reentrante: Se puede acceder desde más de una tarea a la vez, sin que los datos se corrompan.

```
void addOneHundred( long var ) {  
    return var + 100;  
}
```

Ejemplo de NO reentrante

```
long total = 0;  
void addOneHundredToTotal( void ) {  
    return total + 100;  
}
```

Conceptos: Func. Reentrante

Función reentrante: Se puede acceder desde más de una tarea a la vez, sin que los datos se corrompan.

```
/* A parameter is passed into the function. This will either be passed on the stack,
or in a processor register. Either way is safe as each task or interrupt that calls
the function maintains its own stack and its own set of register values, so each task
or interrupt that calls the function will have its own copy of lVar1. */
long lAddOneHundred( long lVar1 )
{
/* This function scope variable will also be allocated to the stack or a register,
depending on the compiler and optimization level. Each task or interrupt that calls
this function will have its own copy of lVar2. */
long lVar2;

    lVar2 = lVar1 + 100;
    return lVar2;
}
```

Listing 112. An example of a reentrant function

Conceptos: Func. Reentrante

EJEMPLO DE FUNCIÓN NO REENTRANTE

```
/* In this case lVar1 is a global variable, so every task that calls
lNonsenseFunction will access the same single copy of the variable. */
long lVar1;

long lNonsenseFunction( void )
{
    /* lState is static, so is not allocated on the stack. Each task that calls this
    function will access the same single copy of the variable. */
    static long lState = 0;
    long lReturn;

    switch( lState )
    {
        case 0 : lReturn = lVar1 + 10;
                 lState = 1;
                 break;

        case 1 : lReturn = lVar1 + 20;
                 lState = 0;
                 break;
    }
}
```

Listing 113. An example of a function that is not reentrant


Conceptos: Func. Reentrante

Baja prioridad

```
...  
while(1) {  
    x = 1;  
    y = 1;  
  
    swap(&x, &y);  
  
    vTaskDelay(10);  
}
```

Alta prioridad

```
...  
while(1) {  
    z = 3;  
    t = 4;  
  
    swap(&z, &t);  
  
    vTaskDelay(10);  
}
```



```
void swap(int *a, int* b) {  
    temp = *a;  
    *a = *b;  
    *b = temp;  
}
```

Conceptos: Func. Reentrante

Baja prioridad

```
...  
while(1) {  
    x = 1;  
    y = 1;
```

```
    swap(&x, &y) {  
        temp = *x;
```

```
        *x = *y;  
        *y = temp;
```

```
    }
```

```
    vTaskDelay(10);
```

```
}
```

temp == 1

ISR → OS

OS

temp == 3

Alta prioridad

```
...  
while(1) {  
    z = 3;  
    t = 4;
```

```
    swap(&z, &t) {  
        temp = *z;
```

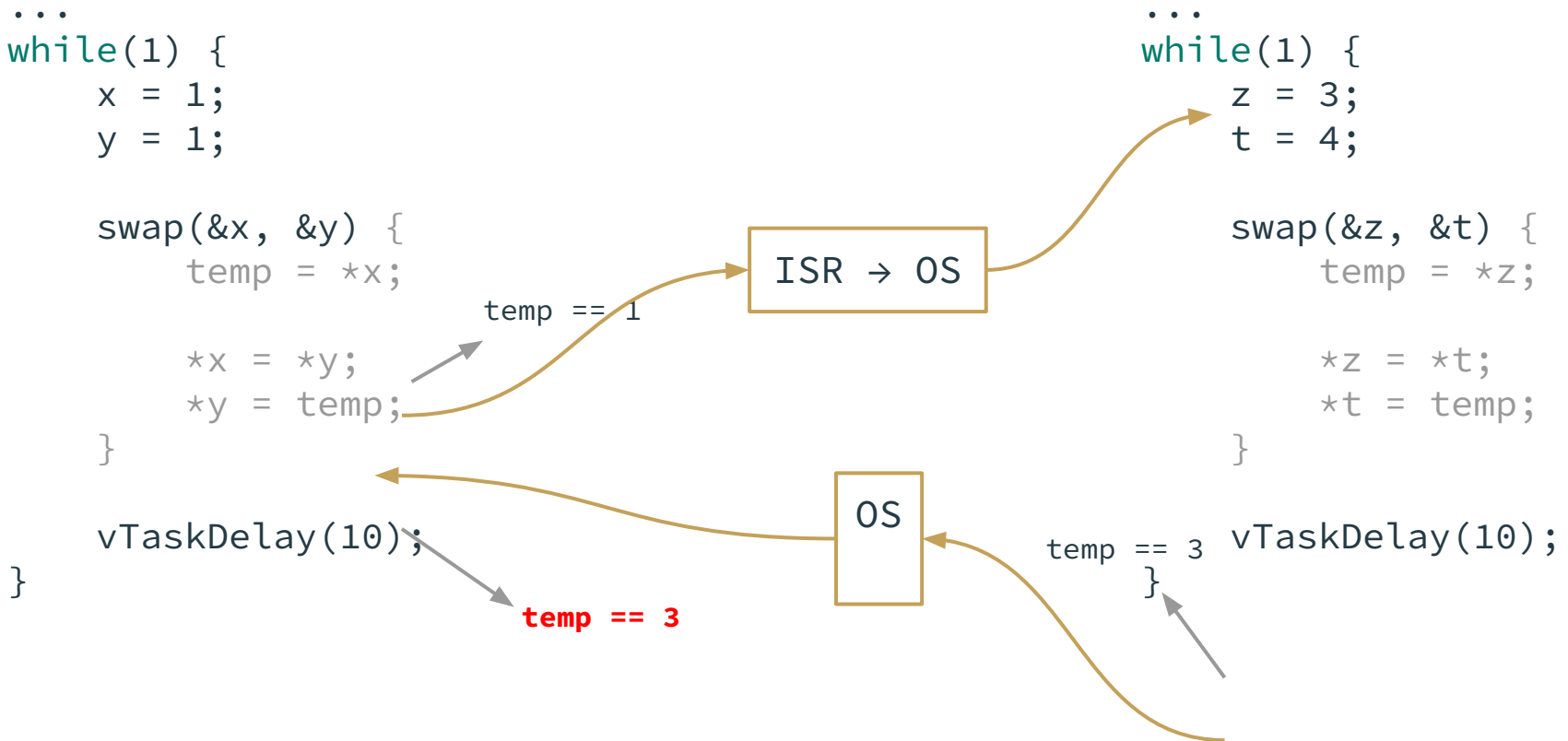
```
        *z = *t;  
        *t = temp;
```

```
    }
```

```
    vTaskDelay(10);
```

temp == 3

```
}
```



freeRTOS: Sincronización

Para la sincronización entre tareas y uso de recursos compartidos, se utilizan:

- Binary semaphore (sincronización)
- Mutex (y recursive mutex)
- Counting semaphore

freeRTOS: Binary Semaphore

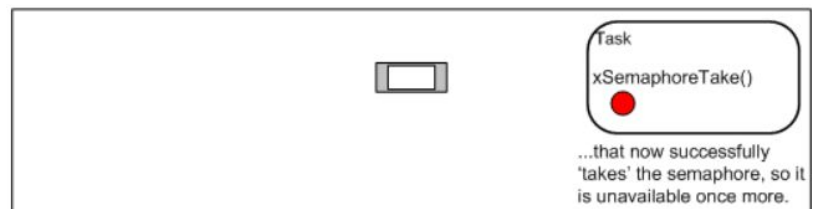
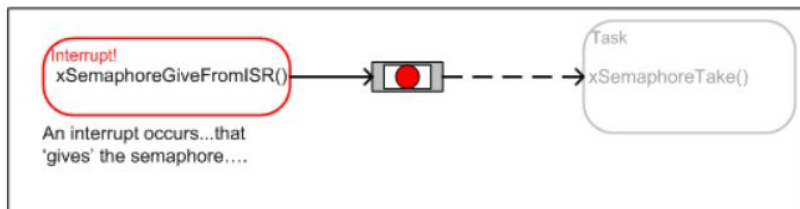
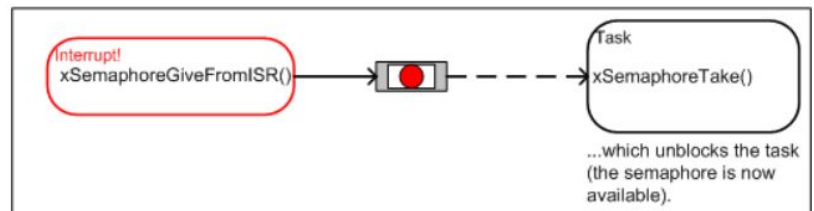
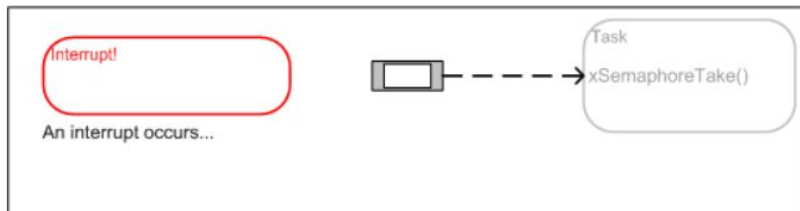
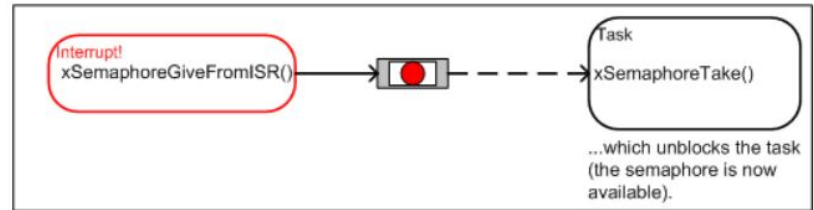
- Más apropiado para sincronización y señalización entre tareas
- Funciones:

```
SemaphoreHandle_t xSemaphoreCreateBinary(void);  
SemaphoreHandle_t xSemaphoreCreateBinaryStatic(void);
```

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

```
bool xSemaphoreGive(SemaphoreHandle_t xSemaphore);
```

freeRTOS: Binary Semaphore



freeRTOS: Mutex

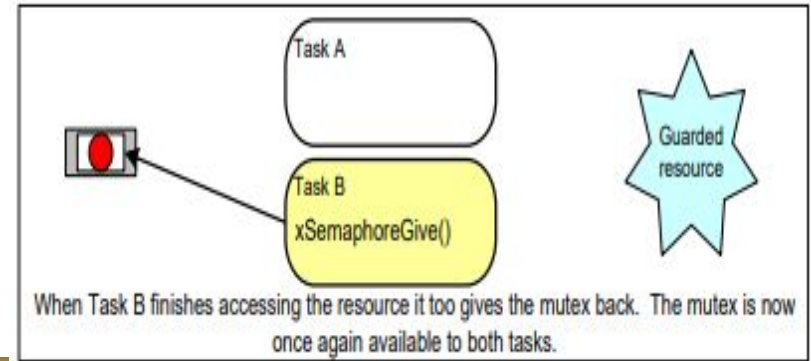
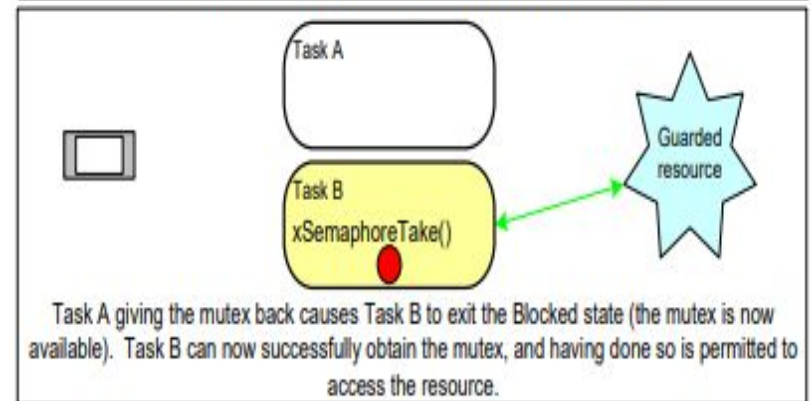
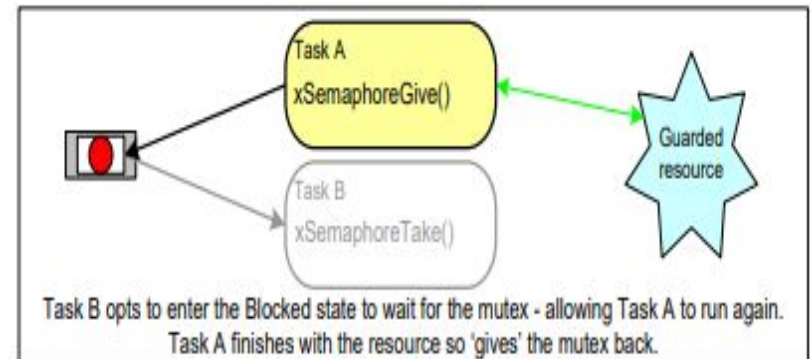
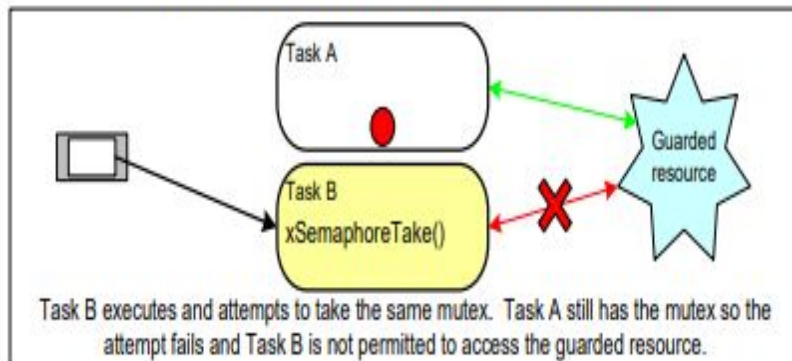
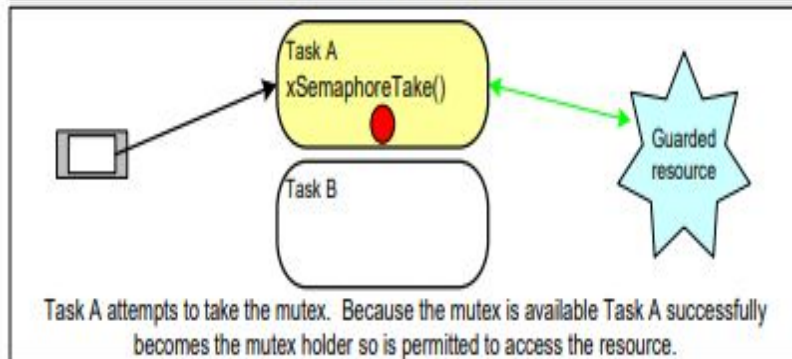
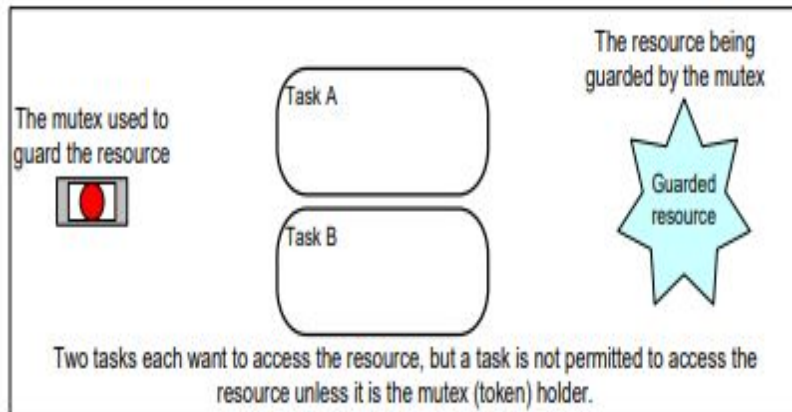
- Controla el acceso a un recurso compartido por dos o más tareas.
- Funciones:

```
SemaphoreHandle_t xSemaphoreCreateMutex(void);  
SemaphoreHandle_t xSemaphoreCreateMutexStatic(void);
```

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

```
bool xSemaphoreGive(SemaphoreHandle_t xSemaphore);
```

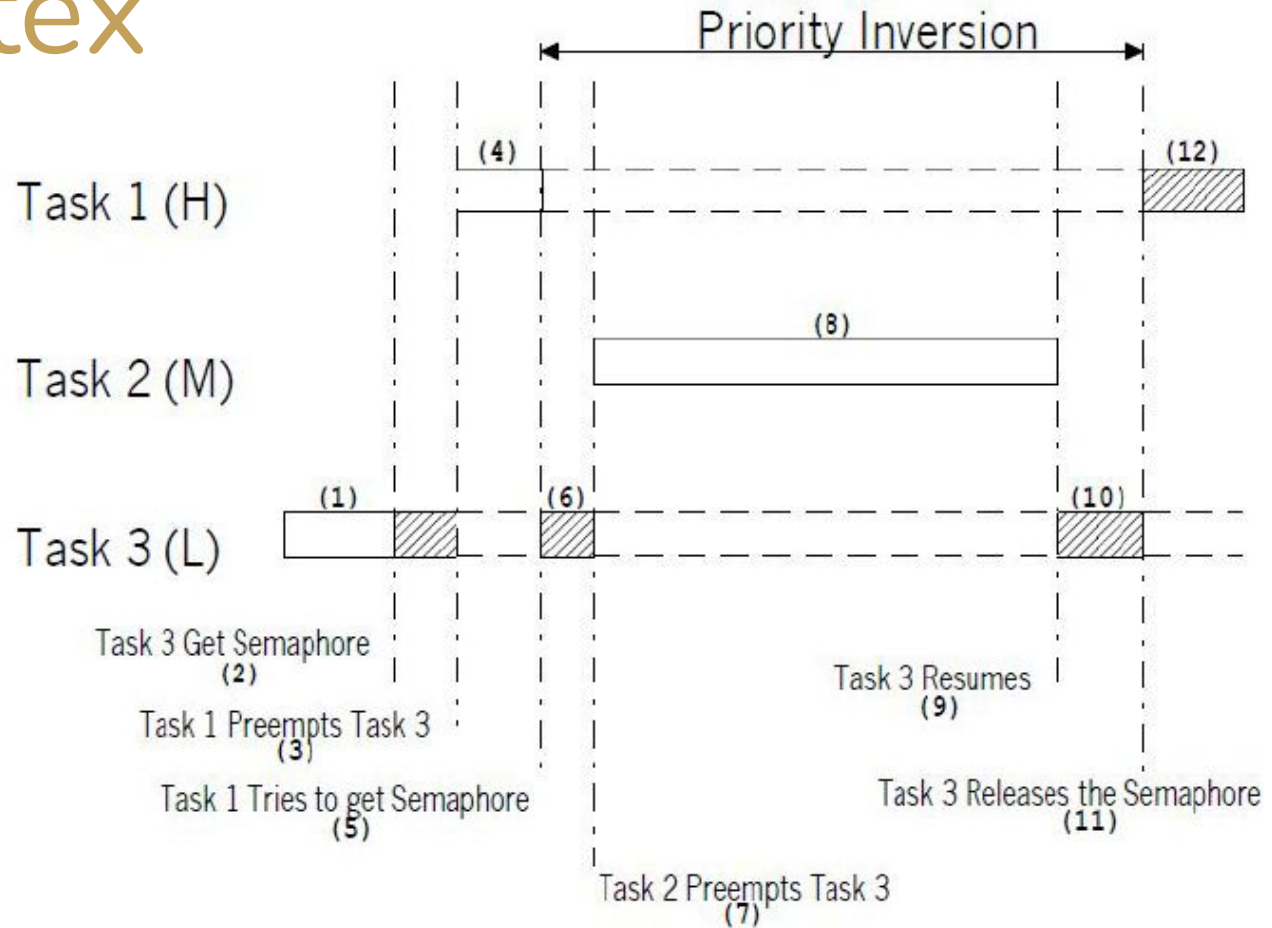
freeRTOS: Mutex



RTOS: Mutex

Inversión de prioridades

Mutex implementan “priority inheritance” para resolverlo



freeRTOS: Mutex recursivo

- Similar al mutex, pero permite que sea tomada varias veces por una misma task. Debe ser devuelto tantas veces como fue tomado antes de estar disponible para otra task.
- Funciones:

```
SemaphoreHandle_t xSemaphoreCreateRecursiveMutex(void);  
SemaphoreHandle_t xSemaphoreCreateRecursiveMutexStatic(void);
```

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

```
bool xSemaphoreGive(SemaphoreHandle_t xSemaphore);
```

freeRTOS: Queues

- Cola de datos de algún tipo con un largo determinado. El largo y el tipo se fijan al crearse.
- FIFO
- Pueden guardar datos (queue by copy) o punteros (queue by reference)

freeRTOS: Queues

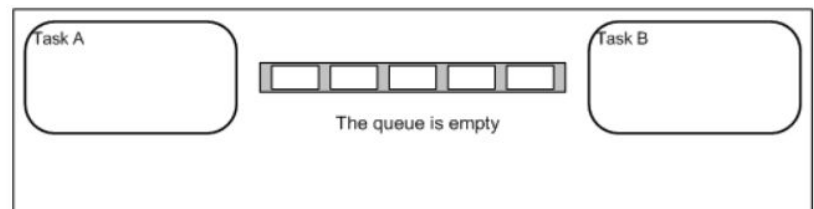
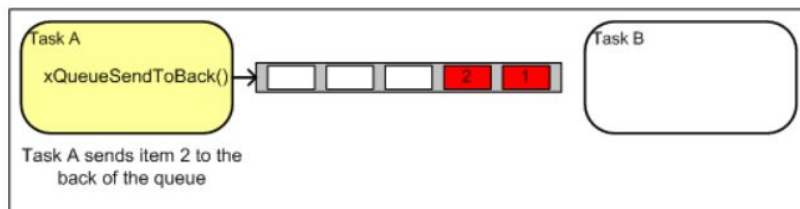
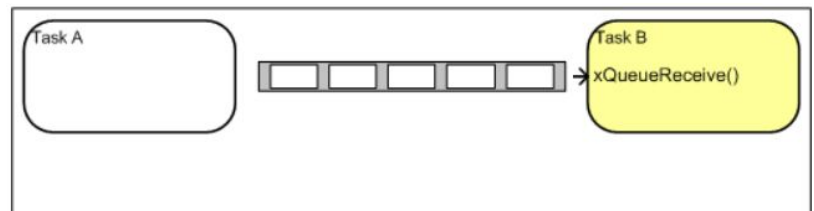
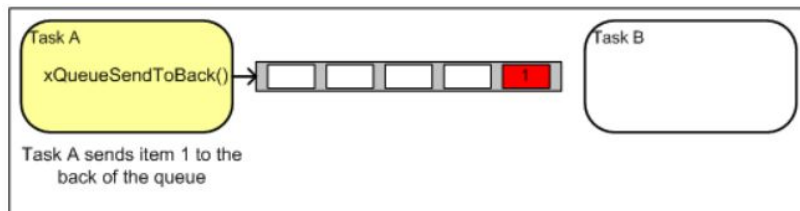
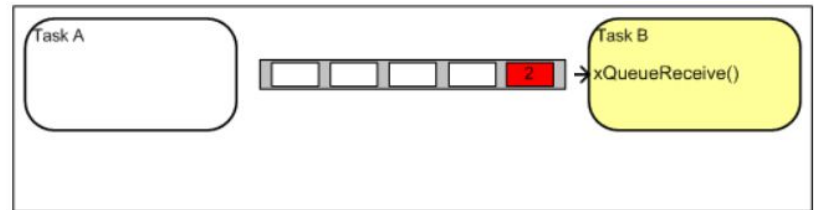
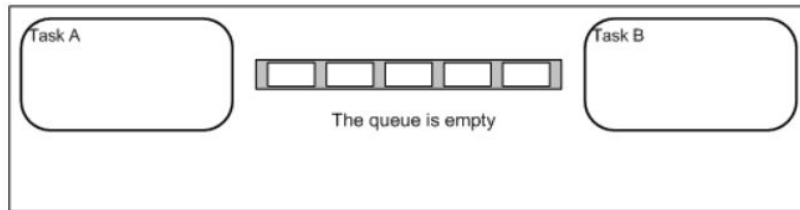
```
QueueHandle_t xQueueCreate (UBaseType_t uxQueueLength, UBaseType_t uxItemSize);
```

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

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

```
UBaseType_t uxQueueMessagesWaiting (QueueHandle_t xQueue);
```

freeRTOS: Queues



freeRTOS: Timers

- Permite que una función (callback) se ejecute en el futuro (por única vez, o de forma periódica)
- Funciones:

```
TimerHandle_t xTimerCreate (const char * const pcTimerName,  
                             const TickType_t xTimerPeriod,  
                             const UBaseType_t uxAutoReload,  
                             void * const pvTimerID,  
                             TimerCallbackFunction_t pxCallbackFunction);  
  
void vTimerCallback (TimerHandle_t xTimer); //Callback
```

freeRTOS: Memoria dinámica

- FreeRTOS gestiona la memoria heap, y existen funciones para solicitar y liberar dinámicamente bloques de memoria
- Funciones:

```
void* malloc (size_t size);  
void* calloc (size_t nitems, size_t size);  
void* realloc (void *ptr, size_t size);  
  
void free (void* ptr);
```

freeRTOS: Memoria dinámica vs estática

- El kernel del RTOS necesita RAM cada vez que se crea una tarea, una cola, un semáforo u otro evento.
- La memoria se puede asignar de forma dinámica por el RTOS o estática por el programador. Cuál utilizar depende del programador y cada una tiene ventajas y desventajas.

freeRTOS: Ventajas de memoria dinámica

- Mayor simplicidad
- Potencial disminución del máximo de RAM utilizado:
 - Menos parámetros necesarios
 - La memoria aloja la API del RTOS
 - El programador no debe preocuparse del alojamiento de memoria.
 - Reutilización de RAM entre objetos

freeRTOS: Ventajas de memoria estática

- Mayor control por parte del programador:
 - Objetos colocados en direcciones de memoria específicas.
 - Fijar el máximo de RAM a utilizar a nivel de linker y no de ejecución.
 - Permite utilizar el RTOS en sistemas que no permiten manejo de memoria dinámico

Referencias

- Richard Barry: ***Mastering the FreeRTOS™ Real Time Kernel. A Hands-On Tutorial Guide.*** Real Time Engineers Ltd. 2016.
- Brian Amos: **Hands-On RTOS with Microcontrollers.** Building real-time embedded systems using FreeRTOS, STM32 MCUs, and SEGGER debug tools. Packt Publishing Ltd. May, 2020.
- ***<https://www.freertos.org/index.html>***