

Departamento de Arquitectura y Tecnología de Computadores



UNIVERSIDAD DE SEVILLA



E.T.S. de Ingeniería Informática
Avda. Reina Mercedes, S/N.
41012 Sevilla, SPAIN



Escuela Universitaria Politécnica
C/ Virgen de África, 7.
41011 Sevilla, SPAIN

Análisis de núcleos en tiempo real

SETR en nuestro dia a dia...

Aerospace

- Flight management systems
- Jet engine controls
- Weapons systems

Audio

- MP3 players
- Amplifiers and tuners

Automotive

- Antilock braking systems
- Climate control
- Engine controls
- Navigation systems (GPS)

Communications

- Routers
- Switches
- Cell phones

Computer peripherals

- Printers
- Scanners

Domestic

- Air conditioning units
- Thermostats
- White goods

Office automation

- FAX machines / copiers

Process control

- Chemical plants
- Factory automation
- Food processing

Robots

Video

- Broadcasting equipment
- HD Televisions

And many more

Tiempo real

- **Tiempo Real Duro:**

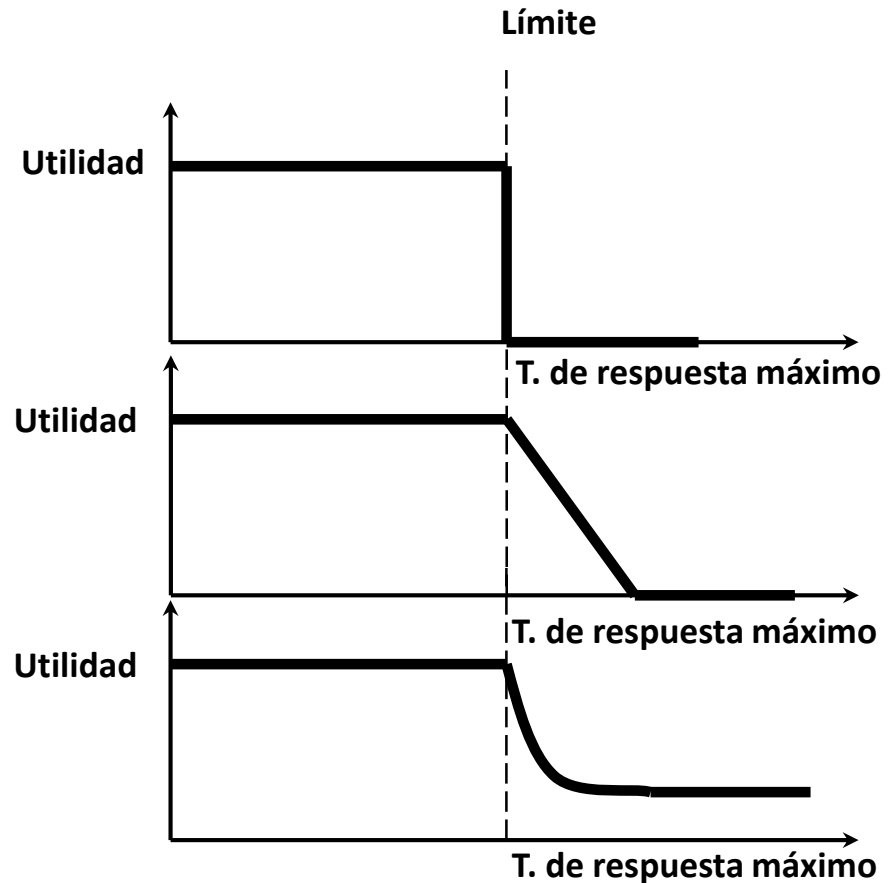
- El tiempo de respuesta es crítico, pasado un plazo, la respuesta pierde utilidad.

- Marcapasos.

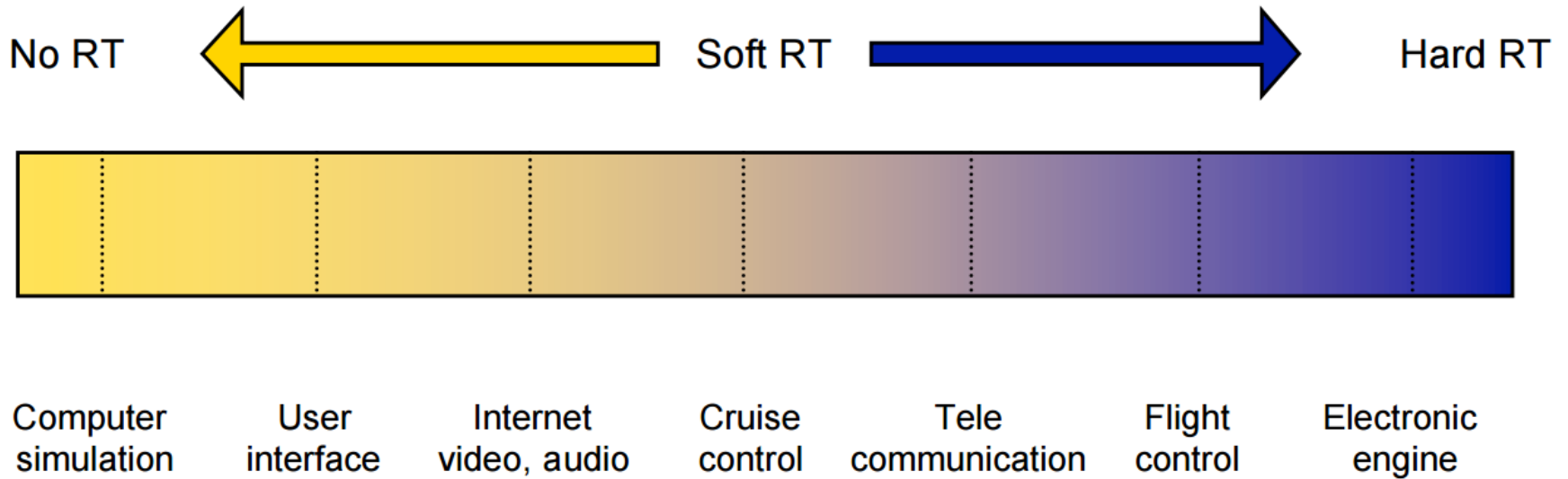
- **Tiempo Real Blando:**

- El retraso resta utilidad pero no es crítico, la respuesta sigue teniendo validez.

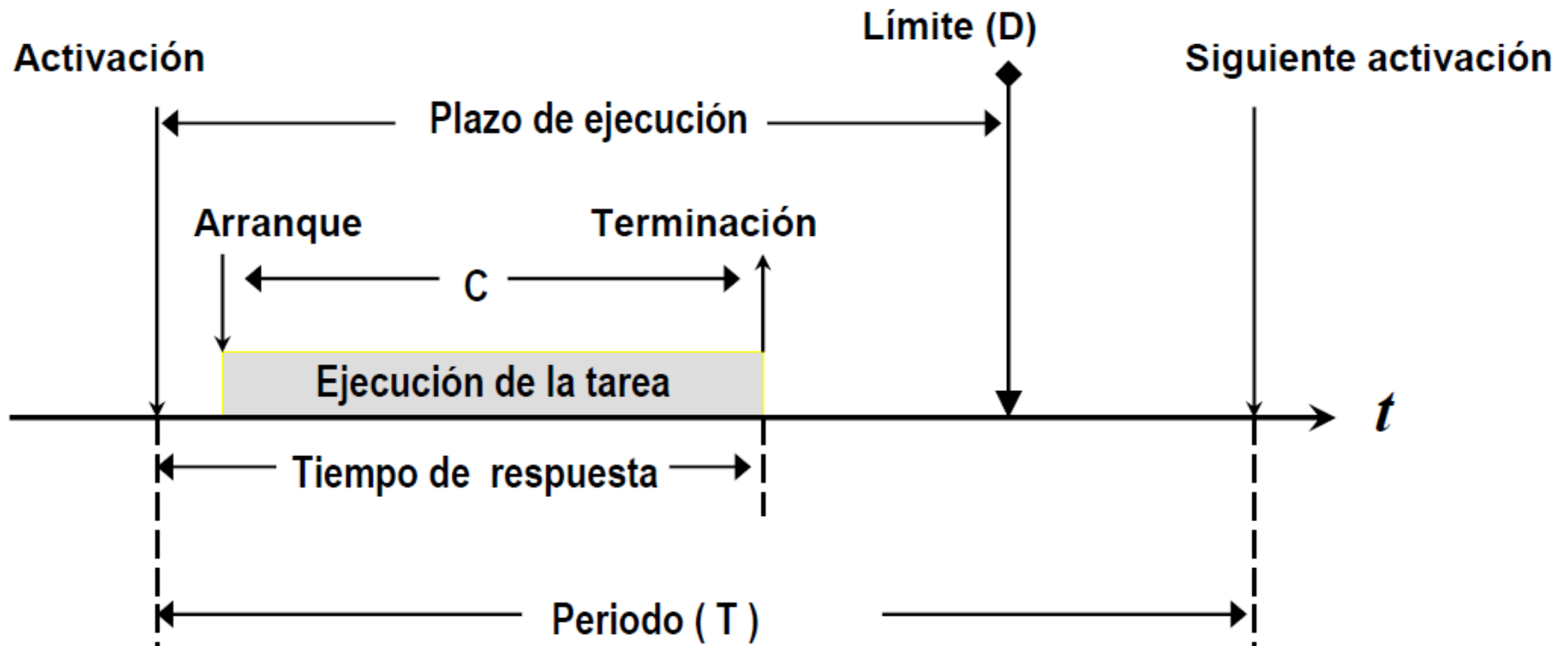
- Termómetro con display.



El espectro del tiempo real



Plazo de ejecución de una tarea



Modelos de programación

- 1- Procesamiento secuencial (bucle scan)

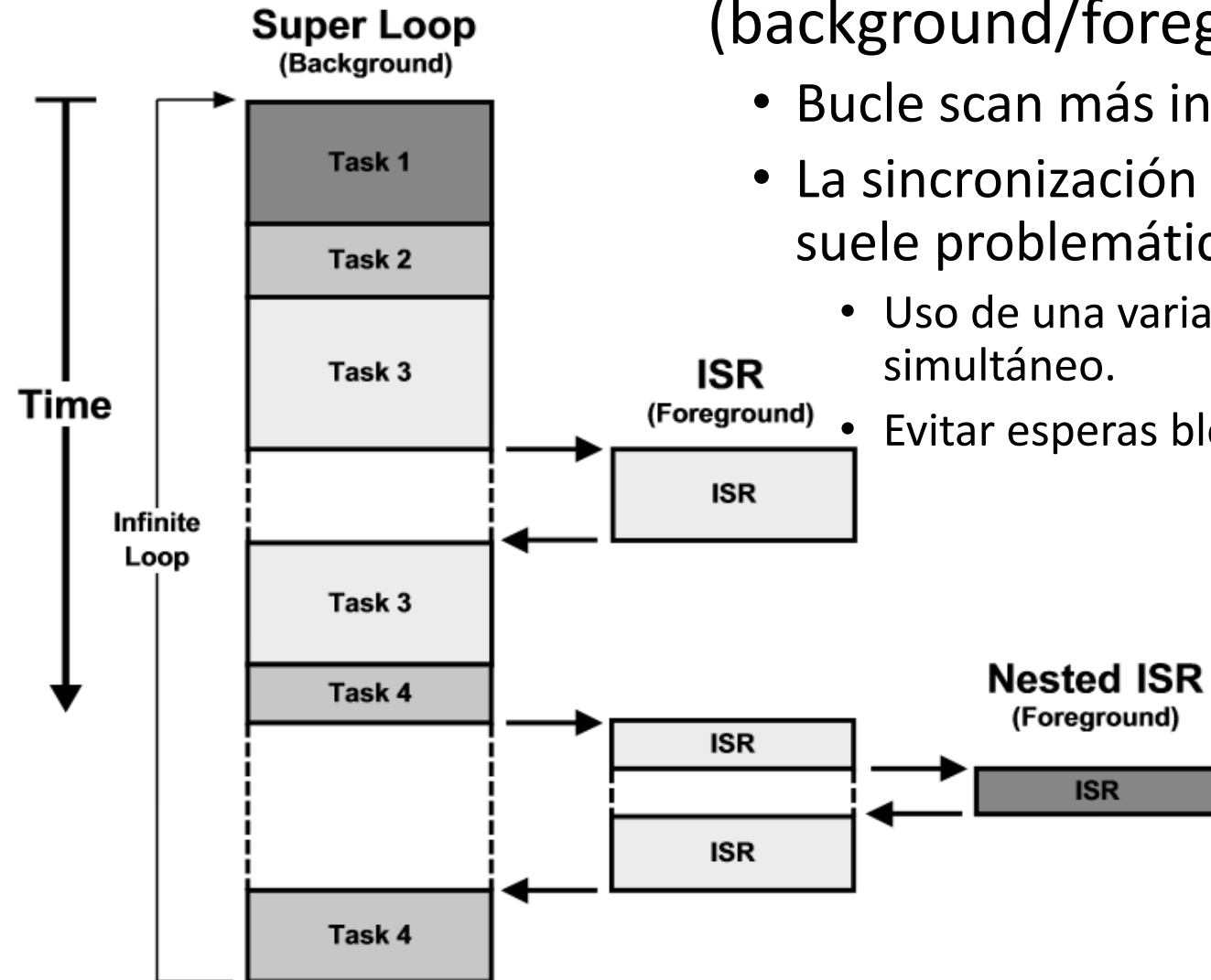
```
main()
{
/* Inicialización del sistema */
while(1){ /* Bucle infinito */
Tarea1();
Tarea2();
/* ... */
TareaN();
}
}
```

- Las tareas se ejecutan todas hasta el final antes de iniciar la siguiente (no bloqueantes). Se comparte la información mediante variables globales.
- Latencia relativamente grande, lo que dura el bucle.

Modelos de programación

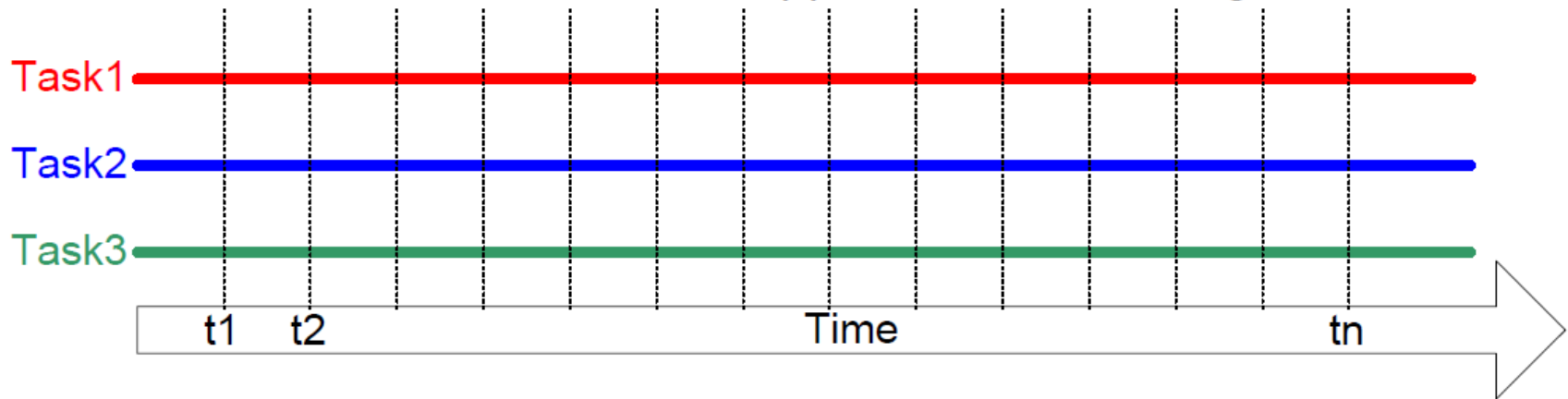
- 2- Primer plano / segundo plano (background/foreground):

- Bucle scan más interrupciones.
- La sincronización con interrupciones suele problemática:
 - Uso de una variables globales -> acceso simultáneo.
 - Evitar esperas bloqueantes.

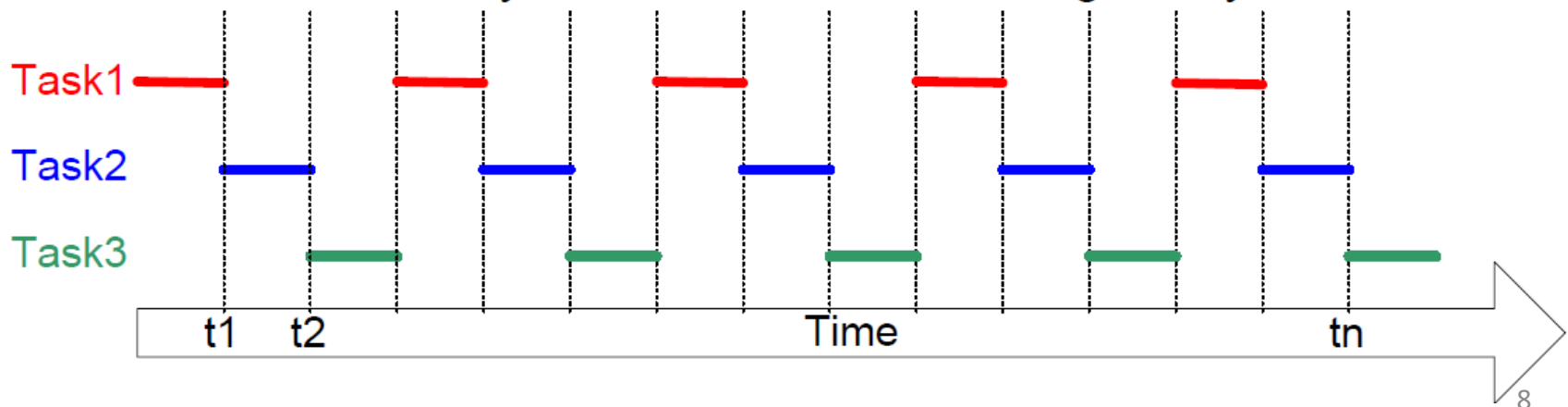


3 - Ejecución Multi-Tarea Genérica

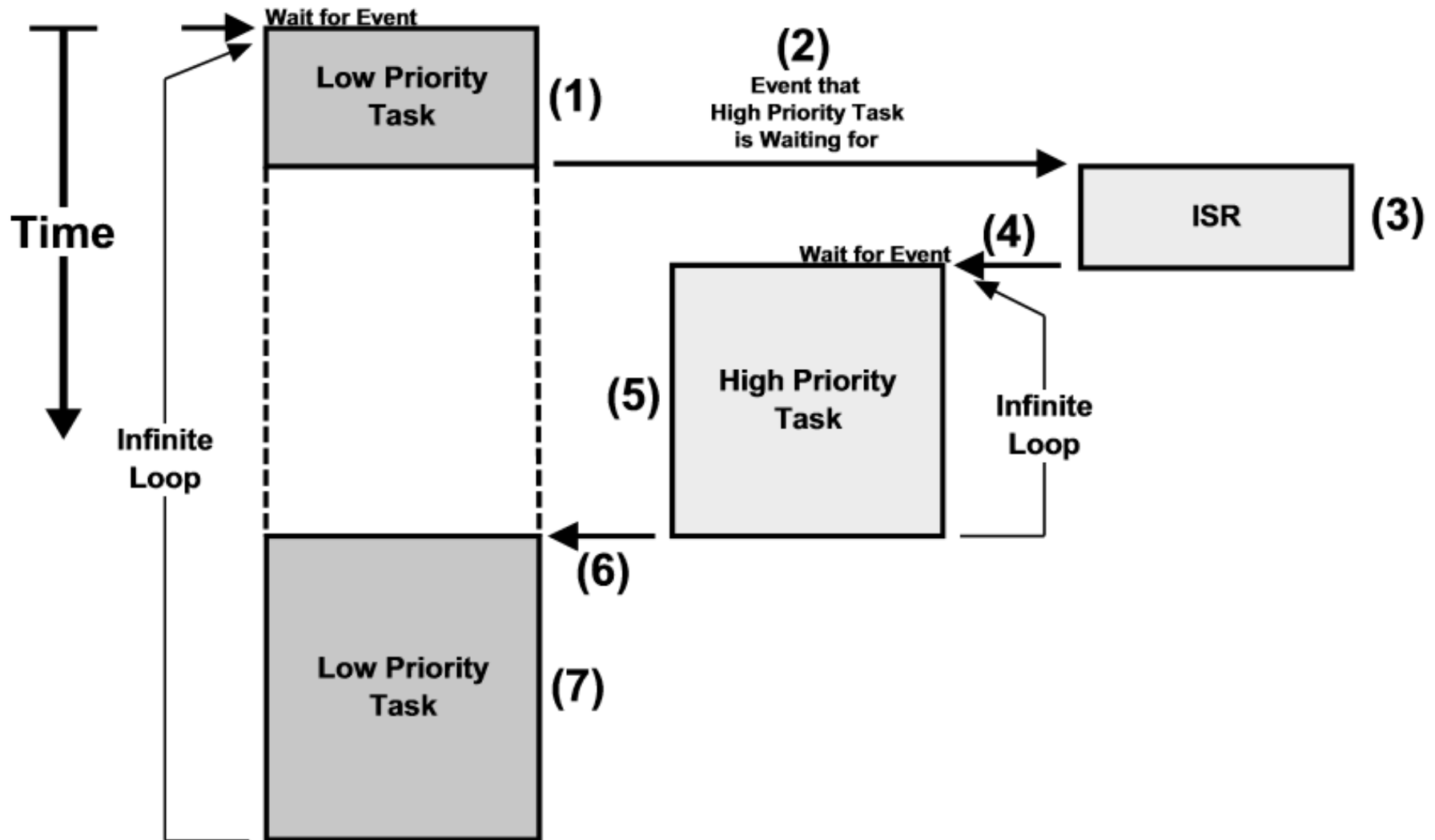
All available tasks appear to be executing ...



... but only one task is ever executing at any time.



3.1 - Ejecución multi-tarea apropiativa (prioridades)



Sistemas Operativos en Tiempo Real (SOTR – RTOS)

- Los RTOS proporcionan:
 - Ejecución **multitarea** basada en **prioridades**.
 - Planificador **apropiativo determinista**.
 - **Baja latencia** en la ejecución del **planificador**.
 - **Control de tiempos** de espera, **periodicidad** en la ejecución de tareas.
 - Soporte para **timers software** (funciones periódicas).
 - Soporte para la **ejecución concurrente** de tareas:
 - Sincronización / Comunicación entre tareas.
 - Acceso a secciones críticas (**tiempo crítico de ejecución**)
 - Control de uso de recursos compartidos.
 - Sincronización / Comunicación entre interrupciones y tareas.
 - **Gestión dinámica de memoria**.

Planificación apropiativa

- El planificador es el **corazón** del RTOS:
 1. Determina **que tarea se tiene que ejecutar** en cada momento.
 2. Tiene la capacidad de **quitar una tarea en ejecución**, sin que esta llame a una función bloqueante.
- Está implementado en **código ensamblador (uso instrucciones específicas de apoyo al RTOS)**, ha de ser **muy eficiente**, y se ejecuta:
 - **Tras un tiempo fijo**, usando timers especiales (SysTick).
 - Cuando una tarea llama a **una función del RTOS** se ha de replanificar:
 1. **Bloqueo de la tarea en ejecución**: espera por una bandera.
 2. **Ejecución de una tarea bloqueada**: liberación de un semáforo.

Time Driven Kernels

- En general la mayoría de Kernels del RTOS son por defecto **“time driven”**: Necesitan **una interrupción periódica**.
- Muchos procesadores poseen una **interrupción dedicada** (SysTick), que indica **la frecuencia del RTOS**: Tick del sistema.
- **Frecuencia configurable** en los ficheros de configuración del RTOS.
- **Normalmente** el tick del sistema se configura entre **1ms y 100ms**:
 1. Si alta frecuencia se dedica mucho tiempo de computación a la ejecución del planificador, pero tenemos una gran precisión temporal.
 2. Si baja frecuencia, el procesador tiene menos carga computacional, pero la granularidad temporal es mayor.
- La interrupción periódica **incrementa el tiempo del sistema**.

RTOS más extendidos

- **FreeRTOS:**
 - **Código abierto y gratuito**, versiones certificadas de pago.
 - Poca documentación abierta, documentación comercial.
 - **Fácil de usar.**
- **Micrium – uCOS-III:**
 - Código abierto, pero **sólo gratuito para aplicaciones no lucrativas.**
 - Requiere licencia para aplicaciones comerciales, **aprox. 20k€.**
 - Muy bien documentado: **Labrosse.**
 - Certificado para aplicaciones médicas, aeroespaciales y automotivas.
 - Relativamente complejo de usar.
 - Versiones dedicadas con soporte TCP/IP, USB e interfaces gráficas.
- **CooCox - CoOs:**
 - Código abierto, muy parecido al uCOS, y gratuito.
 - Documentación abierta y aceptable, fácil apoyarse en el Labrosse.
 - Fácil de usar.



CoOs

- El CoOs es un RTOS para los Cortex.
- Es gratuito, abierto y de libre distribución.
- Soporta diversos Cortex: STM32, SAM3, LPC, LM3S, etc...
- Tiene un **bajo** consumo de memoria.

Table 1.1.2 Space Specifications

Description	Space
RAM Space for Kernel	168 Bytes
Code Space for Kernel	974 Bytes
RAM Space for a Task	TaskStackSize + 24 Bytes(MIN) TaskStackSize + 48 Bytes(MAX)
RAM Space for a Mailbox	16 Bytes
RAM Space for a Semaphore	16 Bytes
RAM Space for a Queue	32 Bytes
RAM Space for a Mutex	8 Bytes
RAM Space for a User Timer	24 Bytes

Características CoOs

- Posee un planificador apropiativo, el cual permite la creación de **un número ilimitado de tareas**.
- Soporta **múltiples tareas con la misma prioridad** gracias a un algoritmo de round-robin.
- Implementa un módulo para la **gestión dinámica de memoria**.
- **Gestión del tiempo:**
 - Retrasos temporales.
 - Timers software.
- **Sincronismo entre tareas y recursos compartidos:**
 - Banderas.
 - Semáforos.
 - Mutex.
- **Comunicación entre tareas:**
 - Mailboxes.
 - Colas de mensajes.

Tareas

- Una tarea implementa comúnmente un bucle infinito que ejecuta una sección de código, y a continuación se bloquea en espera de algún evento del sistema operativo.
- Al bloquearse, la tarea devuelve el control al planificador del CoOs.
- El planificador conmuta a la tarea lista para ejecutarse con mayor prioridad.
- En caso de existir una tarea lista con mayor prioridad, pasará a ejecutarse.
- En caso de que existan tareas listas con la misma prioridad, se ejecutarán de una en una durante un tick del CoOs, rotando su ejecución mediante un algoritmo de round-robin.
- **Los bloqueos se realizan ante la espera de:**
 - un semáforo.
 - una bandera.
 - acceso a una sección crítica.
 - la recepción de un mensaje.
- **Bloqueos temporales:**
 - Un número fijo de ticks del sistema operativo.
 - Un periodo de tiempo preciso (Horas, minutos, segundos y milisegundos).

Ejemplo de tareas

Tarea Periódica



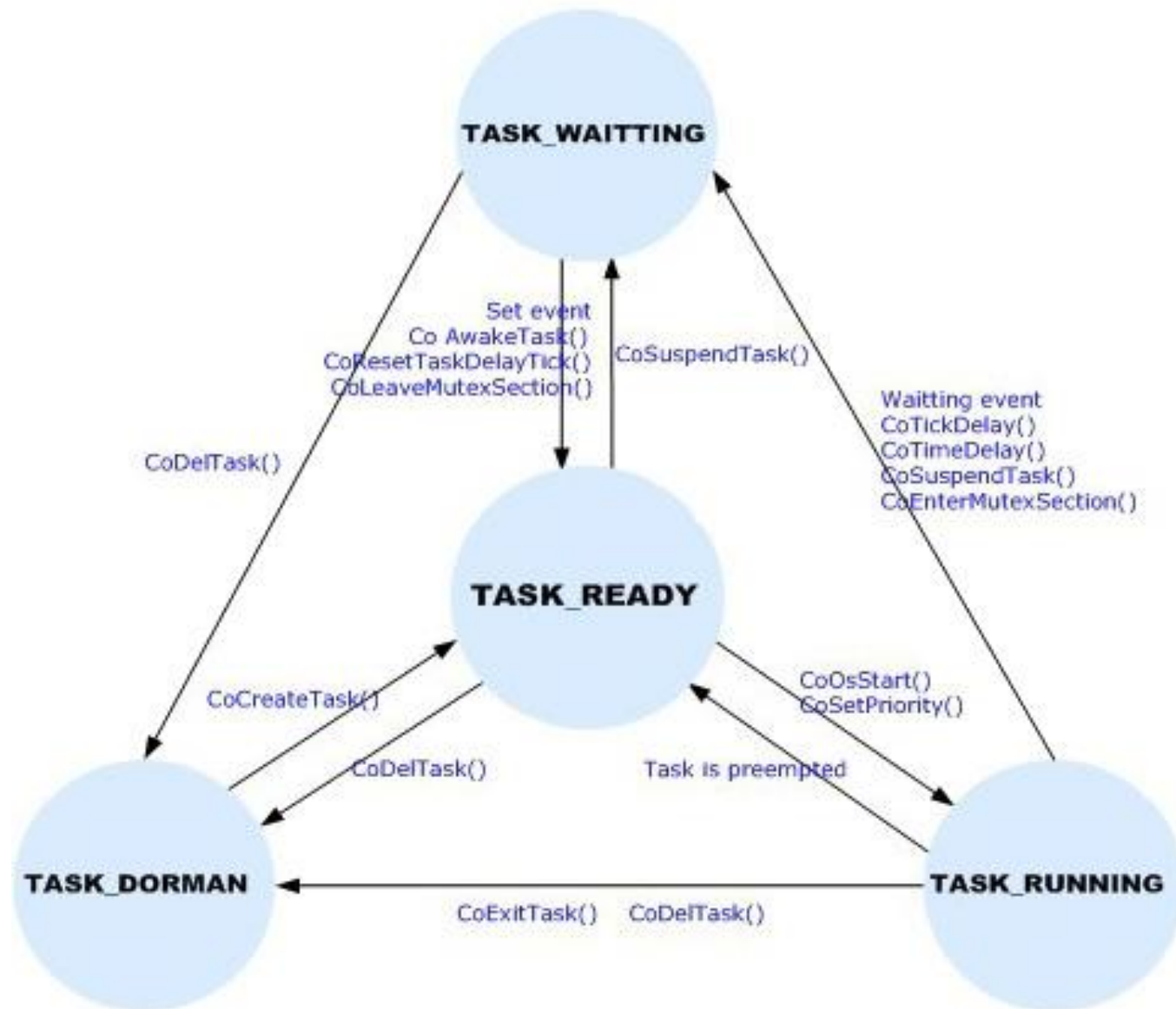
```
void miTarea (void * pdata){  
    //Variables privadas  
    int i;  
  
    //Inicialización de la tarea  
    i=0;  
  
    for(;;){    //Bucle infinito  
        //Cuerpo de la Tarea  
        i++;  
  
        //Bloqueo en espera de un evento del SSOC  
        CoTimeDelay(0,0,1,500); //Espera 1.5 segundos  
    }  
}
```

Tarea One-Shot

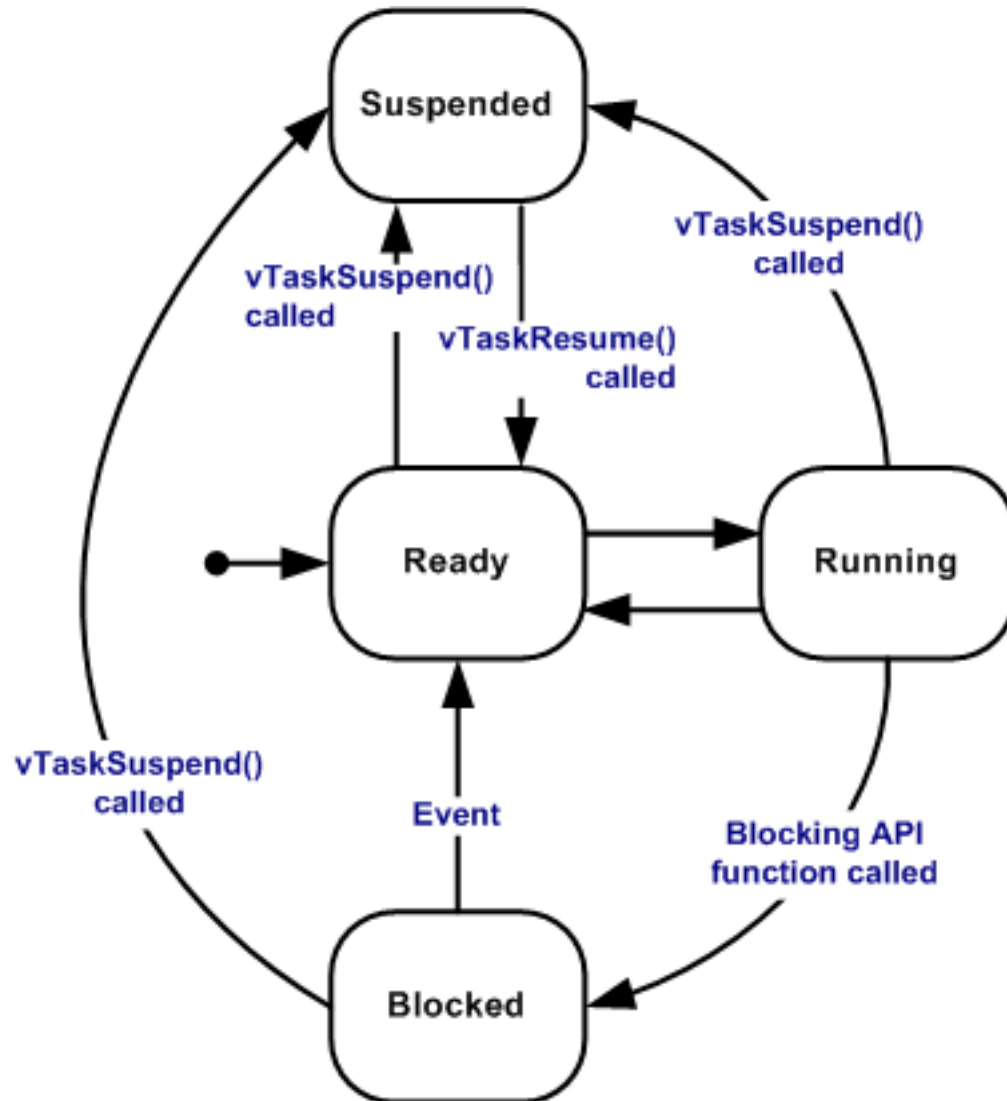


```
void miTarea (void * pdata){  
    //Variables privadas  
    int i;  
  
    //Inicialización de la tarea  
    i=0;  
  
    //Cuerpo de la Tarea  
    i++;  
  
    //Finalización de la tarea  
    CoExitTask();  
}
```

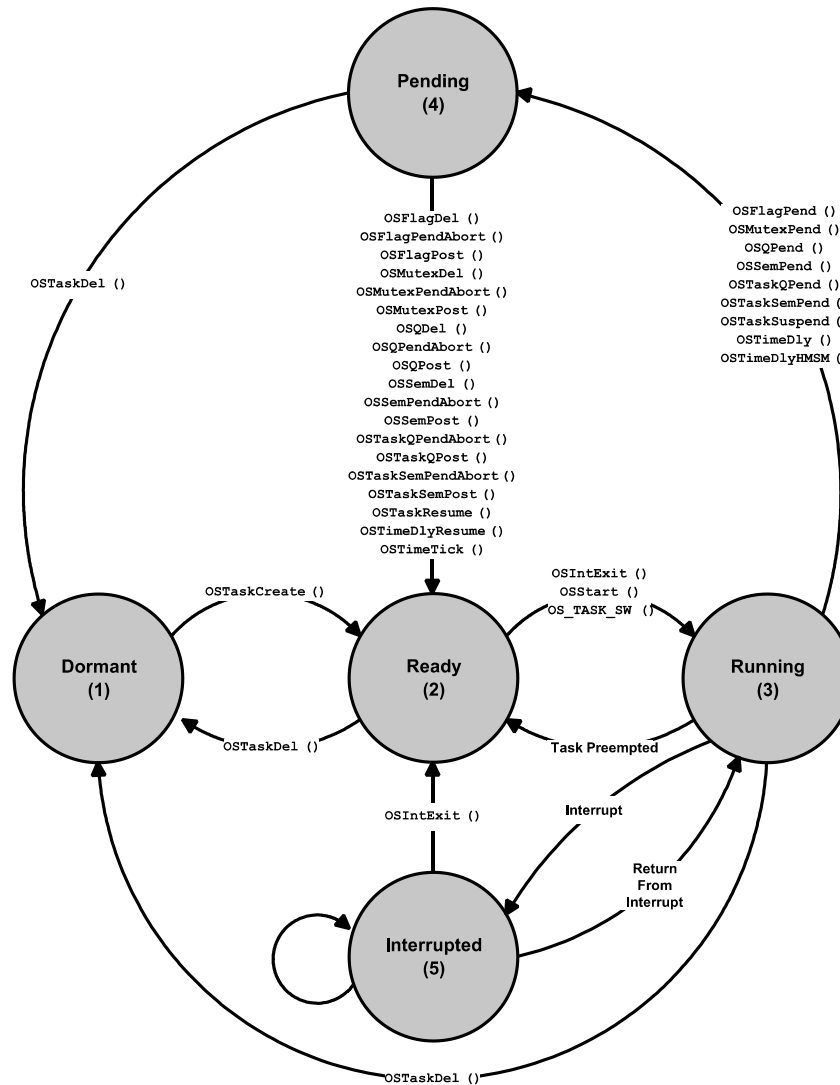
Estados de una tarea (CoOs)



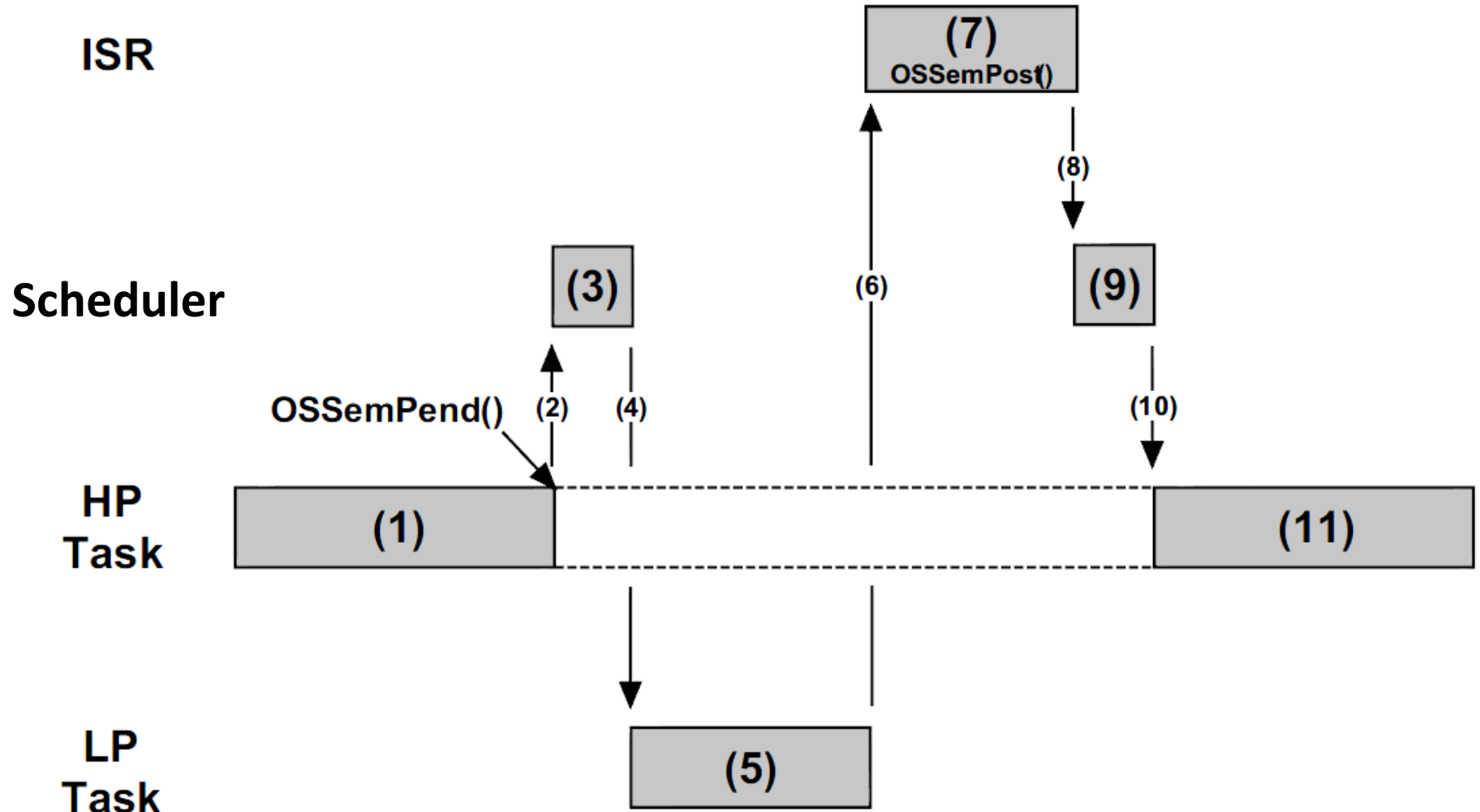
Estados de una tarea (FreeRTOS)



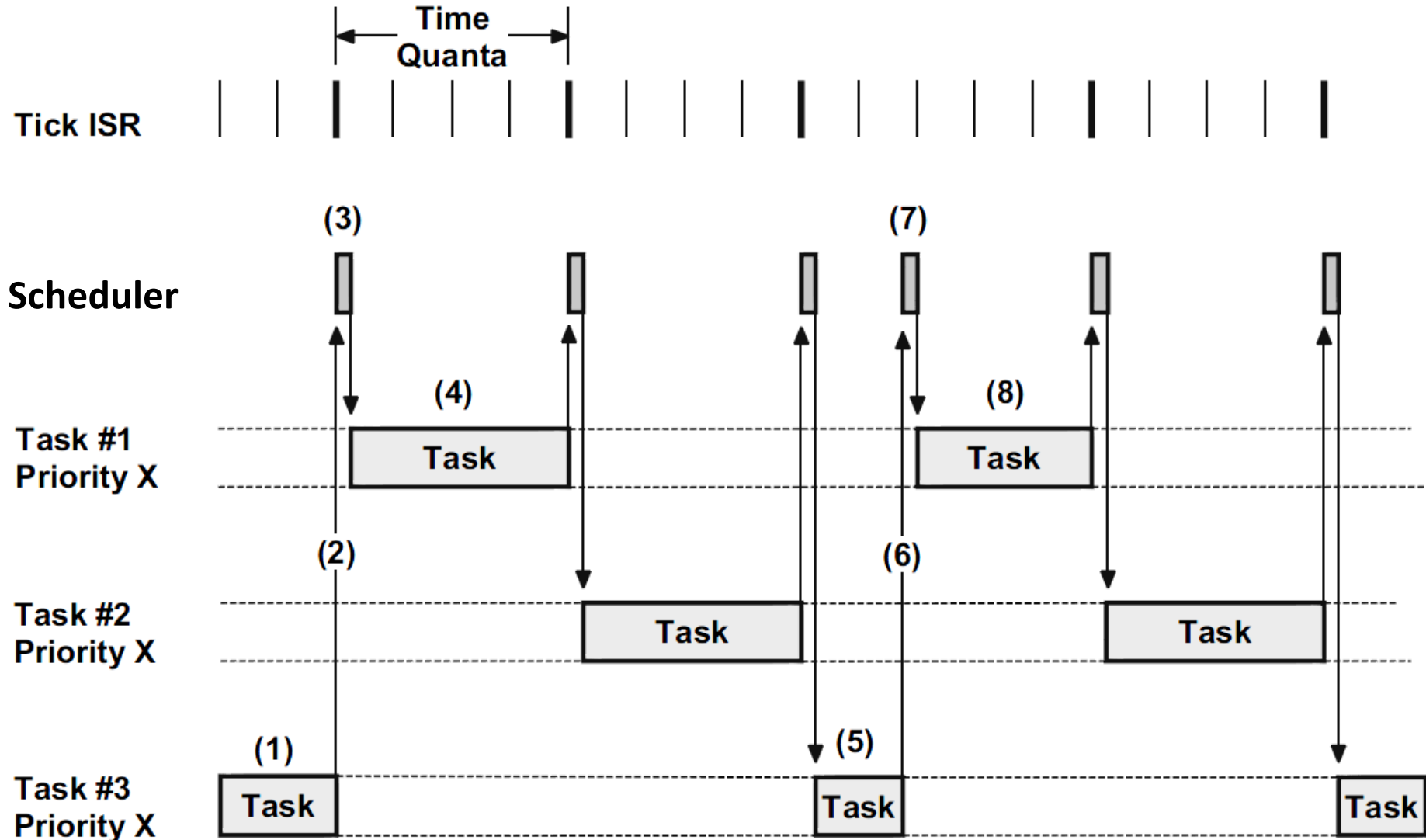
Estados de una tarea (uCOS)



Planificador Apropriativo

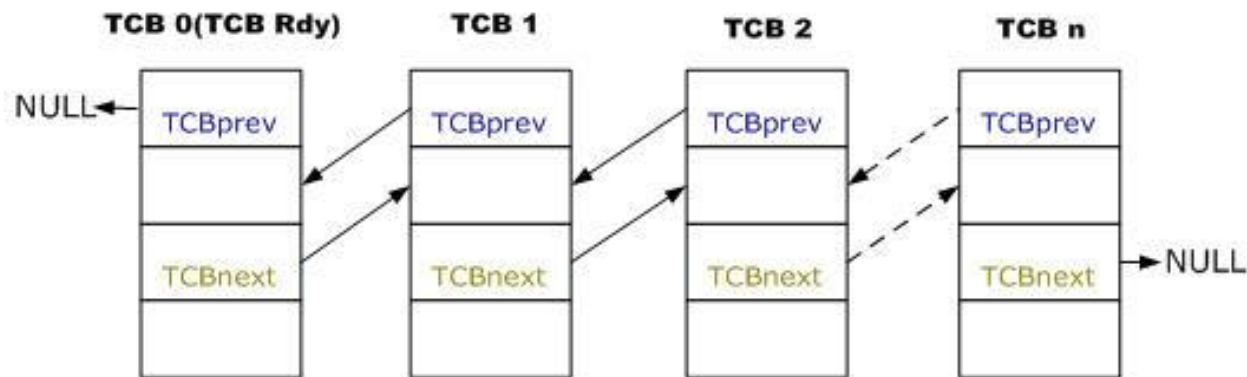


Tareas con la misma prioridad: Round-Robin Scheduling



Bloques de Control de Tareas (TCB)

- El **Task Control Block (TCB)** es una estructura de datos definida por el RTOS, en la que se agrupa toda la información necesaria para ejecutar una tarea:
 - Punteros al código/pila de la tarea.
 - Prioridad.
 - Estado de ejecución de la tarea.
 - Listas de elementos en espera.
 - Información estadística: uso de la pila, tiempo de CPU...
- El planificador implementa internamente **una lista enlazada de TCB**, esta lista **representa las tareas que se están ejecutando en el sistema**.
- Crear/destruir una tarea implica añadir/retirar un TCB de la lista.



Bloques de Control de Tareas (uCOS)

```

struct os_tcb {
    CPU_STK      *StkPtr,
    void          *ExtPtr,
    CPU_STK      *StkLimitPtr,
    OS_TCB        *NextPtr,
    OS_TCB        *PrevPtr,
    OS_TCB        *TickNextPtr,
    OS_TCB        *TickPrevPtr,
    OS_TICK_SPOKE *TickspokePtr,
    OS_CHAR       *NamePtr,
    CPU_STK      *StkBasePtr,
    OS_TASK_PTR   TaskEntryAddr,
    void          *TaskEntryArg,
    OS_PEND_DATA  *PendDataTblPtr,
    OS_STATE      PendOn,
    OS_STATUS     PendStatus,
    OS_STATE      TaskState,
    OS_PRIO       Prio,
    CPU_STK_SIZE  StkSize,
    OS_OPT        Opt,
    OS_OBJ_QTY    PendDataEntries,
    CPU_TS        TS,
    OS_SEM_CTR    SemCtr,
    OS_TICK       TickCtrPrev,
    OS_TICK       TickCtrMatch,
    OS_TICK       TickRemain,
    OS_TICK       TimeQuanta,
    OS_TICK       TimeQuantaCtr,
    void          *MsgPtr,
    OS_MSG_SIZE   MsgSize,
    OS_MSG_Q      MsgQ,
    CPU_TS        MsgQPendTime,
    CPU_TS        MsgQPendTimeMax,
    OS_REG        RegTbl[OS_TASK_REG_TBL_SIZE],
    OS_FLAGS      FlagsPend,
    OS_FLAGS      FlagsRdy,
    OS_OPT        FlagsOpt,

```

```

    OS_NESTING_CTR    SuspendCtr,
    OS_CPU_USAGE     CPUUsage,
    OS_CTX_SW_CTR    CtxSwCtr,
    CPU_TS           CyclesDelta,
    CPU_TS           CyclesStart,
    OS_CYCLES        CyclesTotal,
    OS_CYCLES        CyclesTotalPrev,
    CPU_TS           SemPendTime,
    CPU_TS           SemPendTimeMax,
    CPU_STK_SIZE     StkUsed,
    CPU_STK_SIZE     StkFree,
    CPU_TS           IntDisTimeMax,
    CPU              SchedLockTimeMax,
    OS_TCB           DbgNextPtr,
    OS_TCB           DbgPrevPtr,
    CPU_CHAR         DbgNamePtr,
},

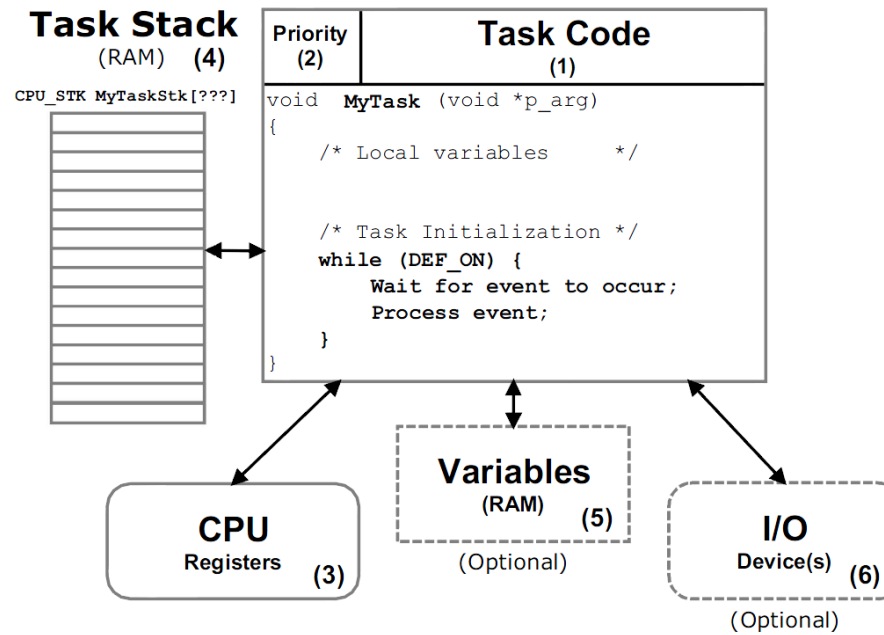
```


Bloques de Control de Tareas

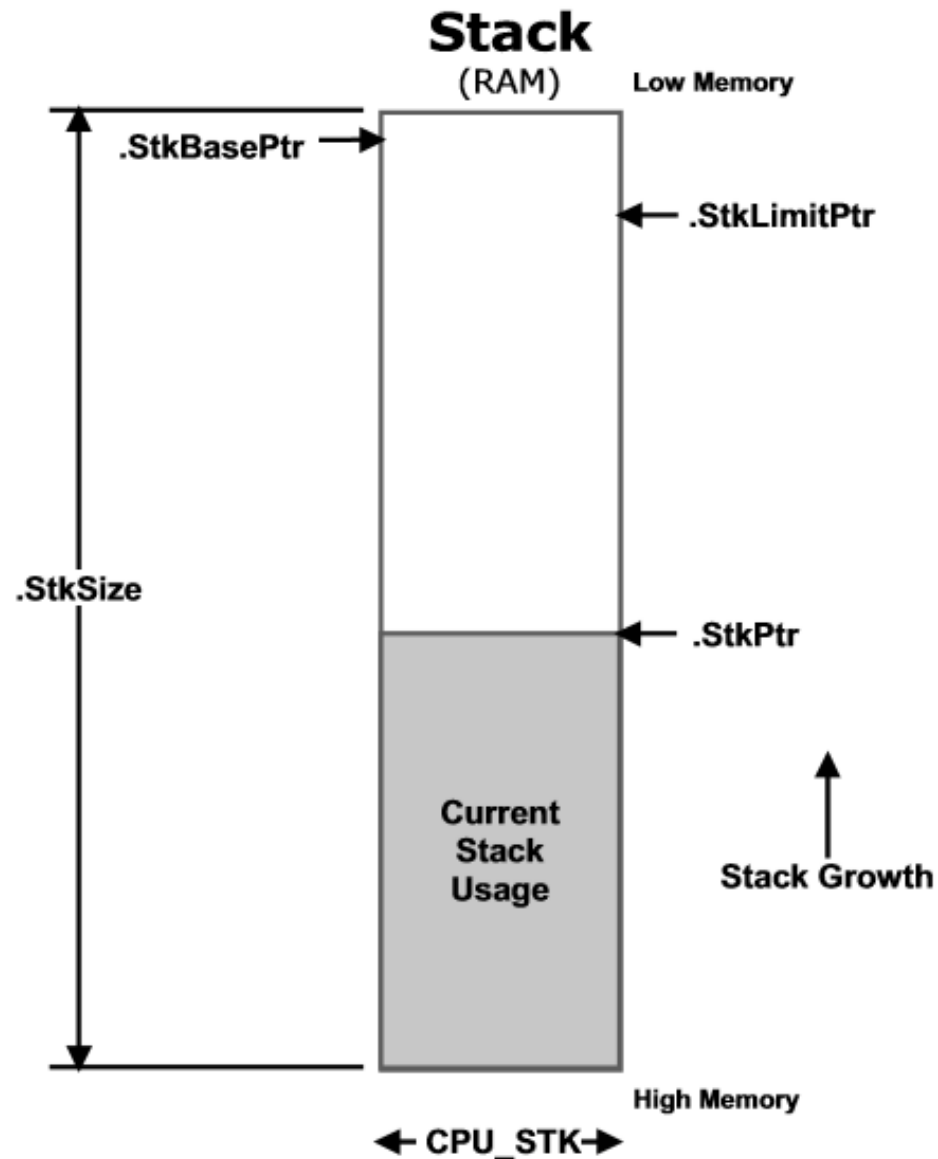
- Algunos campos destacables:
 - **TaskState**: estado de la tarea.
 - **Prio**: prioridad de la tarea.
 - **StkBasePtr**: Puntero al origen de la pila
 - **StkPtr**: Puntero a la cima de la pila.
 - **StkLimitPtr**: Puntero al límite de la pila.
 - **NextPtr** y **PrevPtr**: punteros a otros TCBs, implementación de la lista enlazada.
 - **TaskEntryAddr**: puntero al origen del código que implementa la tarea.
 - **TaskEntryArg**: puntero a los argumentos de entrada de la tarea.
 - **PendDataTblPtr**: puntero a un vector donde se encuentra los elementos por los que está bloqueada la tarea.
 - **PendTime**: tiempo que resta de bloqueo de la tarea, se usa tanto para esperas temporales, como time-outs.

Espacio de memoria de las tareas

- **Pila (Stack): Espacio consecutivo de memoria (vector).**
- **Cada tarea debe poseer su propia pila en exclusiva.**
 - El código de la tarea puede invocarse tantas veces como se necesite.
 - Se necesita un espacio de memoria exclusivo por invocación.
- **La pila contiene el contexto de la tarea.**
 - Se guardan todas las variables locales que se van invocando desde la tarea.
 - Árbol de llamadas a las funciones locales: parámetros, valores devueltos y direcciones de retorno.
 - Se almacena el contexto de la tarea cuando el planificador deja de ejecutarla.
- **Las variables globales se guardan en la memoria global.**



Ejemplo: Control de la pila



Funciones para la gestión de tareas

- Para la creación y gestión de las tareas, el CoOs nos proporciona una serie de funciones:
 - **CoCreateTask**: Crea una nueva tarea en un segmento de pila, y devuelve un manejador de la tarea.
 - **CoExitTask**: Finaliza la tarea en ejecución.
 - **CoDelTask**: Una tarea elimina a otra.
 - **CoSuspendTask**: Una tarea suspende a otra, o a si misma.
 - **CoAwakeTask**: Una tarea despierta a otra.

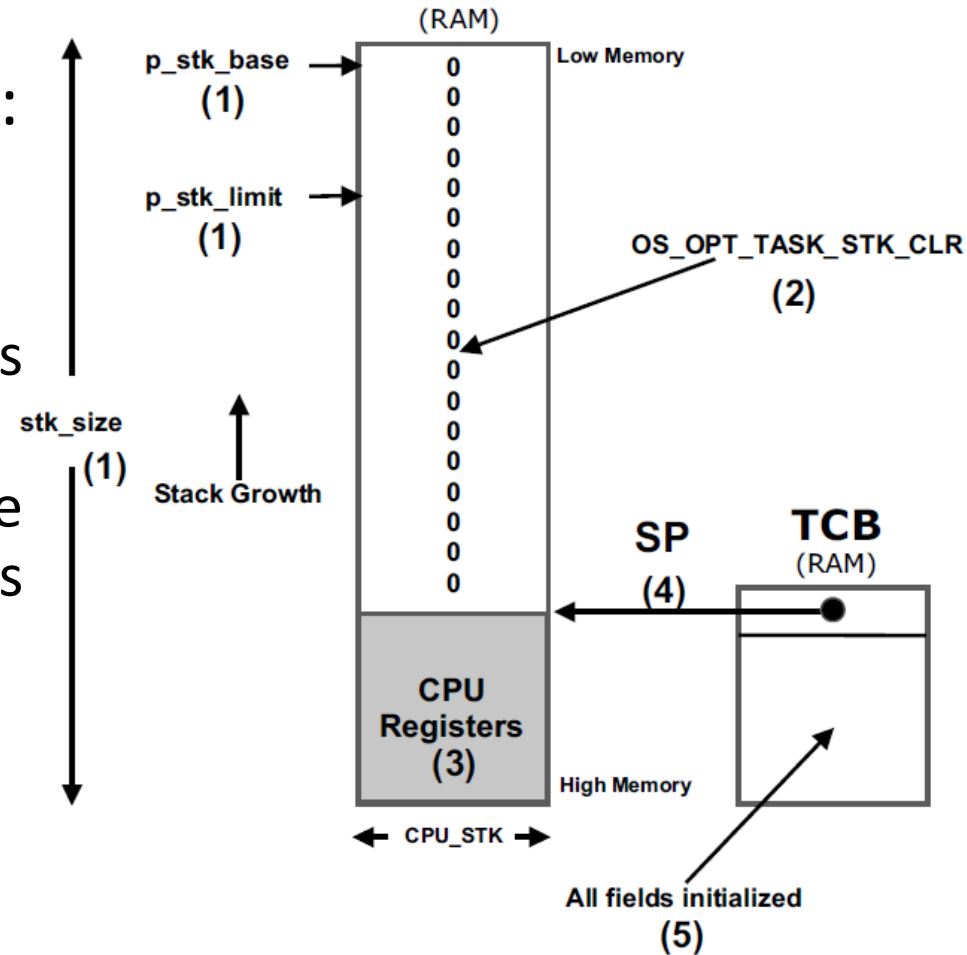
Implementación y creación de Tareas

- **Para la implementación de una tarea se necesita:**
 - La función que implementa la tarea.
 - Un vector de pila para alojar las variables de la tarea.
 - Un bloque control de tarea (TCB), interno al CoOs y que contiene toda la información relativa a la tarea.
- **Para crear una tarea llamamos a CoCreateTask, la cual recibe como parámetros:**
 1. Puntero a la función de la tarea.
 2. Puntero a los argumentos de la tarea.
 3. La prioridad de la tarea (cuando más alta, menos prioritaria).
 4. Puntero a la primera posición libre de la pila.
 5. Tamaño de la pila.
- **Devuelve un manejador de la tarea (opcional)**

Creando una tarea

- Tras llamar a **CoCreateTask**:

1. Se inicializa la pila con 0's.
2. Se crea un nuevo TCB.
3. Se le dan valores a todos los campos del TCB.
4. Se enlaza con en la lista de los TCBs de las tareas creadas anteriormente.



Ejemplo: Creando una tarea

Vector de pila
(64 elementos)

Cuerpo de la tarea

Creación de la tarea

```
OS_STK    pila[64];    //Pila de 64 elementos
```

```
//Conmuta el led especificado como argumento  
//al activarse la bandera flag
```

```
void miTarea (void * parg){
```

```
    int nLed;
```

```
    nLed=parg; //Obtiene el número del led a conmutar
```

```
    for(;;){
```

```
        LED_TOGGLE(nLed); //Conmuta el led
```

```
        CoPendFlag(flag,0); //Espera la bandera
```

```
    }
```

```
}
```

```
void main(void){
```

```
    // .
```

```
    // .
```

```
    // .
```

```
    //Lanza la tarea miTarea con el número 3 como argumento
```

```
    //Prioridad 1 y una pila de 64 elementos
```

```
    CoCreateTask ( miTarea , 3 , 1 , &pila[63] , 64 );
```

```
    // .
```

```
    // .
```

```
    // .
```

```
}
```

Ejemplo: Creación/Eliminación dinámica de tareas

```
#define STACK_SIZE_LCD 1024          /*!< Define "taskA" task size */
OS_STK LCD_stk[2][STACK_SIZE_LCD]; /*!< Define "taskA" task stack */

void CreateLCDTask(void){

    Init_AnalogJoy();

    LCD_Initialization();
    LCD_Clear(Blue);

    CoCreateTask (LCDManagerTask,0,1,&LCD_stk[0][STACK_SIZE_LCD-1],STACK_SIZE_LCD);
}

void LCDManagerTask (void* pdata) {

    OS_TID lcdId;

    for (;;) {

        lcdId = CoCreateTask (LCDHelloWorldTask,0,1,&LCD_stk[1][STACK_SIZE_LCD-1],STACK_SIZE_LCD);
        waitForKey(5,0);
        CoDelTask(lcdId);

        lcdId = CoCreateTask (LCDGradientTask,0,1,&LCD_stk[1][STACK_SIZE_LCD-1],STACK_SIZE_LCD);
        waitForKey(5,0);
        CoDelTask(lcdId);

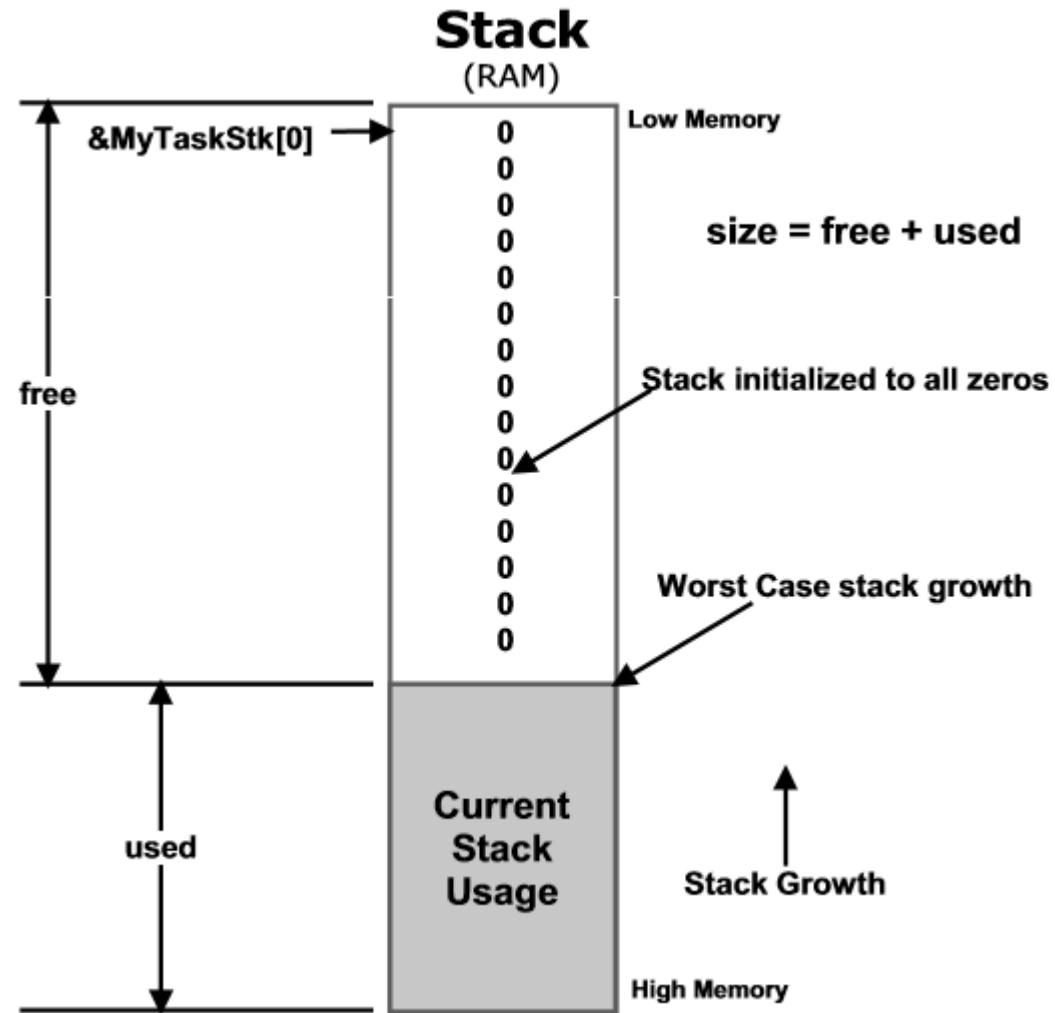
        lcdId = CoCreateTask (LCDDrawAreaTask,0,1,&LCD_stk[1][STACK_SIZE_LCD-1],STACK_SIZE_LCD);
        waitForKey(5,0);
        CoDelTask(lcdId);

        lcdId = CoCreateTask (LCDScopeTask,0,1,&LCD_stk[1][STACK_SIZE_LCD-1],STACK_SIZE_LCD);
        waitForKey(5,0);
        CoDelTask(lcdId);

    }
}
```


Determinar el tamaño de la pila

- Es **difícil** saber de manera **determinista** el tamaño de la pila.
- Algunos compiladores proporcionan herramientas para **simular** un árbol de llamadas y **determinar** la cantidad de memoria utilizada.
- **Solución:**
 - Mecanismo experimental



Detección del desborde de la pila

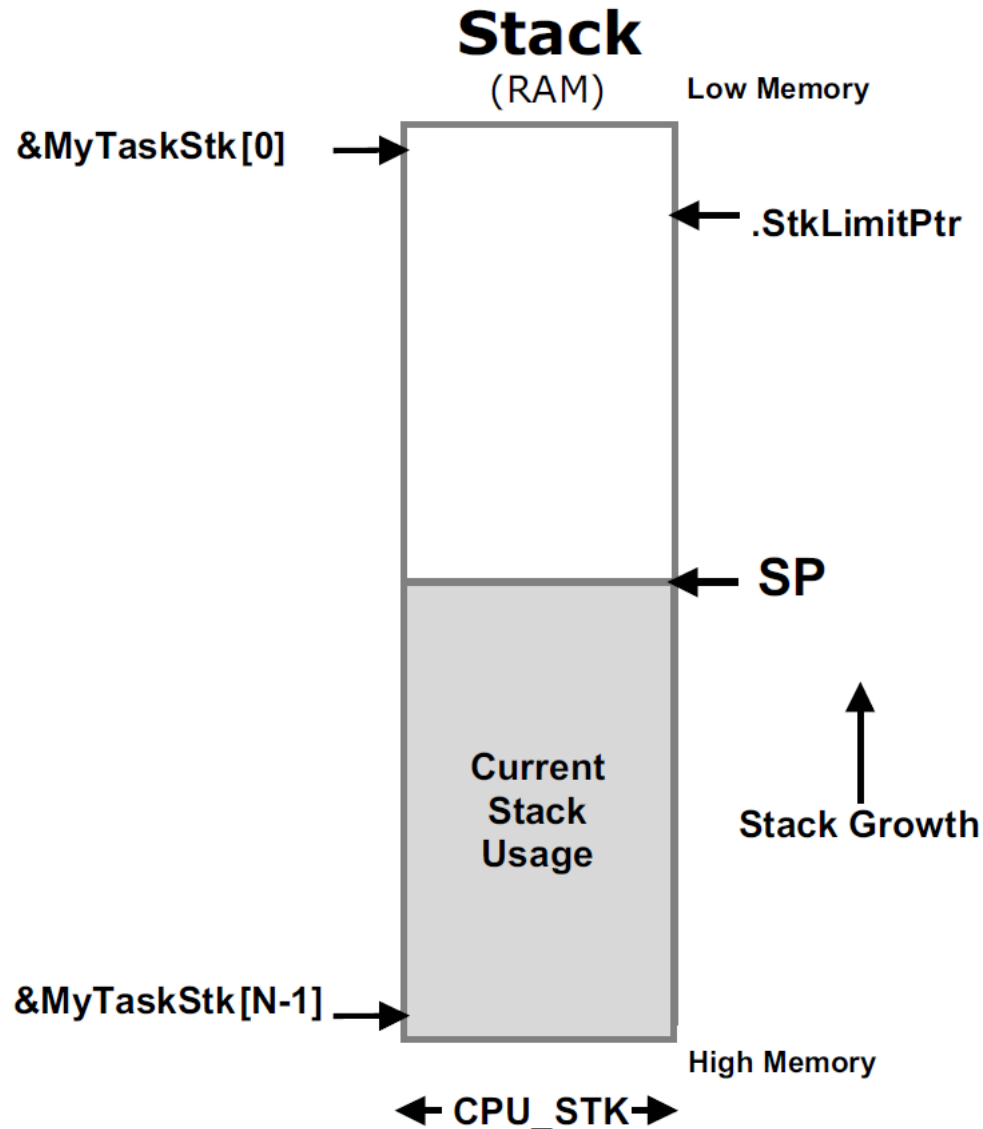
1. Usando una MMU o MPU:

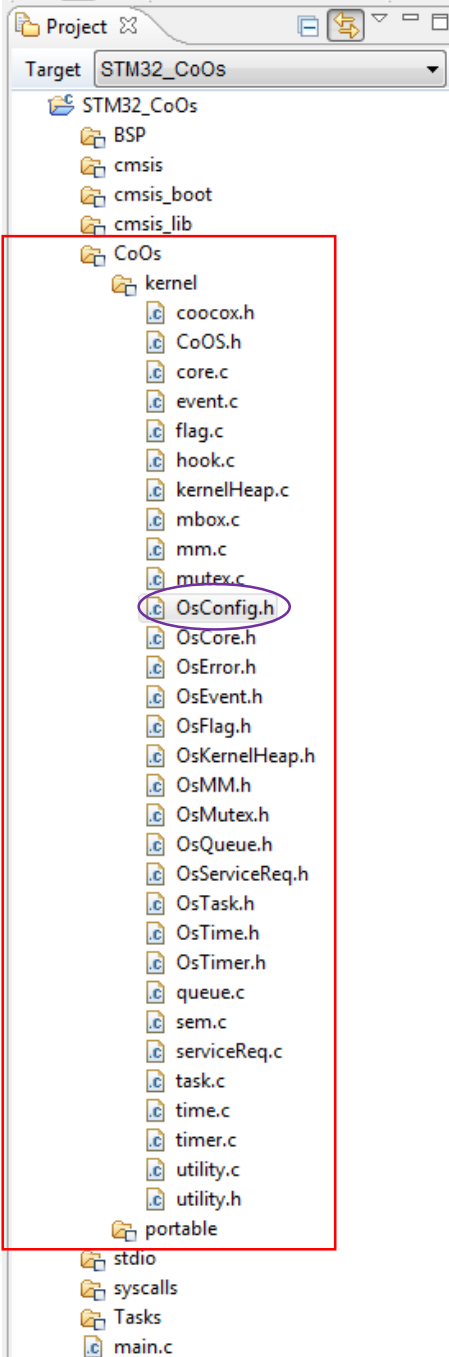
- MMU: Memory Manager Unit.
- MPU: Memory Protection Unit.
- Ambos son dispositivos hardware que observan los accesos a memoria por parte de la tareas.
- Detectan cuando una tarea intenta acceder más allá de su espacio de memoria o accesos a direcciones inválidas.
- Se dispara una excepción hardware para gestionar esta situación.
- Estos dispositivos están presentes en procesadores de gama alta: Cortex-A9, Cortex-Rx.

2. Detección del desborde mediante la software.

- El kernel es el encargado de vigilar el crecimiento de la pila.
- Realizar estas comprobaciones en cada acceso a memoria es muy ineficiente.
- Cada vez que se conmuta una nueva tarea, se comprueba si la pila de la anterior se ha desbordado.
- Se dispara una excepción software (llamada a una función especial del sistema operativo) para gestionar esta situación.
- Cuando se detecta ya es tarde: Invasión del espacio de memoria de otra tarea.
- Solución: Lanzar la excepción software ANTES de desbordar la pila. Reservar la pila con un margen de seguridad.

Detección del desborde de la pila





```
16
17
18 #ifndef _CONFIG_H
19 #define _CONFIG_H
20
21
22 /*!<
23 Defines chip type,cortex-m3(1),cortex-m0(2)
24 */
25 #define CFG_CHIP_TYPE (1)
26
27 /*!<
28 Defines the lowest priority that be assigned.
29 */
30 #define CFG_LOWEST_PRIO (64)
31
32 /*!<
33 Max number of tasks that can be running.
34 */
35 #define CFG_MAX_USER_TASKS (16)
36
37 /*!<
38 Idle task stack size(word).
39 */
40 #define CFG_IDLE_STACK_SIZE ((64))
41
42 /*!<
43 System frequency (Hz).
44 */
45 #define CFG_CPU_FREQ (168000000)
46
47 /*!<
48 systick frequency (Hz).
49 */
50 #define CFG_SYSTICK_FREQ (1000)
51
52 /*!<
53 max systerm api call num in ISR.
54 */
55 #define CFG_MAX_SERVICE_REQUEST (5)
56
57 /*!<
58 Enable(1) or disable(0) order list schedule.
59 If disable(0),CoOS use Binary-Scheduling Algorithm.
60 */
61 #if (CFG_MAX_USER_TASKS) <15
62 #define CFG_ORDER_LIST_SCHEDULE_EN (1)
63 #else
64 #define CFG_ORDER_LIST_SCHEDULE_EN (0)
65 #endif
66
67
68 /*!<
69 Enable(1) or disable(0) Round-Robin Task switching.
70 */
71 #define CFG_ROBIN_EN (1)
72
```

CoOs Kernel

Control del CoOs

- Funciones de inicialización y control del CoOs:
 - **ColnitOS(void):**
 - Inicializa las variables del sistema operativo.
 - **CoStartOS(void):**
 - Comienza la ejecución del planificador.
 - El CoOs toma el control de la ejecución
 - El código no continua con normalidad después de llamarlo.
 - **Deben existir tareas creadas antes de lanzarlo.**

Lanzando el CoOs

- Siempre se sigue el mismo proceso:
 1. Se inicializa el reloj del sistema.
 2. Se inicializan las estructuras internas del CoOs.
 3. Se crean los elementos de sincronismo y comunicaciones (banderas, colas...).
 4. Le crean las tareas de la aplicación.
 5. Se lanza el scheduler del CoOs.

```
int main(void)
{
    SystemInit();           //Inicialización del reloj
    CoInitOS ();            //Inicialización del CoOs
    CreateSystemObjetscs(); //Inicialización Sem, flags, queues...
    CreateUserTasks();      //Creación de Tareas
    CoStartOS ();           //Comienzo de ejecución del planificador
    while(1)                //La ejecución nunca debería llegar aquí
    {
    }
}
```

Lanzando tareas

- Las tareas se lanzan en dos pasos:
 1. Se crean los **elementos del CoOs compartidos entre tareas** (banderas, colas, semáforos...) para evitar que **una tarea acceda a un elemento antes de ser creado**.
 2. Se crean las tareas y quedan a la **espera de que se lance el planificador**.

```
void CreateSystemObjetscs(void){  
    //Inicialización de los elementos compartidos: flags, semaforos, colas...  
    CreateJoyFlags();           //Crear banderas del joystick  
    CreateSerialQueue();        //Crear cola para el puerto serie  
    CreateLCDSem();             //Crear Semaforo del LCD  
}  
  
void CreateUserTasks(void){  
    //Creación de las tareas de usuario  
    CreateJoyTask();  
    CreateLedTask();  
    CreatSerialTask();  
    CreateLCDTask();  
}
```

La tarea Idle

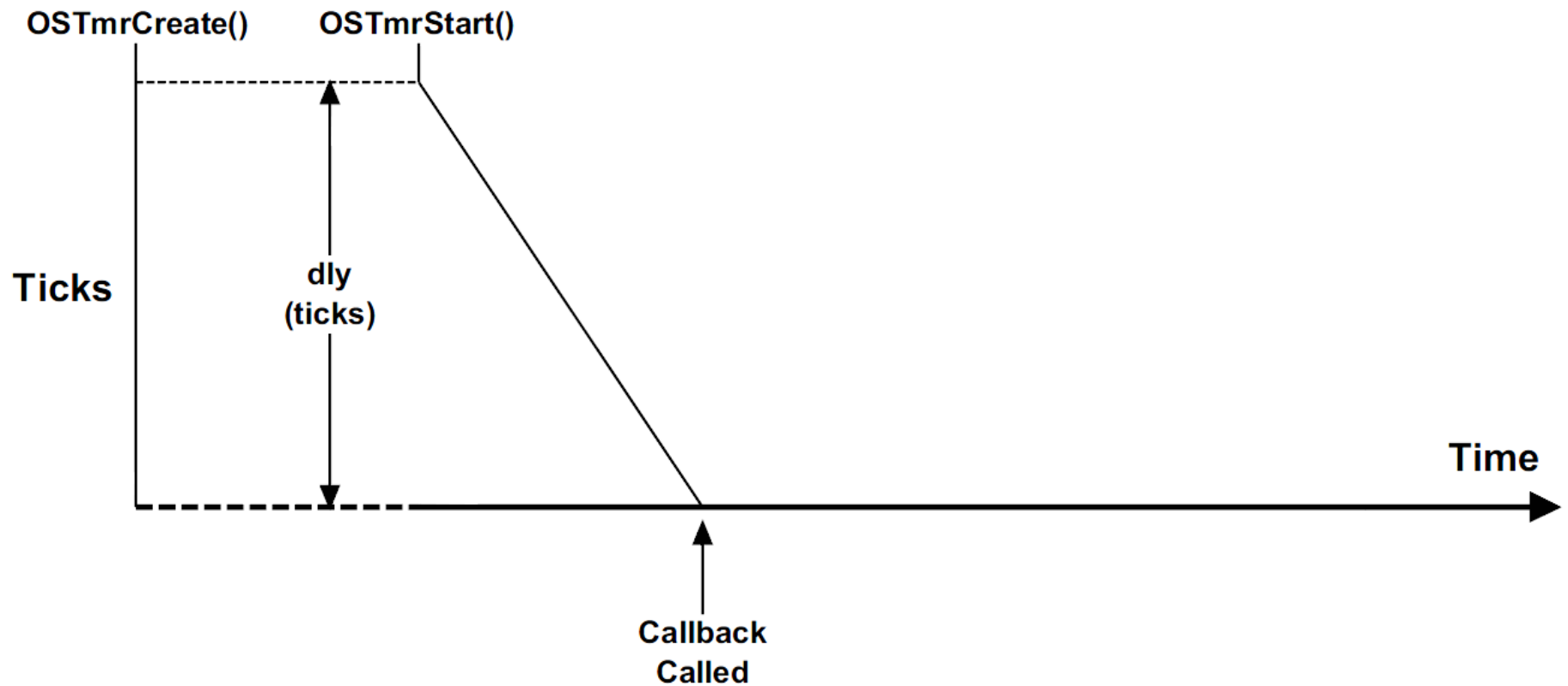
- CoOs lanza una tarea por defecto: **ColdleTask**
- Tiene la mínima prioridad y sustituye al bucle principal: **CFG_LOWEST_PRIO**
- Cuando todas las tareas están bloqueadas, la ejecuta a la espera de tener tareas listas para su ejecución.
- Se encuentra en **hook.c**

```
/**
*****
* @brief      IDLE task of OS
* @param[in]  pdata    The parameter passed to IDLE task.
* @param[out] None
* @retval     None
*
* @par Description
* @details    This function is system IDLE task code.
*****
*/
void ColdleTask(void* pdata)
{
    /* Add your codes here */
    for(;;)
    {
        /* Add your codes here */
    }
}
```

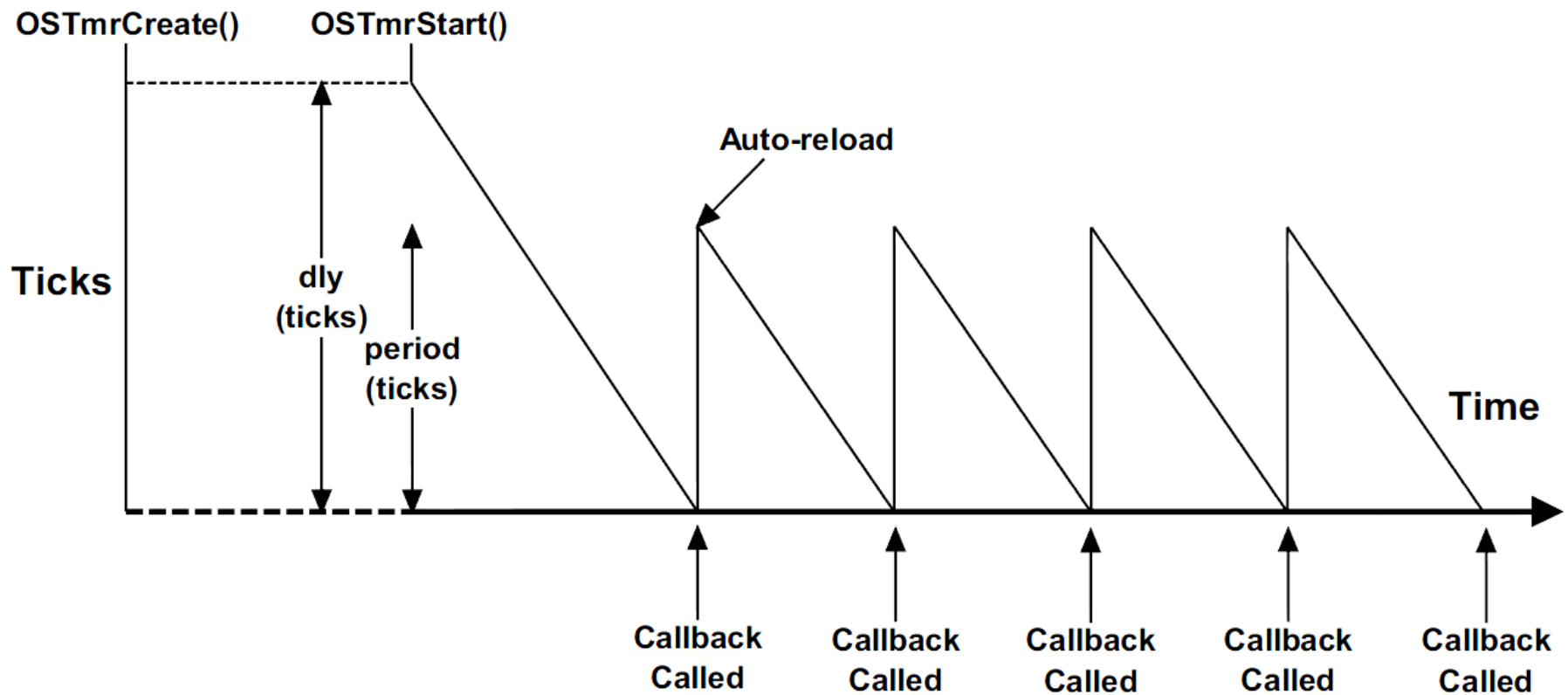

Timers Software

- Los RTOS además de las tareas, permiten la creación de **“timer software”**.
- **Un timer software es una función que se ejecuta con una periodicidad predefinida.**
- Como particularidad, la función que ejecuta **el timer no puede llamar a funciones del RTOS que la bloqueen.**
- **Pueden postear:** banderas, semáforos, colas...
- **Existen dos tipos:**
 - Timers periódicos (**Periodic Timer**).
 - Timers que se ejecutan una sola vez tras un tiempo determinado (**One-Shot Timer**).

One-Shot Timer



Periodic Timer



Funciones para la creación de timers

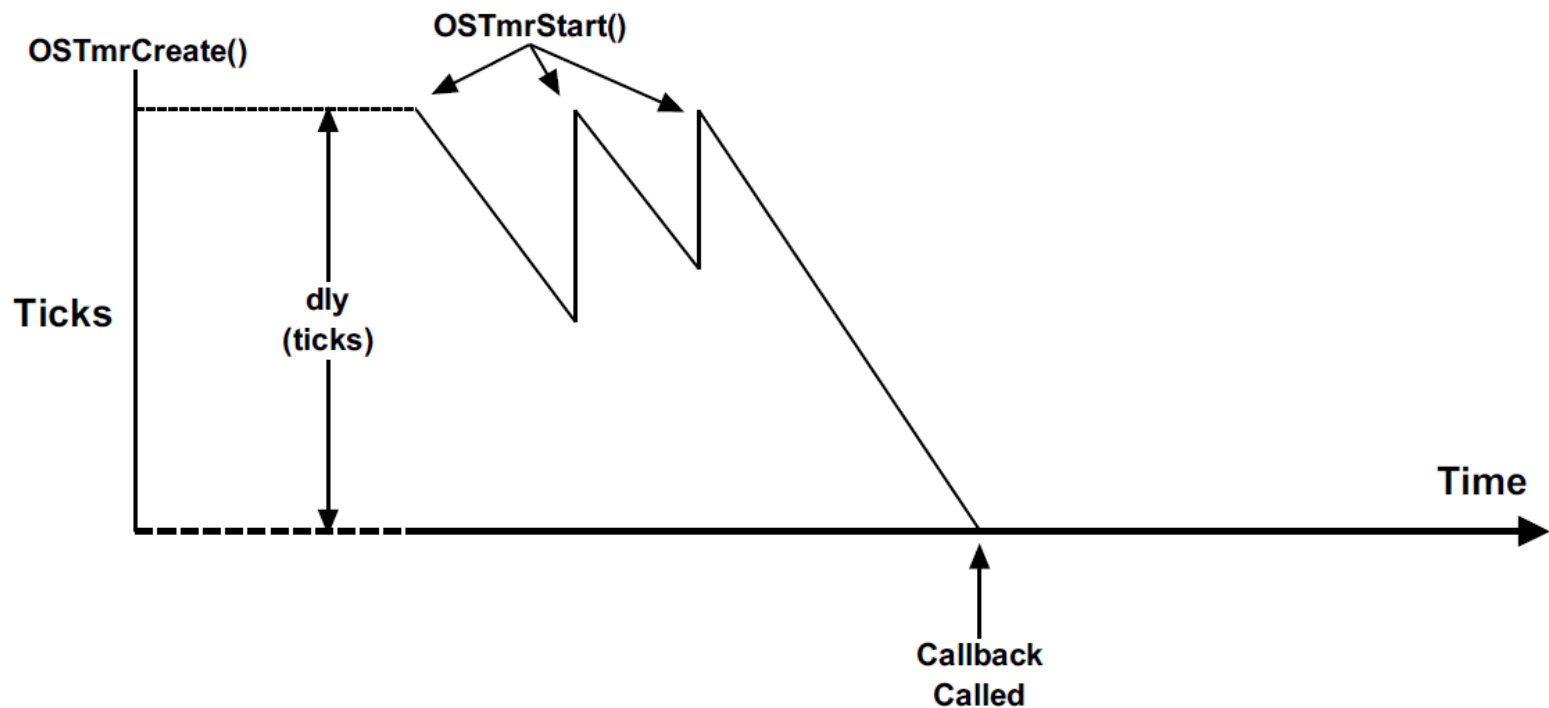
- Para crear un timer software hay que definir:
 - Una variable del tipo **OS_TCID**: identificador del timer.
 - `OS_TCID miTimerID;`
 - Puntero a la función a ejecutar:
 - ```
void miTimer (void) {
 //...
}
```
- El CoOs proporciona las siguientes funciones:
  - `OS_TCID CoCreateTmr (tmrType, tmrCnt, tmrReload, func);`
  - `StatusType CoStartTmr (tmrID);`
  - `StatusType CoStopTmr (tmrID);`
  - `StatusType CoDelTmr (tmrID);`

# Ejemplo de creación de un timer software

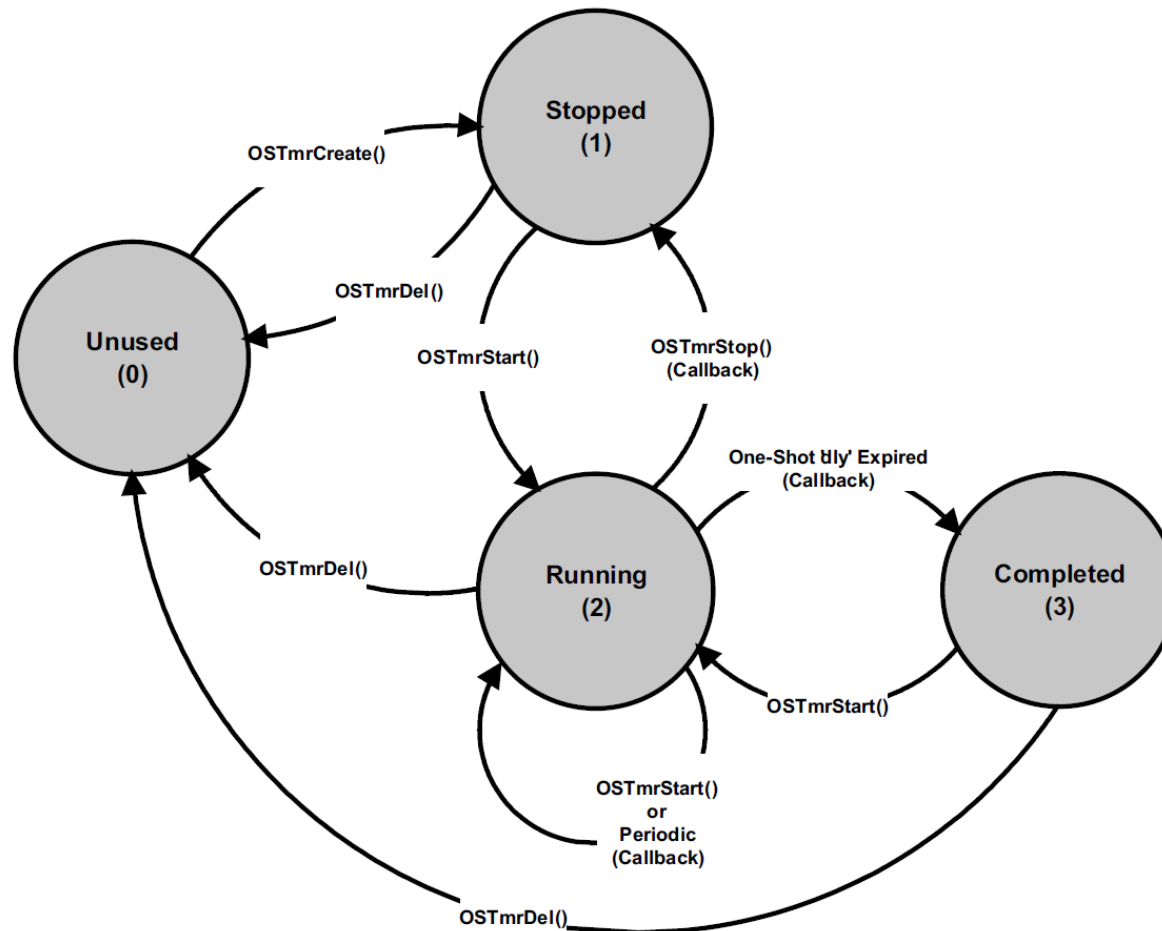
```
void TimerCreate(void){
 OS_TCID timerId; //Identificador del timer
 // ...
 //Creación
 timerId=CoCreateTimer(TMR_TYPE_PERIODIC, ticksDelay, ticksPeriod, miTimer);
 //Inicio
 CoStartTmr(timerId);
 // ...
}

void miTimer (void){
 uint8_t key;
 key = readJoy();
 LED_TOOGLE(key);
}
```

# One-Shot Timer: WatchDog Timer

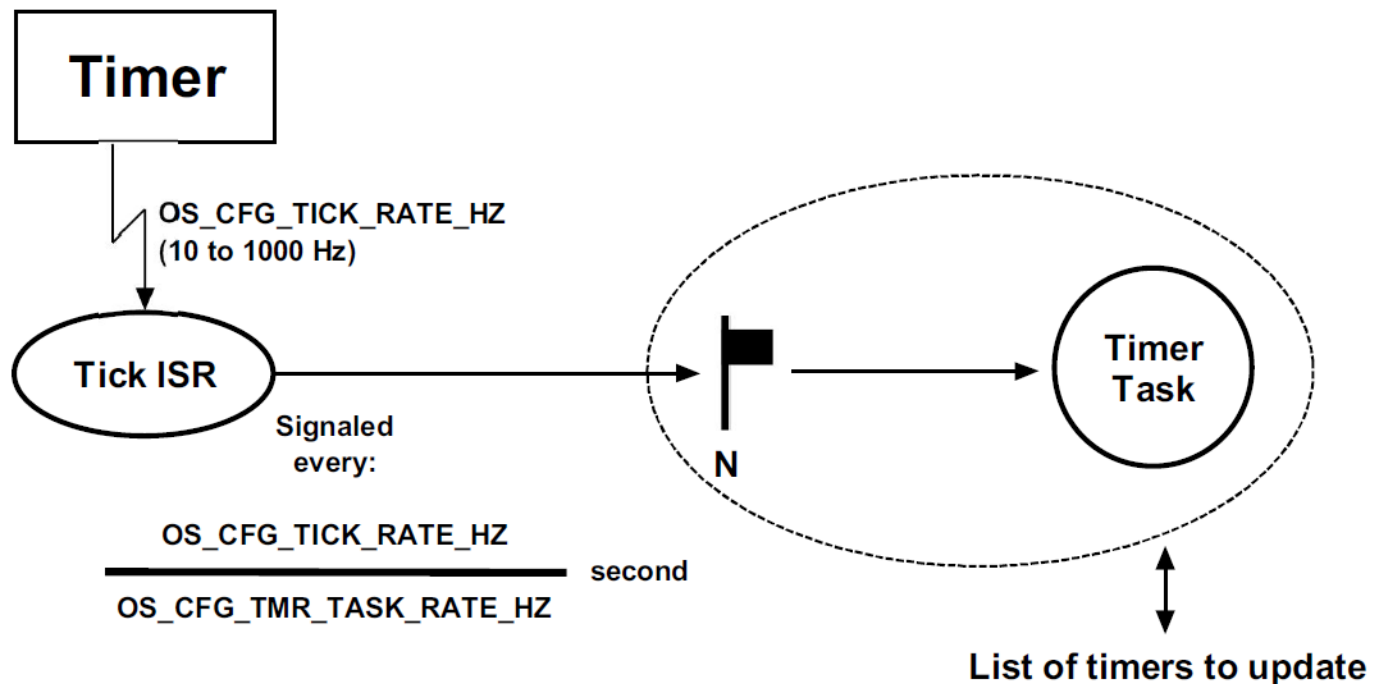


# Diagrama de estado de los timers software



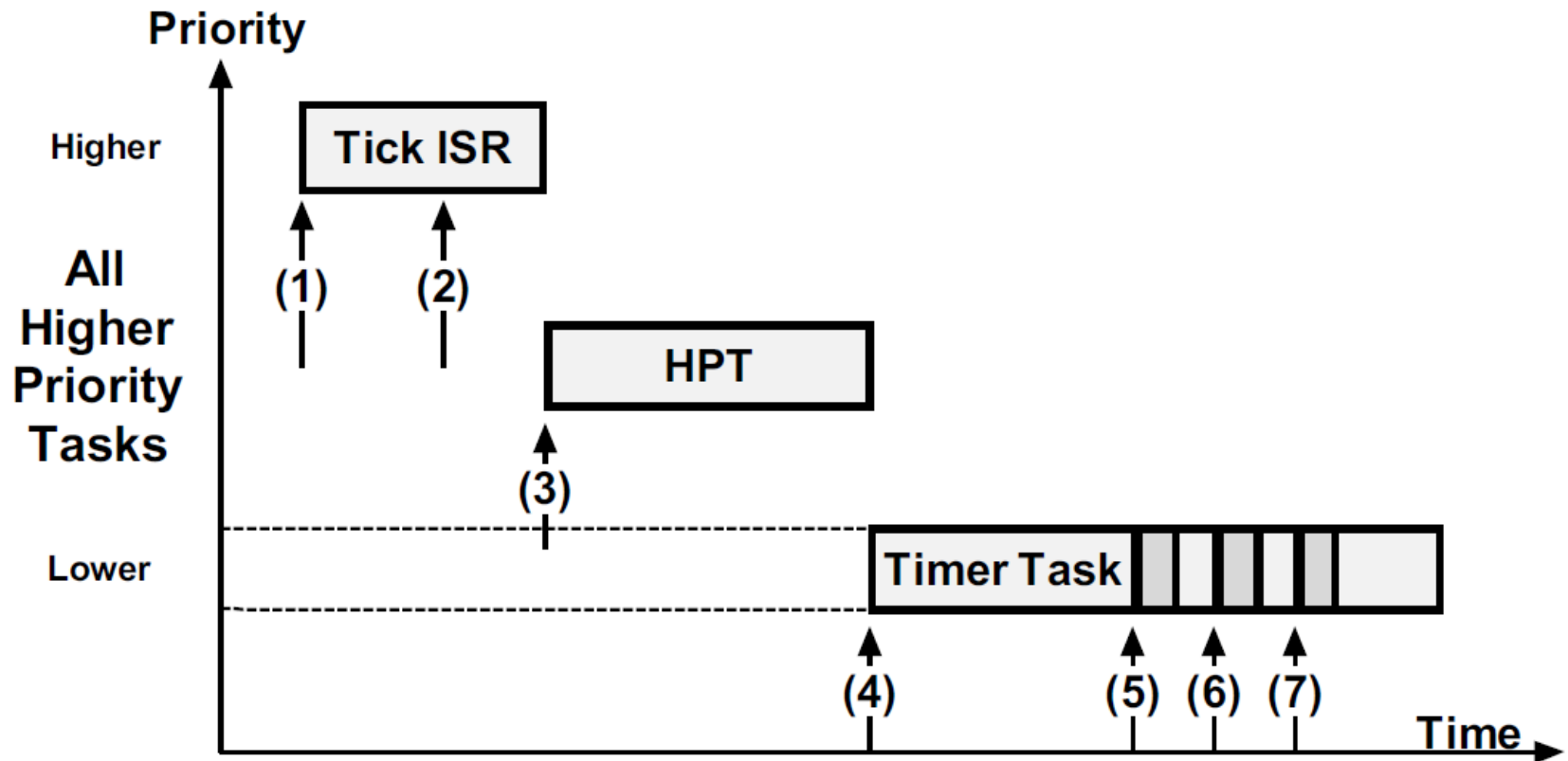
# TimerTask

- Los timers software no se “disparan mágicamente”, realmente son funciones llamadas desde la tarea **TimerTask**.
- Al igual que IdleTask, TimerTask es una tarea interna del RTOS.





# Ejecución de los timers software



# Secciones críticas en CoOs

- Una sección crítica es una región de código que debe ejecutarse sin interferencia por parte de otras tareas/interrupciones:
  - Acceso en exclusiva a un recurso.
  - Funciones temporalmente críticas.
- Antes de ejecutar una sección crítica:
  - Se inhabilitan las interrupciones.
  - Se paraliza el planificador.
- **CoSchedLock(void) y CoSchedUnlock(void):**
  - Des/bloquean el planificador.
- Uso con cuidado por parte del usuario:
  - Temporalmente se rompe con la política de prioridades.
  - El sistema se queda “insensible” a eventos externos.
- Al estar el planificador parado: está prohibido llamar a funciones bloqueantes dentro de una sección crítica.
  - “Cuelgue” del sistema.

# Secciones críticas en CoOs

## Code 7 Critical Section

```
void Task1(void* pdata)
{

 CoSchedLock(); // Enter Critical Section
 // Critical Code
 CoSchedUnlock(); // Exit Critical Section

}
```

# Banderas en CoOs

- Las banderas son usadas para comunicar un evento en el sistema a una o varias tareas.
- Este evento puede provenir de otra tarea, un timer software o una interrupción.
- Una bandera se representa mediante una variable del tipo:
  - **OS\_FlagID miBandera;**
- Creando una bandera:
  - **miBandera = CoCreateFlag (autoReset, initialState)**
    - autoReset = 0 => La bandera se resetea manualmente.
    - autoReset = 1 => La bandera se “consume” al despertar a una tarea.
    - initialState: Estado inicial de la bandera (1: Ready, 0: Non-ready)
- Modificando su estado:
  - Activar: **CoSetFlag(miBandera)**
  - Desactivar: **CoClearFlag(miBandera)**
- Esperando su activación:
  - **Status = CoWaitForSingleFlag (miBandera, TimeOut)**
    - **TimeOut:** Tiempo de espera en ticks del sistema (1 mSeg). Si vale 0, la espera es indefinida.
    - **Status:**

|               |                                         |
|---------------|-----------------------------------------|
| E_INVALID_ID, | The incoming ID of the flag is invalid. |
| E_TIMEOUT,    | Wait overtime.                          |
| E_OK,         | Obtain the flag successfully.           |

# Esperando una bandera

```
void myTaskA(void* pdata)
{

 flagID = CoCreateFlag(0,0); // Reset manually, the original state is not-ready
 CoWaitForSingleFlag(flagID,0);

}

void myTaskB(void* pdata)
{

 CoSetFlag(flagID);

}
```

# Esperando una bandera

- Múltiples tareas esperando una bandera:
  - Las tareas se encolan en orden de petición (FIFO).
  - Si la bandera está configurada como auto-reset, sólo se activa la primera tarea que la solicitó.
  - Si la bandera tiene el reset manual, al activar la bandera todas las tareas pasarán a estado de ready. La bandera debe ser desactivada por la tarea que la activó.

```
.
.
CoSetFlag(flag);
CoClearFlag(flag);
.
.
```

# Esperando múltiples banderas

- Una tarea puede esperar múltiples banderas de forma simultánea:

- Las banderas deben empaquetarse usando una operación OR.
- Hacer uso de la función:

- `flagId = CoWaitForMultipleFlags(flags, options, timeout, &error)`

- En las opciones podemos especificar si esperar la activación de una sola de las banderas (espera OR), o la activación de todas ellas (espera AND).

|                            |                        |
|----------------------------|------------------------|
| <code>OPT_WAIT_ALL,</code> | Wait for all the flags |
| <code>OPT_WAIT_ANY,</code> | Wait for a single flag |

- En caso de una espera OR, devuelve la bandera que se activó.
- En error devuelve el estado de la espera:

|                                   |                                    |
|-----------------------------------|------------------------------------|
| <code>E_INVALID_PARAMETER,</code> | The parameter is invalid.          |
| <code>E_TIMEOUT,</code>           | Wait overtime.                     |
| <code>E_FLAG_NOT_READY,</code>    | The flag isn't in the ready state. |
| <code>E_OK,</code>                | Obtain successfully.               |

# Esperando múltiples banderas

```
void myTaskA(void* pdata)
{
 U32 flag;
 StatusType err;

 flagID1 = CoCreateFlag(0,0); // Reset manually, the original state is not-ready
 flagID2 = CoCreateFlag(0,0); // Reset manually, the original state is not-ready
 flagID3 = CoCreateFlag(0,0); // Reset manually, the original state is not-ready
 flag = flagID1 | flagID2 | flagID3;
 CoWaitForMultipleFlags(flag,OPT_WAIT_ANY,0,&err);

}

void myTaskB(void* pdata)
{

 CoSetFlag(flagID1);

}

void myISR(void)
{
 CoEnterISR();

 isr_SetFlag(flagID2);
 CoExitISR();
}
```



# Semáforos en CoOs

- Los semáforos se utilizan comunmente para controlar el acceso a recursos compartidos.
- Una semáforo es una variable del tipo:
  - `OS_EventID miSem;`
- Creando un semáforo:
  - **`miSem = CoCreateSem (initCount, maxCnt, sortType)`**
    - **`initCount`**: Contador inicial del semáforo.
    - **`maxCnt`**: Valor máximo del contador.
    - **`sortType`**: Ordenación de tareas a la espera:
      - **`EVENT_SORT_TYPE_FIFO`**: se encolan en una fifo.
      - **`EVENT_SORT_TYPE_PRIO`**: se ordenan en base a la prioridad.
- Esperar a que un semáforo esté libre:
  - **`status = CoPendSem(miSem, timeout)`**
    - `E_INVALID_ID,` the semaphore ID that was  
incomed is invalid
    - `E_TIMEOUT,` time is out for waiting resources
    - `E_OK,` to obtain the resources successfully
- Liberando un semáforo:
  - **`CoPostSem (miSem)`**

# Ejemplo semáforo binario (inicialmente cerrado)

```
void myTaskA(void* pdata)
{

 semID = CoCreateSem(0,1,EVENT_SORT_TYPE_FIFO);
 CoPendSem(semID,0);

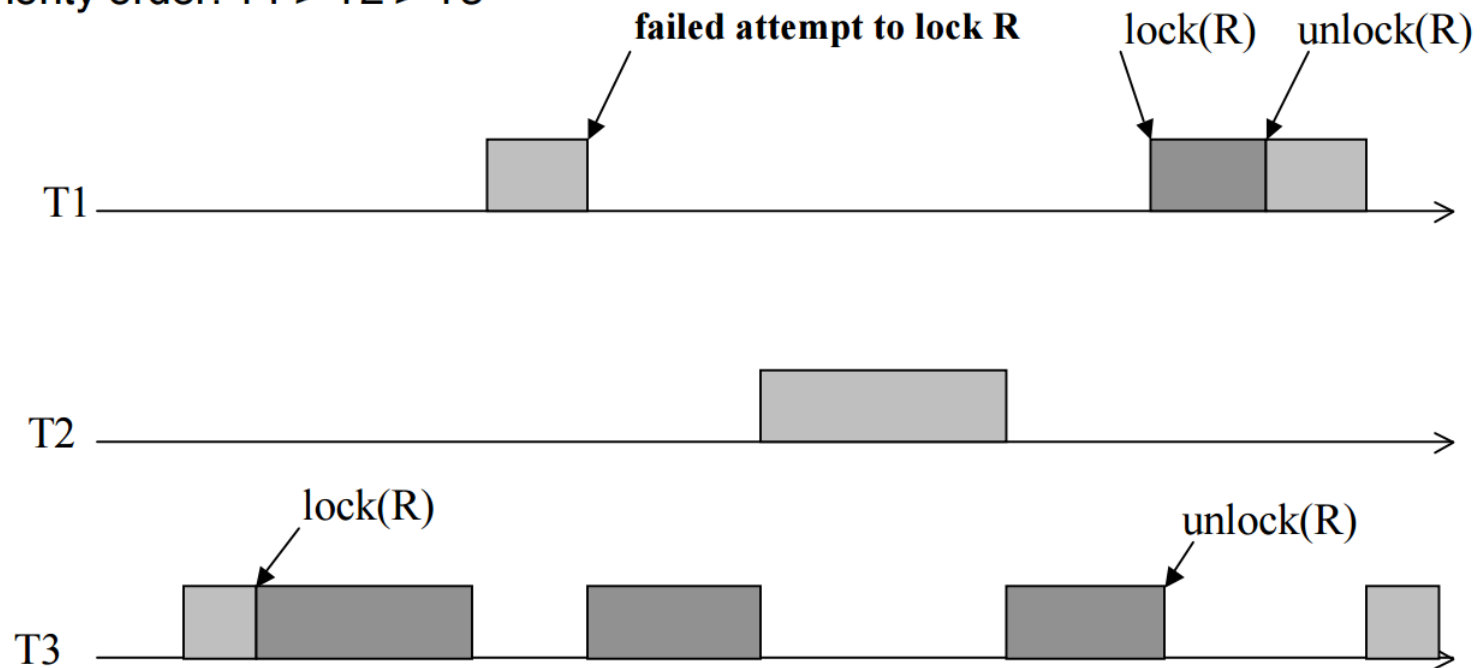
}
void myTaskB(void* pdata)
{

 CoPostSem(semID);

}
```

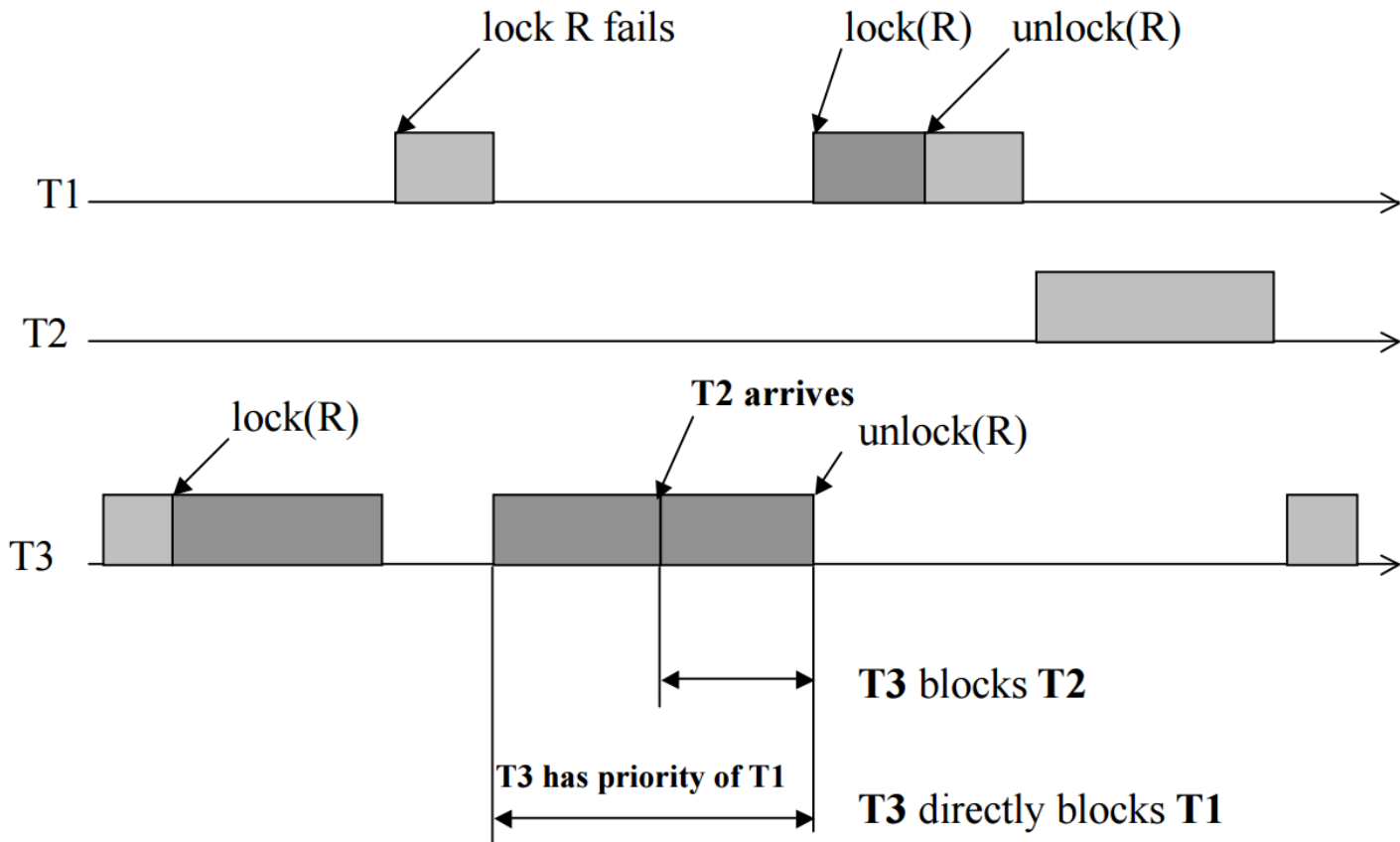
# Inversión de prioridad

Priority order:  $T1 > T2 > T3$



**T2 is causing a higher priority task T1 wait !**

# Herencia de prioridad

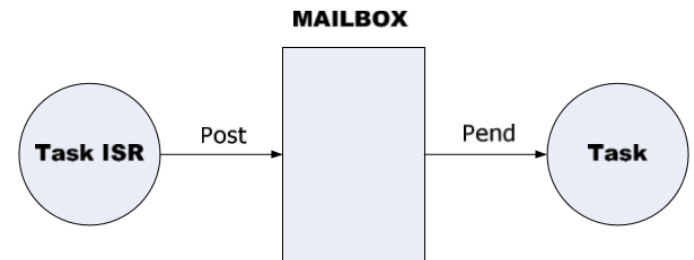


# Comunicación entre tareas/interrupciones

- Cuando hablamos de comunicación nos referimos al intercambio de información entre tareas, así como el intercambio entre tareas e interrupciones.
- Hay tres maneras de comunicar información mediante tareas / interrupciones:
  1. Uso de variables globales.
  2. Buzones de mensajes.
  3. Colas de mensajes.
- En caso de usar **variables globales** es necesario asegurarnos de que sólo una tarea acceda a la variable global de manera simultánea.
- Una forma común para asegurarnos es mediante el uso de semáforos.
- Si embargo las interrupciones NO se pueden bloquear a la espera de la liberación de un semáforo.

# Buzones de Mensajes (Mail Boxes)

- Los Mail Boxes son usados para comunicar tareas e interrupciones de manera concurrente.
- A través de un Mail Box podemos transmitir:
  - Un dato de 32 bits.
  - Un puntero a un buffer de datos.
- Una Mail Box es una variable del tipo:
  - **OS\_EventID miMBox;**
- Creando un Mail Box:
  - **miMBox = CoCreateMBox (sortType)**
    - **sortType:** Ordenación de tareas a la espera:
      - **EVENT\_SORT\_TYPE\_FIFO:** se encolan en una fifo.
      - **EVENT\_SORT\_TYPE\_PRIO:** se ordenan en base a la prioridad.
- Enviando un mensaje:
  - **CoPostMail (miMbox, data)**
- Esperar a recibir un mensaje:
  - **Data = CoPendMail(miMBox, timeout, &status)**



E\_INVALID\_ID, the mailbox ID that was  
incomed is invalid  
E\_TIMEOUT, time is out for waiting resources  
E\_OK, to obtain the resources successfully

```

void myTaskA(void* pdata)
{
 void* pmail;
 StatusType err;

 mboxID = CoCreateMbox(EVENT_SORT_TYPE_PRIO); //Sort by preemptive priority
 pmail = CoPendMail(mboxID,0,&err);

}
void myTaskB(void* pdata)
{

 CoPostMail(mboxID,"hello,world");

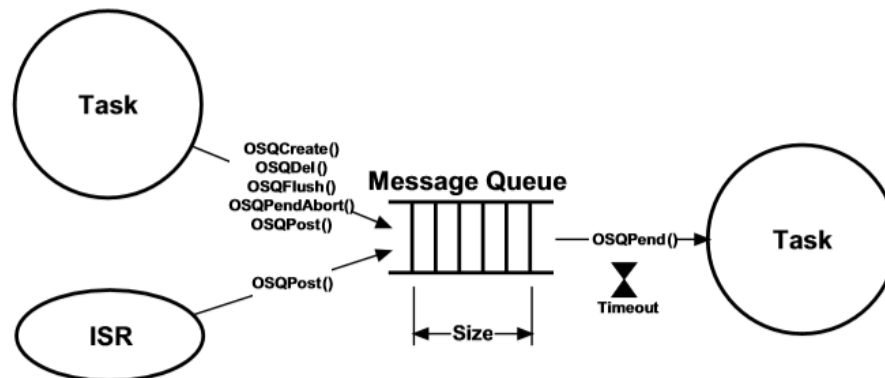
}
void myISR(void)
{
 CoEnterISR();

 isr_PostMail(mboxID,"hello,CooCox");
 CoExitISR();
}

```

# Colas de mensajes en CoOs

- Son similares a los Mail Boxes, pero permiten ir encolando múltiples mensajes.
- Se usan para enviar:
  - Un dato de 32 bits.
  - Un puntero a un buffer de datos.
- Para declarar una cola hacen falta 2 elementos:
  1. Un identificador de la cola:  
**OS\_EventID queueId;**
  2. En espacio de memoria donde almacenar los datos:  
**void \* queueBuffer[tamCola];**





# Colas de mensajes en CoOs

- Creación:
  - **queueId = CoCreateQueue (queueBuffer, tamCola, EVENT\_SORT\_TYPE\_PRIO);**
- Encolar:
  - **CoPostQueueMail (queueId, dato);**
  - **isr\_PostQueueMail (queueId, dato);**
- Esperar a recibir un dato:
  - **Dato = CoPendQueueMail (queueId, timeout, &status);**
  - En el estado podemos encontrar los siguientes valores:

|               |                                    |
|---------------|------------------------------------|
| E_INVALID_ID, | the message queue ID               |
|               | that was incomed is                |
|               | invalid                            |
| E_TIMEOUT,    | time is out for waiting resources  |
| E_OK,         | to obtain the message successfully |

# Ejemplo uso de colas de mensajes

```
void myTaskA(void* pdata)
{
 void* pmail;
 Void* queue[5];
 StatusType err;

 queueID = CoCreateQueue(queue,5,EVENT_SORT_TYPE_PRIO);
 //5 grade, sorting by preemptive priority
 pmail = CoPendQueueMail(queueID ,0,&err);

}
void myTaskB(void* pdata)
{

 CoPostQueueMail(queueID ,"hello,world");

}
void myISR(void)
{
 CoEnterISR();

 isr_PostQueueMail(queueID ,"hello,CooCox");
 CoExitISR();
}
```

# Sincronizando Tareas con ISR

- Tradicionalmente la los periféricos se accede usando:
  - **Esperas activas:** desperdiciando tiempo de CPU.
  - **Interrupciones:** implementando máquinas de estado asíncronas.
  - **DMA:** Programando transacciones en el DMA, e implementando la interrupción del fin de DMA.
- Los RTOS implementan mecanismos para sincronizar tareas con interrupciones, simplificando mucho uso.

# Sincronizando Tareas con ISR

- La E/S se programa iterativamente como si usáramos esperas activas, pero en lugar de usar esperas activas, se usan eventos del SO disparados desde las interrupciones.
- Para sincronizar las tareas con las ISR se usan comúnmente flags, semáforos y colas.
- Conceptualmente podemos clasificar las interrupciones:
  - **Avisan de un evento:** fin de transmisión, timeouts, errores... **Las tareas se despiertan mediante flags o semáforos.**
  - **Obtienen datos:** datos recibidos, fin de conversión ADCs... **Envían el dato a la tarea mediante una cola de mensajes.**

# Interrupciones en CoOs

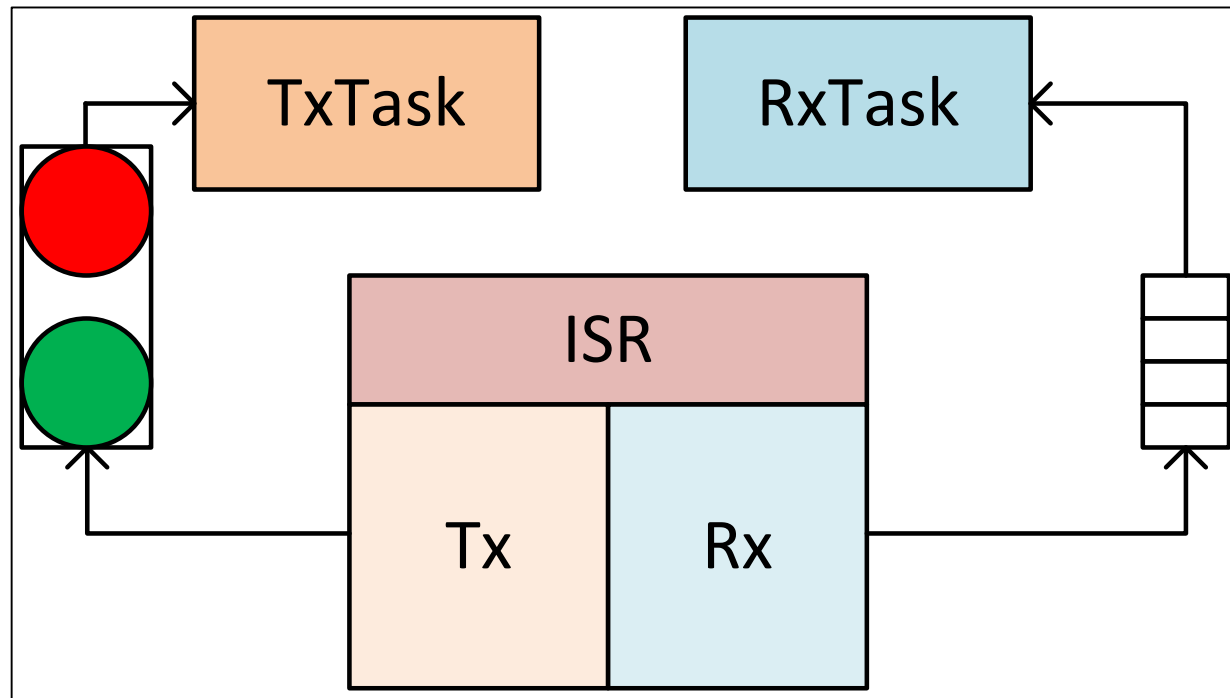
- Las interrupciones funcionan al margen del planificador.
- Se debe avisar que:
  - **Se ha entrado en una ISR** (parar el planificador).
  - **Se ha finalizado una ISR** (replanificar).

```
void WWDG_IRQHandler(void)
{
 CoEnterISR(); // Enter the interrupt
 isr_SetFlag(flagID); // API function
 ; // Interrupt service routine

 CoExitISR(); // Exit the interrupt
}
```

# Sincronizando Tareas con ISR

- Ejemplo transmisión serie:
  - **2 tareas:** Transmisión y recepción.
  - **Para transmitir** se usa un semáforo.
  - **Para recibir** una cola de mensajes.



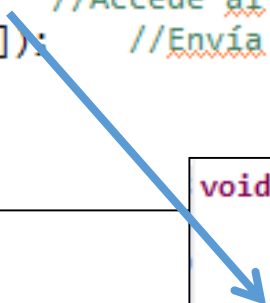
# Sincronizando Tareas con ISR

- En el caso de la USART, la idea es adquirir un semáforo, y en caso de que esté libre, comenzar la transmisión.
- Una vez se complete la transmisión, se disparará la interrupción, y se liberará el semáforo.

```
void TxTask (void * parg){
 char c[]="HOLA";

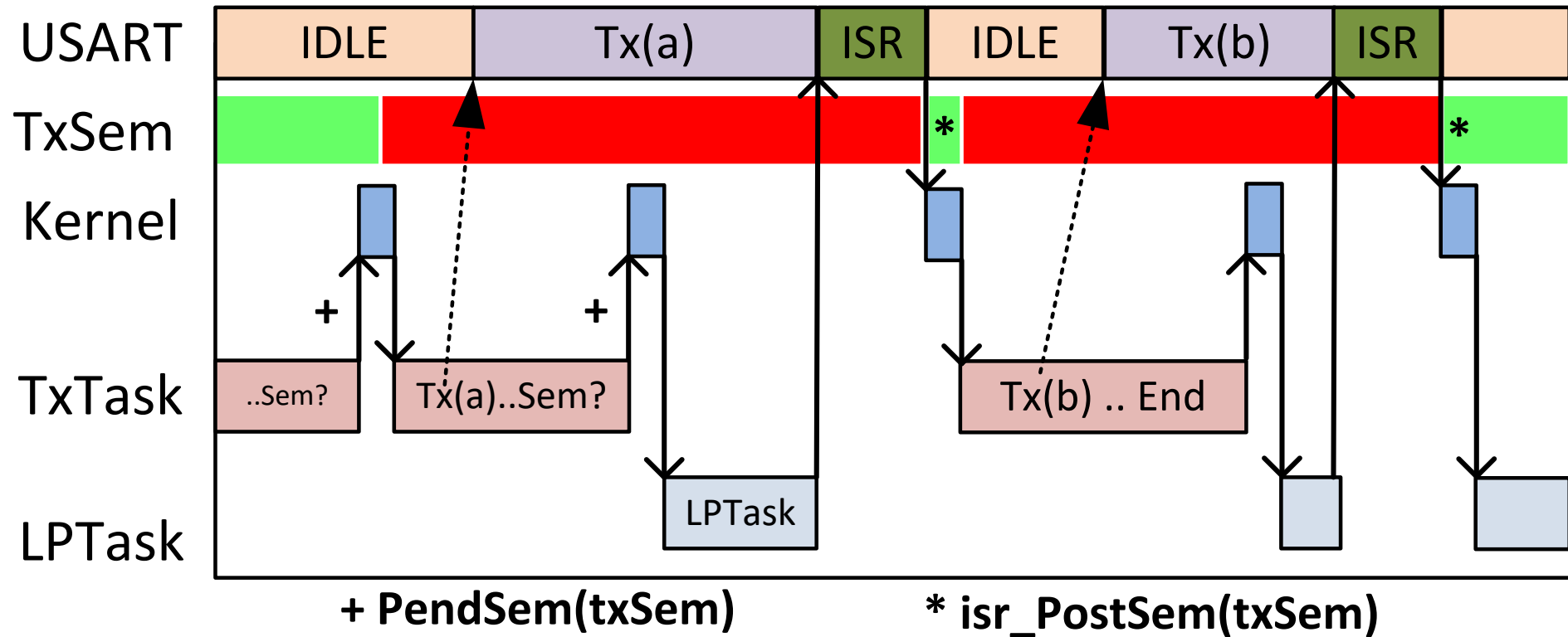
 //..
 for(i=0;i<4;i++){
 PendSem(txSem,0); //Accede al semáforo de la USART
 UART_SendByte(c[i]); //Envía el dato
 }
 //..
}
```

```
void USART_IRQHandler (void){
 //..
 isr_PostSem(txSem); //Se libera el semáforo
 //..
 //al final de la transmisión
}
```



# Tarea a la espera de comenzar una transmisión

- Diagrama de ejecución:





# Enviando información desde la ISR

- Para enviar información desde la ISR a una tarea, hay que utilizar una cola.
- La tarea se bloquea esperando a recibir un dato desde la cola.
- La ISR se encarga de leer el dato y encolarlo.
- La tarea se desbloquea, procesa el dato, y vuelve a bloquearse a la espera de nuevos datos.

```
void RxTask(void * parg){
 char c[4];

 //...
 for(i=0;i<4;i++){
 c[i] = pendQueue(rxQueue);
 }
 //..
}
```

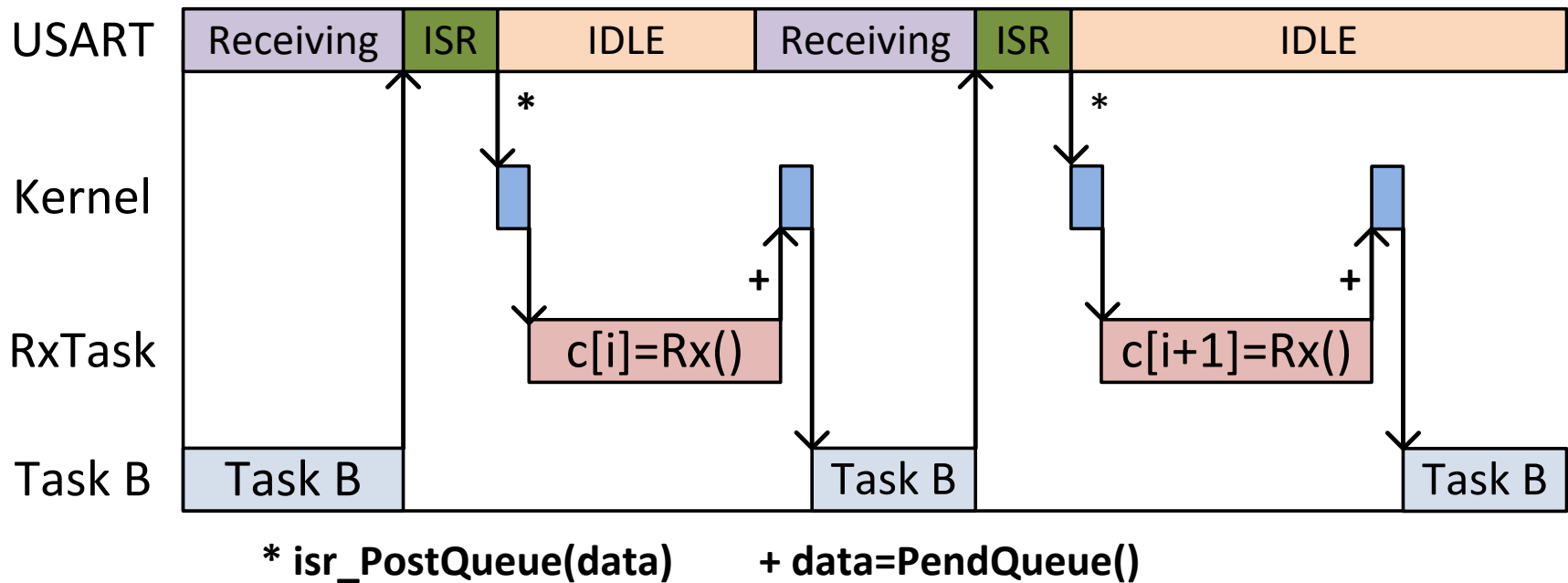
```
void DEV_IRQHandler (void){

 //...
 data = readUsart1();
 isr_postQueue (rxQueue,data);

 //..
```

# Leyendo datos desde una interrupción.

- Diagrama de ejecución:



# Interrupciones en CoOs

- El CoOs proporciona funciones especiales para avisar a las tareas de eventos (banderas y semáforos):
  - `void isr_SetFlag( flag )`
  - `void isr_PostSem( sem )`
- Para transmitir datos desde la ISR a la tarea:
  - `void isr_PostQueueMail (queue, data)`
- **UNA ISR NUNCA DEBE BLOQUEARSE DEBIDO A UNA LLAMADA A Pend\_...**

# Ejemplo: Recibiendo datos del puerto serie

- La idea es tener una cola que comunique los datos recibidos desde la ISR a la tarea de recepción serie.
- A continuación esta encolará el dato recibido en una nueva tarea del LCD, la cual mostrará los caracteres recibidos en el display LCD.

