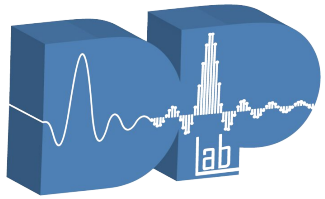
Introducción a los Sistemas Operativos de Tiempo Real

UTN.BA
INGENIERÍA
ELECTRÓNICA

Ing. Pablo Ridolfi
pridolfi@frba.utn.edu.ar

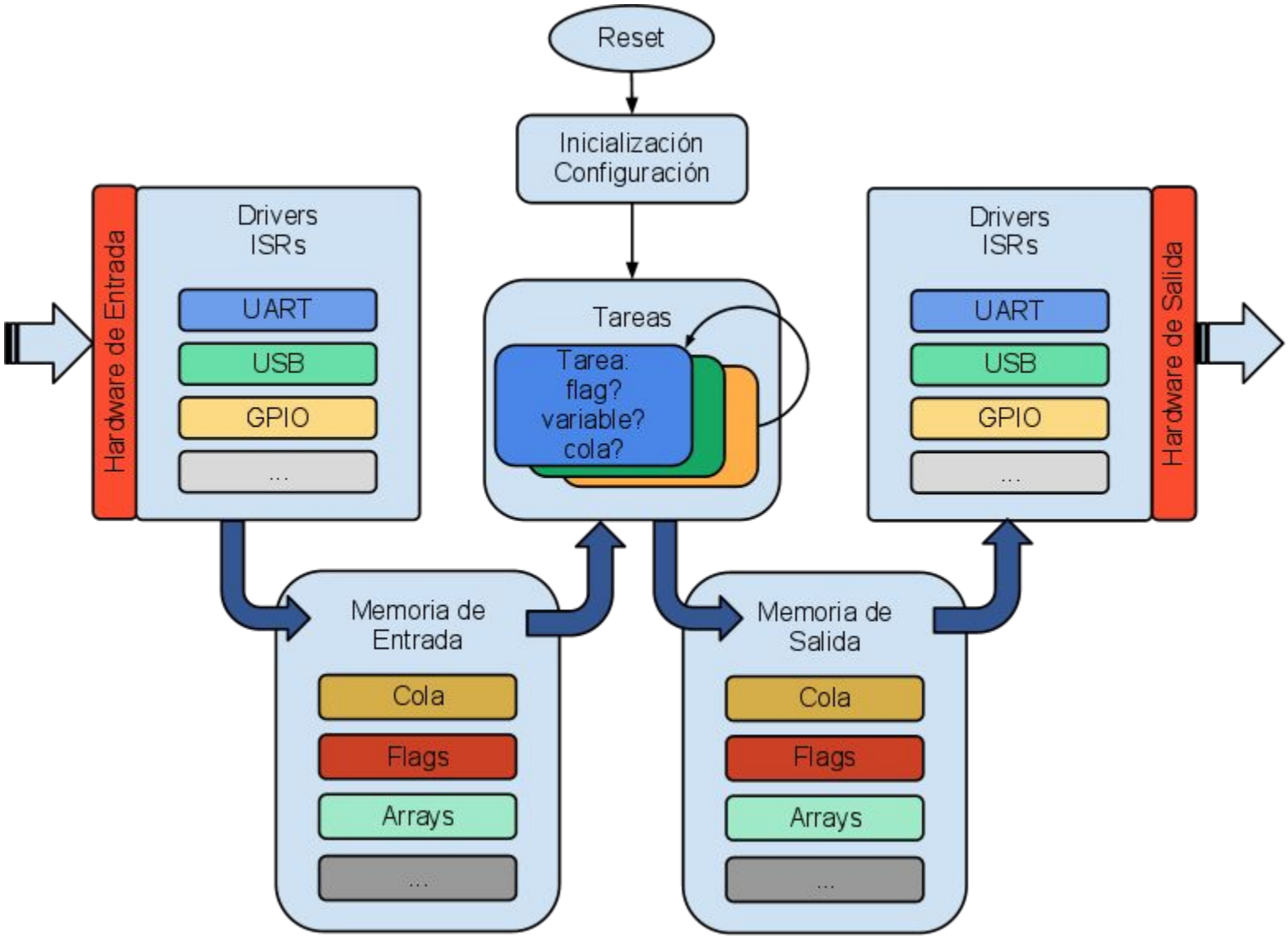
Agosto de 2015



Agenda del curso

- Introducción
  - Modelo de programación “bare-metal”
  - ¿Por qué un RTOS?
- Características generales de un RTOS
  - Tareas
  - Scheduler
  - Mecanismos de sincronización
  - Device drivers
- Casos de estudio
  - FreeRTOS (www.freertos.org)
  - OSEK-OS (www.osek-vdx.org)
  - µPOSIX (github.com/pridolfi/uPOSIX)

Modelo de programación



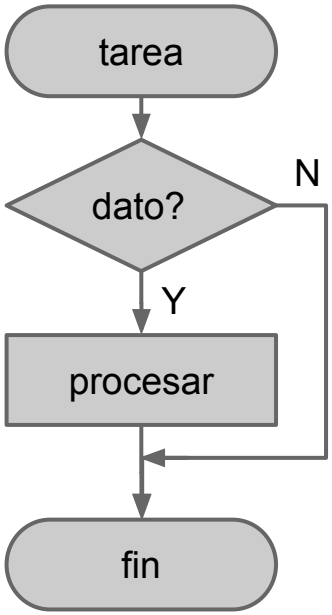
Ing. Juan Manuel Cruz, “El diseño en ingeniería electrónica”

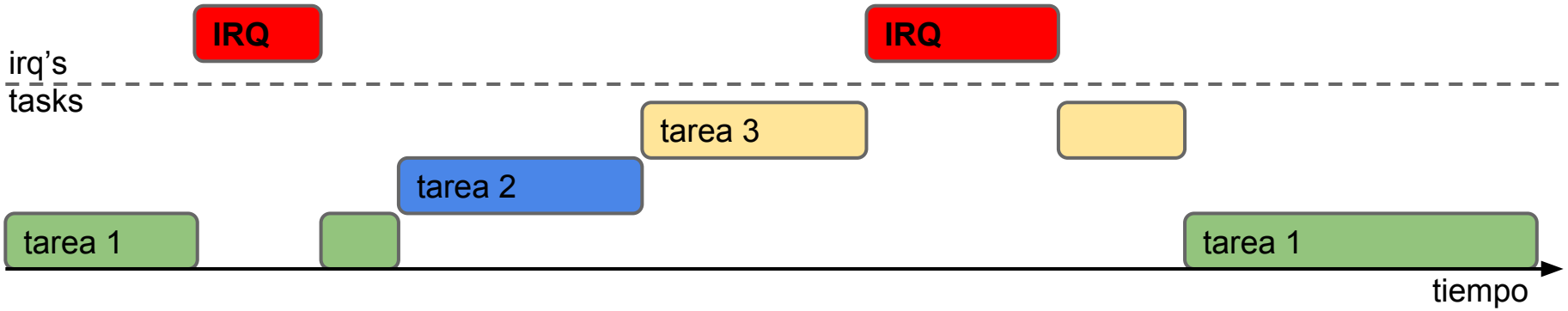
Modelo de programación bare-metal

- Se denomina programación “bare-metal” cuando el MCU no utiliza recursos de un sistema operativo.
- Ejemplos:
  - Foreground-background.
  - Scan-loops.
- Generalmente el comportamiento es *cooperativo*.

Modelo de programación bare-metal

```
while(1)
{
   tarea1();
   tarea2();
   tarea3();
   ...
}
```





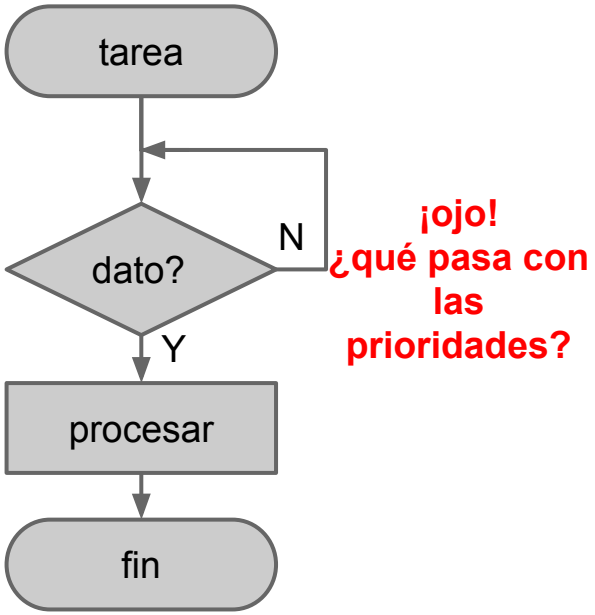
¿Por qué un RTOS?

- Diseñar un programa teniendo presente el comportamiento cooperativo implica una mayor carga para el programador.
- Una de las principales ventajas de usar un OS es la *multitarea*. Es decir que un *hilo* puede suspender su ejecución para que otro hilo se ejecute sin que el hilo original pierda su *contexto*.
- Por eso decimos que usar un OS *aumenta la confiabilidad del sistema*.

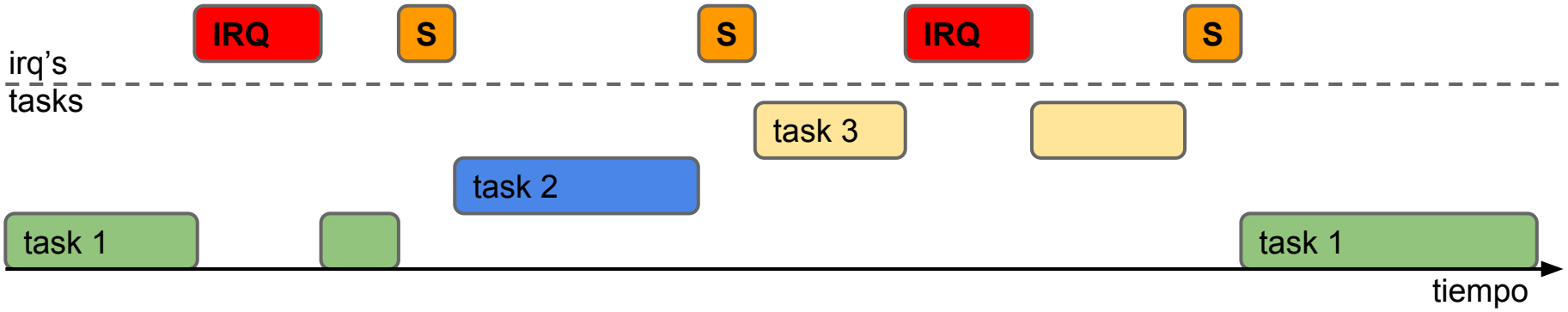
Modelo de programación multitarea

```
int main(void)
{
     addTask(tarea1);
     addTask(tarea2);
     addTask(tarea3);
     startScheduler();
}

void tarea1(void)
{
     ...
     while(1)
     {
          ...
     }
}
```



¡ojo! ¿qué pasa con las prioridades?



¿Por qué Tiempo Real?

- Existen aplicaciones donde el *tiempo de respuesta* a un estímulo del sistema es un parámetro crítico.
- Es requisito que la respuesta del sistema se ubique dentro de una ventana de tiempo. Respuestas demasiado *tempranas* como demasiado *tardías* pueden ser indeseables.
- Por eso decimos que además de la confiabilidad, el uso de un RTOS contribuye a aumentar el *determinismo* del sistema.

¿Por qué Tiempo Real?

- El RTOS pone a disposición del programador recursos que permiten asegurar la respuesta del sistema, con cierta tolerancia.
  - Tareas con **prioridad** definida.
  - **Semáforos** de diferentes tipos (counting, mutex).
  - Recursos desbloqueables desde **IRQs**.
- En función de los requerimientos para dicha tolerancia, podemos clasificar a los sistemas en dos grandes grupos.
  - *Soft* Real-Time.
  - *Hard* Real-Time.

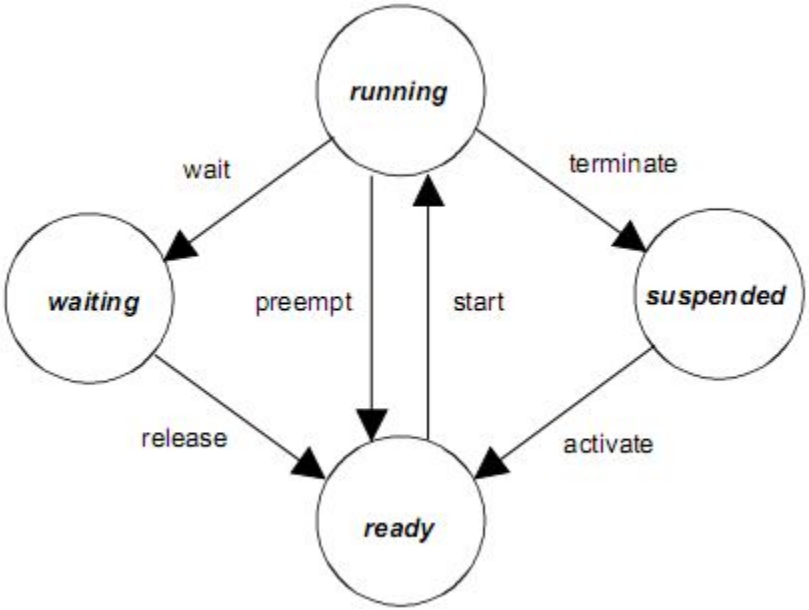
¿Por qué Tiempo Real?

- Un sistema soft Real-Time intenta asegurar el tiempo de respuesta, aunque no se considera crítico que dicho tiempo no se cumpla ya que el sistema sigue siendo utilizable (ejemplo: interfaz de usuario).

- Un sistema hard Real-Time **debe** responder dentro de un período de tiempo definido por los requerimientos del sistema. Una respuesta por fuera de ese período puede significar una falla grave (ejemplo: airbag).

Conceptos

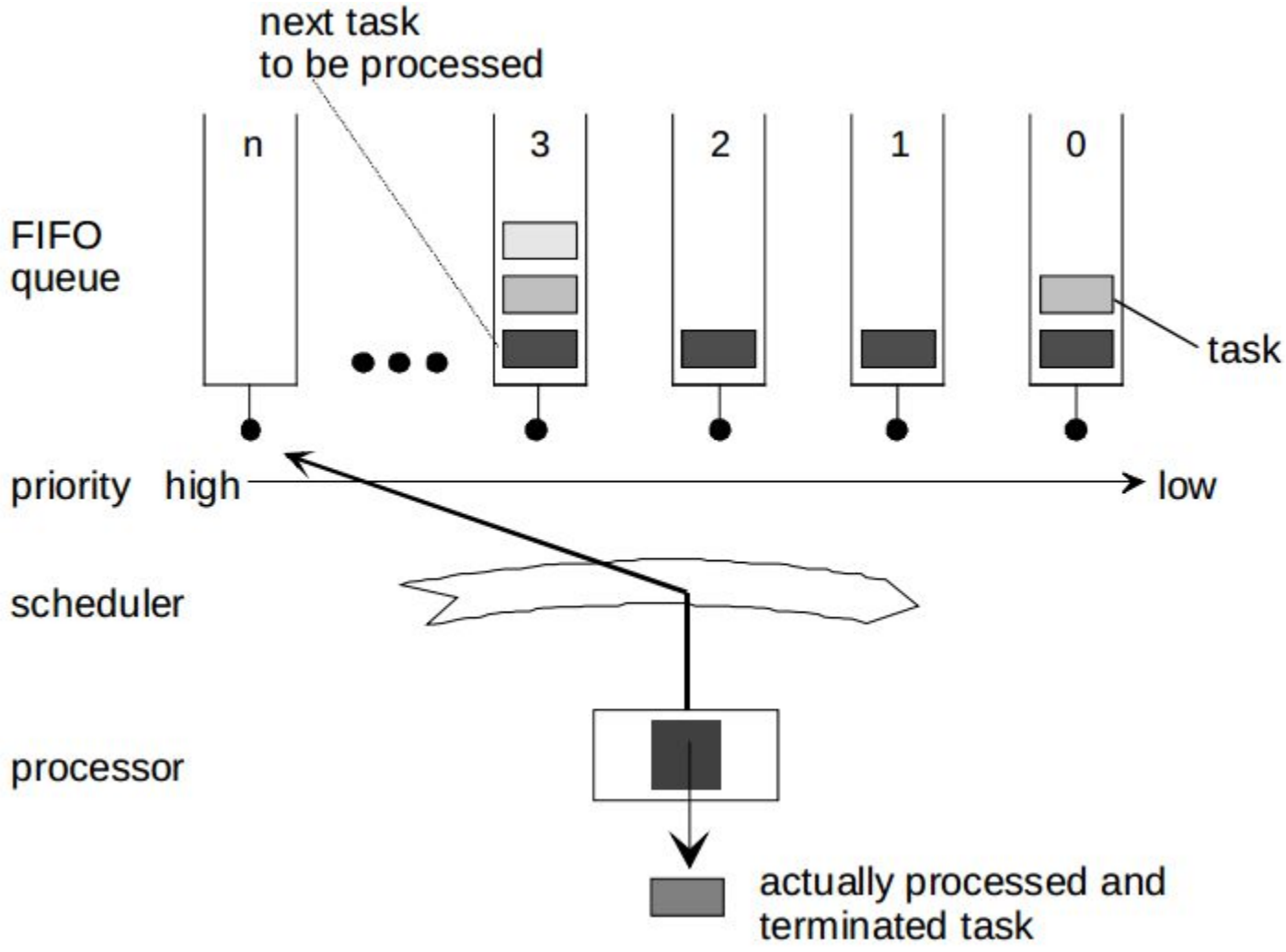
- **Tarea**: Se refiere a cada una de las rutinas de ejecución. También conocidas como *threads*, normalmente solo una se encuentra en ejecución (RUNNING) mientras que las demás esperan su turno para ejecutarse.



Conceptos

- A cada tarea se le asigna una **prioridad**. En función de dicha prioridad será seleccionada para ejecutarse. Quien determina la próxima tarea a ejecutar es el planificador o **scheduler**.

Conceptos: Scheduler



Conceptos

- Muchas veces es necesario *bloquear* una tarea hasta que ocurra un determinado evento. El OS provee mecanismos que brindan esta funcionalidad:
  - Demoras (delays)
  - Semáforos
  - Colas de mensajes
- En otros casos es necesario evitar accesos concurrentes a recursos compartidos por más de una tarea. En ese caso se suelen utilizar semáforos especiales denominados *mutex* (*mut*ual *ex*clusion locks).

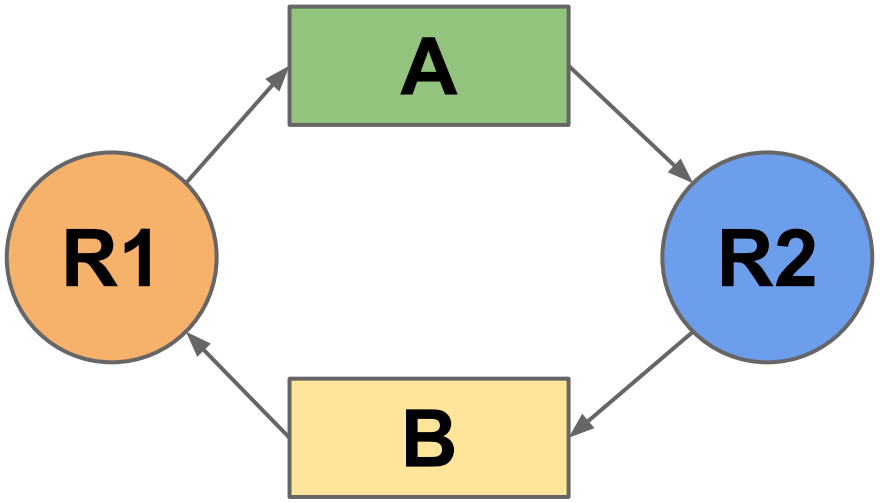
Conceptos: Políticas de Scheduling

- Round Robin (SCHED_RR):
  - Es el esquema visto. Las tareas en estado READY se ordenan en listas según su prioridad y el scheduler irá seleccionando tareas desde la lista de mayor prioridad hacia las de menor prioridad. Mientras haya tareas READY en una lista de prioridad N, las tareas de prioridad N-1 no serán ejecutadas. Tareas de igual prioridad serán ejecutadas alternadamente.
- FIFO (SCHED_FIFO):
  - Similar a Round-Robin, salvo que tareas de igual prioridad serán tratadas en forma cooperativa (la tarea debe ceder el CPU).

Conceptos

- **Deadlock** o **Bloqueo mutuo**.

Ocurre cuando dos (o más) hilos de ejecución toman recursos utilizados por el otro hilo. Esto puede provocar que ambos hilos esperen la liberación del recurso tomado por el otro y se bloqueen indefinidamente.



Conceptos

- **Inversión de prioridades**

Ocurre cuando dos hilos de diferente prioridad comparten el acceso a un recurso, que es obtenido por la tarea de más baja prioridad. Al ejecutarse la de alta prioridad e intentar acceder a ese recurso, se bloqueará, lo que resulta en una inversión de prioridades de esas tareas. Este comportamiento no es deseable ya que *puede afectar el tiempo de respuesta de la tarea de alta prioridad*.

  - Los RTOS suelen proveer mecanismos para evitar o minimizar los efectos de la inversión de prioridades.

Primer Caso de Estudio

www.freertos.org

Introducción

- FreeRTOS incluye:
  - Creación dinámica de tareas.
  - Prioridades estáticas.
  - Funciones de delay con ajuste por tiempo de ejecución.
  - Semáforos binarios, contadores y mutex.
  - Colas de mensaje con tamaño de datos variable.
  - Manejo de IRQs con API especializada.

Hello World!

```
void taskLED(void *pvParameters)
{
    while (1)
    {
        Board_LED_Toggle(0);
        vTaskDelay(500 / portTICK_RATE_MS);
    }
}

int main(void)
{
    prvSetupHardware();
    xTaskCreate(taskLED, (signed char *) "taskLED",
        configMINIMAL_STACK_SIZE, NULL,
        (tskIDLE_PRIORITY + 1UL),
        (xTaskHandle *) NULL);
    vTaskStartScheduler();
    return 1;
}
```

Creación y control de tareas

- **xTaskCreate**
  - Iniciar una tarea desde el principio.
- **vTaskDelete**
  - Finalizar una tarea.
- **vTaskDelay**
  - Detener una tarea un tiempo determinado.
- **vTaskDelayUntil**
  - Detener una tarea un tiempo determinado, teniendo en cuenta una referencia de tiempo anterior.
- **vTaskPrioritySet** / **uxTaskPriorityGet**
  - Cambiar / Obtener la prioridad de una tarea.
- **vTaskSuspend**
  - Suspender la ejecución de una tarea.
- **vTaskResume** / **vTaskResumeFromISR**
  - Resumir la ejecución de una tarea.

Creación y control de tareas

- **API de FreeRTOS:**
  - Creación de tareas: http://www.freertos.org/a00019.html
  - Control de tareas: http://www.freertos.org/a00112.html
  - Funciones auxiliares: http://www.freertos.org/a00021.html

Creación y control de tareas

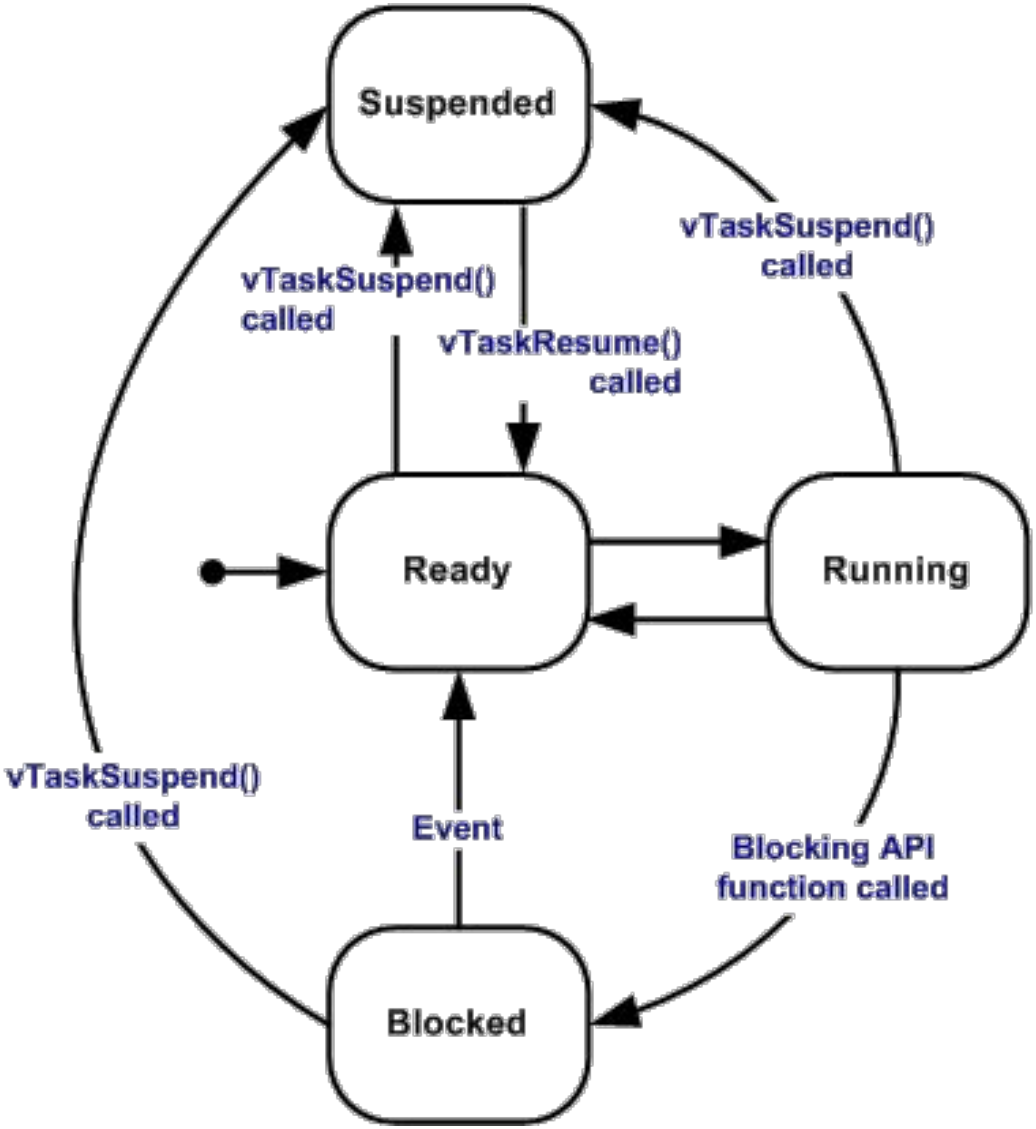
```
void taskFunction(void *pvParameters)
{
   int variable; /* ¿instancias? */

   /* sentencias de inicialización */

   while ( /* condición de lazo */ )
   {
      /* sentencias del lazo */
   }

   vTaskDelete(0); /* nunca return; !! */
}
```

Estados de las tareas



Scheduling cooperativo

- `#define configUSE_PREEMPTION 1`
  - Determina el comportamiento del scheduler (cooperativo o expropiativo/preemptive).

- `taskYield();`
  - Esta función se utiliza para llamar explícitamente al scheduler.

Ejemplos

- Creación de tareas.
- Prueba de prioridades.
- Control de ejecución.
- Comportamiento cooperativo y preemptive.

Sincronización y comunicación entre tareas

- FreeRTOS provee mecanismos para intercambiar información entre tareas y controlar el acceso a recursos.
  - Colas (queues).
  - Semáforos binarios (binary semaphores).
  - Semáforos contadores (counting semaphores).
  - Mutexes.
  - Mutexes recursivos (recursive mutexes).

Colas

- Las colas permiten intercambiar datos entre las tareas. El tamaño de cada ítem y su cantidad almacenable en la cola se definen al momento de su creación.

- Las funciones de lectura y escritura pueden tener comportamientos bloqueantes: La tarea puede bloquearse si la cola está vacía (hasta que haya un dato disponible) o llena (haya lugar para guardar un nuevo dato).
  - También puede especificarse un *timeout*.

Colas

- **xQueueCreate**
  - Crear una cola nueva, especificando cantidad de ítems y tamaño de cada uno.
- **xQueueSend / xQueueSendFromISR**
  - Poner un nuevo dato en la cola.
- **xQueueReceive**
  - Sacar un dato de la cola.
- **xQueuePeek**
  - Leer un dato de la cola sin sacarlo.
- **uxQueueSpacesAvailable**
  - Obtener la cantidad de lugares libres en la cola.

+info: http://www.freertos.org/a00018.html

Ejemplos

- Creación y uso de colas.

- ¿Pueden usarse colas para *sincronizar* tareas?

Semáforos

- Dado que estamos en un contexto de ejecución multitarea, es necesario controlar el acceso concurrente a recursos, es decir que más de una tarea puede intentar utilizar la misma variable, cola, periférico, etc.

- Los semáforos se suelen utilizar para *sincronizar* el acceso a un recurso y *determinar* qué cantidad de tareas pueden hacer uso del mismo en forma concurrente.

Semáforos binarios

- Un semáforo binario puede ser “tomado” (“taken”) por una única tarea. Si otra tarea lo hace, quedará bloqueada hasta que la tarea que lo tomó anteriormente (“owner”) lo “libere” (“give”).

- Para evitar accesos concurrentes haremos:

```
xSemaphoreTake(sem, timeout);
/* acceso al recurso aquí */
xSemaphoreGive(sem);
```

Semáforos binarios

- Los semáforos binarios también se pueden usar para sincronización.

- Una tarea hace “take” y se bloquea, pero el “give” es llamado por otra tarea (o rutina de atención a interrupción), que controlará el momento en que la primera retome su ejecución. Esto es útil para implementar *programación orientada a eventos*.

Semáforos contadores

- Los semáforos contadores se diferencian de los binarios en que más de una tarea puede hacer “take” antes de bloquearse.

- El semáforo contador puede pensarse como un contador hasta **N** o una cola llena con **N** elementos.

Semáforos contadores

- Cada “take” decrementa en 1 el contador, o saca un elemento de la cola.
- Cada “give” incrementa el contador, o pone un elemento en la cola.
- Si una tarea hace “take” y el contador vale cero (o la cola está vacía), *la tarea se bloquea*.

Semáforos contadores

- El semáforo contador es útil cuando un recurso admite que más de una tarea lo use en forma simultánea.

- Otra aplicación es el “conteo de eventos” (“event counting”). Puedo tener varias instancias de una tarea que son capaces de procesar un determinado evento. Cada vez que el evento llega se libera el semáforo, y las tareas procesan cada llegada, *pudiendo ocurrir varios procesamientos en paralelo*.

Mutexes

- El mutex es equivalente a un semáforo binario, aunque incluye un mecanismo de “priority inheritance” para evitar inversión de prioridades. Los mutexes se utilizan exclusivamente para evitar accesos concurrentes a un recurso (de ahí la expresión “mutual exclusion”), es decir que las operaciones “take” y “give” suelen llevarse a cabo en la misma tarea, mientras que los semáforos binarios suelen usarse para sincronización (“take” y “give” en diferentes tareas).

Mutexes recursivos

- El mutex recursivo puede ser tomado (“taken”) repetidamente por una tarea, pero no estará liberado nuevamente hasta que se haga la misma cantidad de “gives”. Incluye el mismo mecanismo de herencia de prioridad que los mutexes normales, pero los mutex recursivos no pueden utilizarse en ISRs.
- A su vez tienen API dedicada:

```
xSemaphoreTakeRecursive(mutexr, timeout);
xSemaphoreGiveRecursive(mutexr);
```

Semáforos y Mutexes

- xSemaphoreCreateBinary
- xSemaphoreCreateCounting
- xSemaphoreCreateMutex
- xSemaphoreCreateRecursiveMutex
- vSemaphoreDelete
- xSemaphoreTake / xSemaphoreTakeFromISR
- xSemaphoreTakeRecursive
- xSemaphoreGive / xSemaphoreGiveFromISR
- xSemaphoreGiveRecursive

+info: http://www.freertos.org/a00113.html

Manejo de IRQs

- Si se necesita usar la API de FreeRTOS en cualquier ISR, es necesario utilizar las funciones que terminan en ...FromISR. El prototipo de esas funciones es similar a sus contrapartes sin FromISR, salvo que agregan otro parámetro que debe pasarse por referencia: `pxHigherPriorityTaskWoken` o `xSwitchRequired`.
- Las funciones ...FromISR modifican esta variable y la setean en **!= 0** si la API invocada desbloqueó a una tarea de mayor prioridad que la que fue interrumpida.

Manejo de IRQs

```
void IRQHandler(void)
{
   portBASE_TYPE xSwitchRequired;

   /* start irq handling */

   /* ... */
   xQueueSendFromISR(queue, data,
                     &xSwitchRequired);
   /* ... */

   /* end irq handling */

   portEND_SWITCHING_ISR(xSwitchRequired);
}
```

Manejo de IRQs

- La macro **`portEND_SWITCHING_ISR`**`(x);` equivale a:

```
if( x )
{
  portNVIC_INT_CTRL_REG =
            portNVIC_PENDSVSET_BIT;
}
```

Es decir, implica una llamada explícita al scheduler. Esta macro depende de la arquitectura: en **Cortex-M** el cambio de contexto se realiza con la excepción **PendSV**.

¿Preguntas?