ARM Assembly

Set de Instrucciones Thumb-2

Ing. Pablo Ridolfi
Técnicas Digitales II

ISA Thumb-2

- Los procesadores ARM clásicos disponen de dos modos de ejecución:
  - **ARM** (instrucciones de 32-bits).
    Se aprovecha al máximo la arquitectura con instrucciones potentes. Admite ejecución condicional.

  - **Thumb** (instrucciones de 16-bits).
    Menos (y más simples) instrucciones que el modo ARM, pero admite mayor densidad de código y optimización en sistemas con memoria crítica. Algunas operaciones que en modo ARM requieren sólo una instrucción, pueden requerir más en modo Thumb.

ISA Thumb-2

- La familia de procesadores Cortex (mejor dicho, la arquitectura ARMv7) buscando aumentar la performance del modo Thumb incorpora nuevas instrucciones de 32-bits, convirtiéndolo en un ISA mixto de 16/32-bits.
  - Se le llama **Thumb-2**.

- Es importante remarcar que Cortex-M3 **no soporta el modo ARM.**
  - Si intentamos pasar a modo ARM, se generará una excepción tipo Usage Fault.

(en procesadores que lo soporten, se puede pasar a modo ARM poniendo a cero el bit T del PSR, haciendo un branch cuyo address tiene el LSb en cero, o modificando EXC_RETURN del LR)

ISA Thumb-2

Una línea de código Assembler (GNU AS) se ve así:

```
etiqueta: opcode op1,op2,...  //comentario
```

Ejemplo:

```
    eor r0,r0       // r0 ^= r0;
    push {lr}       // sp -= 4; *sp = lr
repetir:
    add r0,12       // r0 += 12
    cmp r0,12*5     // comparar
    bne repetir     // saltar si no es igual
    pop {lr}        // lr = *sp; sp += 4
    ldr r0,=0x12345678 /* r0 = 0x12345678
(pseudo instrucción) r0 = mem[pc+offset] */
```

ISA Thumb-2

Sufijos: **S** Actualiza los flags de estado (`cmp` siempre lo hace).

```
add r0,r1 //no actualiza
adds r0,r1 //actualiza
```

Condicionales:

Suffix	Flags	Meaning
EQ	Z = 1	Equal
NE	Z = 0	Not equal
CS or HS	C = 1	Higher or same, unsigned ≥
CC or LO	C = 0	Lower, unsigned <
MI	N = 1	Negative
PL	N = 0	Positive or zero
VS	V = 1	Overflow
VC	V = 0	No overflow
HI	C = 1 and Z = 0	Higher, unsigned >
LS	C = 0 or Z = 1	Lower or same, unsigned ≤
GE	N = V	Greater than or equal, signed ≥
LT	N != V	Less than, signed <
GT	Z = 0 and N = V	Greater than, signed >
LE	Z = 1 and N != V	Less than or equal, signed ≤
AL	Can have any value	Always; default when no suffix is specified

ISA Thumb-2

Direccionamiento indirecto e indexado:

```
ldr r1,[r0] // r1 = *r0
ldr r1,[r0,960] // r1 = *(r0+960)
ldr r1,[r0,960]! // r1 = *(r0+960); r0+=960

str r1,[r0],-4 // *r0 = r1; r0-=4
ldrd r0,r1,[r2,0x10]
    // r0 = *(r2+16); r1 = *(r2+16+4)

str r5,[r1,r2] // *(r1+r2) = r5
ldr r5,[r1,r2,LSL 2] // r5 = *(r1 + (r2 << 2))

ldm r3!,{r0-r2}
    // r0=*r3; r1=*(r3+4); r2=*(r3+8); r3+=12
```

ISA Thumb-2

Ejemplos:

```
//lectura/escritura de registros especiales
msr psp,r0 // psp = r0 OJO! Solo privilegiado
msr apsr,r0 // apsr = r0 ok en ambos modos
mrs r0, apsr // r0 = apsr

cpsid i //PRIMASK = 1
cpsid f //FAULTMASK = 1
cpsie i //PRIMASK = 0
cpsie f //FAULTMASK = 0

//bloque IF-THEN (máx. 4 instr. condicionales)
cmp r0,r1
ittee eq          // if(r0==r1) {
addeq r3,r4,r5 //      r3 = r4 + r5
asreq r3,r3,1  //      r3 /= 2 }
addne r3,r6,r7 // else { r3 = r6 + r7
asrne r3,r3,1  // r3 /= 2 }
```

ISA Thumb-2

Ejemplos:

```
//compare-branch condicional
//OJO: solo forward branch!
loop: cbz r0,salir // while(r0 != 0) {
      bl funcion   //      funcion();
      b loop       // }
salir:            //

//Otro ejemplo
      mov r0,10+1
loop2:sub r0,1
      cbnz r0,endloop2
      add r1,1
      b loop2
endloop2:
```

ISA Thumb-2

Ejemplos:

```
//reversión de bytes: si r0 = 0x12345678
    rev r1,r0     // r1 = 0x78563412
    revh r2,r0    // r2 = 0x34127856


//reversión de bits:
//r0=1011 0100 1110 0001 0000 1100 0010 0011
    rbit r1,r0
//r1=1100 0100 0011 0000 1000 0111 0010 1101
```

ISA Thumb-2

Ejemplos:

```
//manejo de campos de bits

// bit-field clear
//  bfc rd,#lsb,#width
    ldr r0,=0x1234FFFF
    bfc r0,4,8 // r0=0x1234F00F

// bit-field insert
//  bfi rd,rn,#lsb,#width
    ldr r0,=0x12345678
    ldr r1,=0x3355AACC
    bfi r1,r0,8,16 // r1 = 0x335678CC
```

ISA Thumb-2

Ejemplos:

```
//manejo de campos de bits

// bit-field extract
//  ubfx rd,rn,#lsb,#width
    ldr r0,=0x5678ABCD
    ubfx r1,r0,4,8 // r1=0x000000BC

//  sbfx rd,rn,#lsb,#width
    ldr r0,=0x5678ABCD
    sbfx r1,r0,4,8 // r1=0xFFFFFFBC
```

ISA Thumb-2

Ejemplos: //tablas de saltos

```
// tbb [rn,rm] rn:&tabla, rm:index(byte)
        tbb [pc,r0]
tabla:.byte (dst0-tabla)/2
      .byte (dst1-tabla)/2
      .byte (dst2-tabla)/2
      .byte (dst3-tabla)/2
dst0: ldr r0,=0x12345678
      b fin
dst1: ldr r0,=0x23456789
      b fin
dst2: ldr r0,=0x34567890
      b fin
dst3: ldr r0,=0x4567890A
      b fin
fin:  bx lr
```

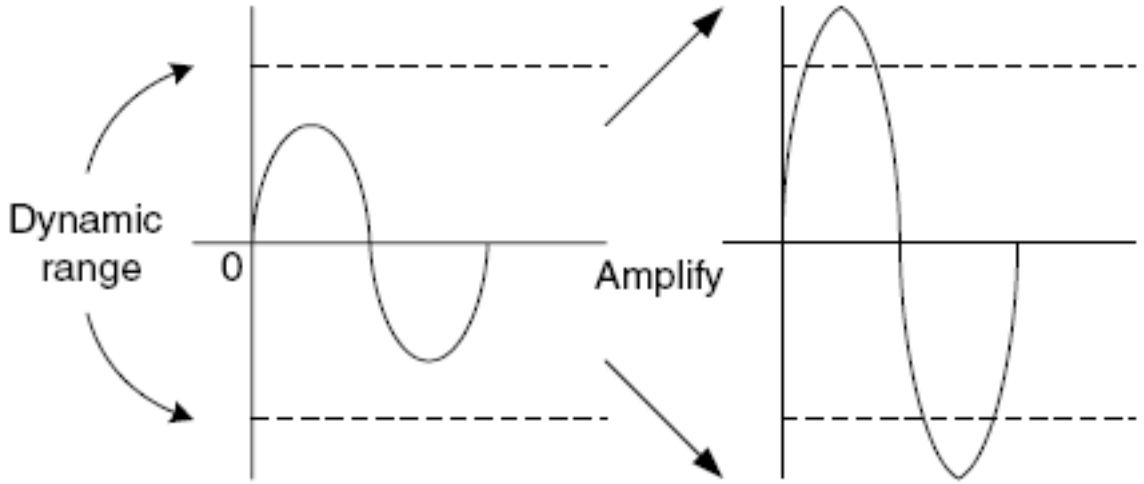
ISA Thumb-2

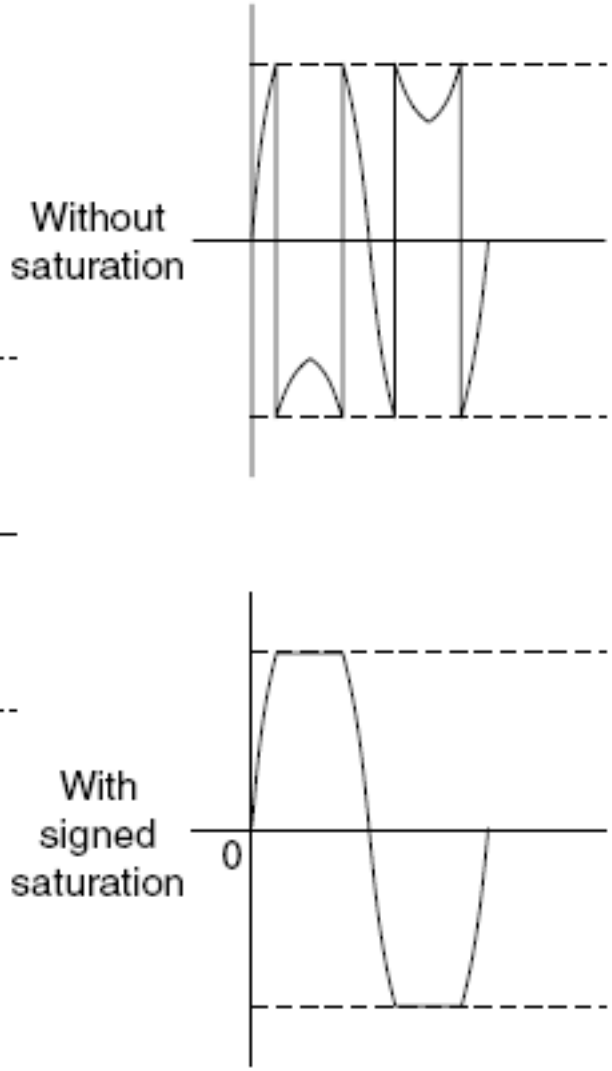
Ejemplos: //**tbh** [rn,rm,lsl 1]

```
        tbh [pc,r0,lsl 1]
tabla:  .short (dst0-tabla)/2
        .short (dst1-tabla)/2
        .short (dst2-tabla)/2
        .short (dst3-tabla)/2
dst0:   ldr r0,=0x12345678
        bx lr
dst1:   ldr r0,=0x23456789
        bx lr
dst2:   ldr r0,=0x34567890
        bx lr
dst3:   ldr r0,=0x4567890A
        bx lr
```

ISA Thumb-2

Saturación:



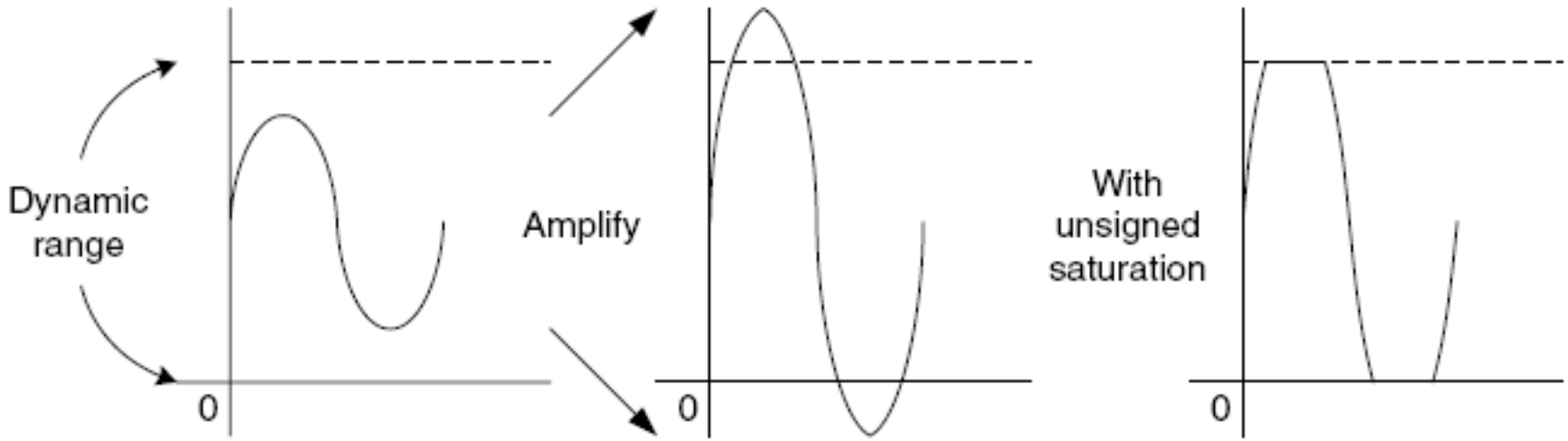



```
//{s|u}sat rd,#bits,rn
ssat r1,16,r0
```

Input (R0)	Output (R1)	Q Bit
0x00020000	0x00007FFF	Set
0x00008000	0x00007FFF	Set
0x00007FFF	0x00007FFF	Unchanged
0x00000000	0x00000000	Unchanged
0xFFFF8000	0xFFFF8000	Unchanged
0xFFFF7FFF	0xFFFF8000	Set
0xFFFE0000	0xFFFF8000	Set

ISA Thumb-2

Saturación:



usat r1,16,r0

Input (R0)	Output (R1)	Q Bit
0x00020000	0x0000FFFF	Set
0x00008000	0x00008000	Unchanged
0x00007FFF	0x00007FFF	Unchanged
0x00000000	0x00000000	Unchanged
0xFFFF8000	0x00000000	Set
0xFFFF8001	0x00000000	Set
0xFFFFFFFF	0x00000000	Set

Interfaz entre Assembler y C

- Al combinar los lenguajes es necesario tener en cuenta algunas reglas establecidas por el estándar **AAPCS** (ARM Architecture Procedure Call Standard).

- Sea **`asm_funcion`** una rutina escrita en assembler.
  - Si se la llama desde C con el siguiente prototipo
    `asm_funcion``(a,b,c,d);`
    entrando en el código assembler tendremos que
    `r0=a; r1=b; r2=c; r3=d;`
    - Parámetros de 64-bits usan r0|r1 y r2|r3 combinados.
  - Más de cuatro parámetros deberán pasarse usando punteros (por referencia) o mediante la pila.
  - **`asm_funcion`** puede devolver un valor cualquiera *dejándolo en* ***`r0`*** antes de retornar.
    - O bien en r0|r1 para entidades de 64-bits.

Diseño de Software Embebido

Exclusión Mutua – Sincronización de tareas:

- Si dos o más tareas necesitan acceder a un recurso (un periférico, escribir en un vector, etc.) pueden aparecer problemas si se produce un cambio de contexto en el medio de una operación de lectura/escritura.
- Para este caso se utilizan los denominados **Semáforos** ó **MutEx**'s (Mutual Exclusion Locks):
  - La tarea que desea acceder a un recurso controlado por un semáforo, verificará primero que el semáforo esté liberado (unlocked). En ese caso, lo toma (locked) y accede al recurso. Una vez que finalizó el acceso, libera el semáforo.
  - Si otra tarea encuentra al semáforo tomado, se bloqueará hasta que el semáforo se libere y le permita el acceso al recurso. En ese instante lo tomará para que cualquier otra tarea espere a que termine.

Diseño de Software Embebido

Exclusión Mutua – Sincronización de tareas:

- En Cortex-M3, se usa una variable en RAM como registro de bloqueo. Ese registro podrá valer cero o uno.
- Leer esa posición de memoria para saber si el mutex está tomado puede llevar más de un ciclo de máquina debido a los múltiples buses (arq. Harvard), y en el medio de la lectura puede ocurrir un cambio de contexto.
- Necesitamos que la lectura/escritura del registro de bloqueo sea **atómica** (que nadie nos interrumpa esa lectura/escritura).
- Para eso la arquitectura ARMv7-M pone a nuestra disposición dos instrucciones Thumb-2:

  - Lectura exclusiva:

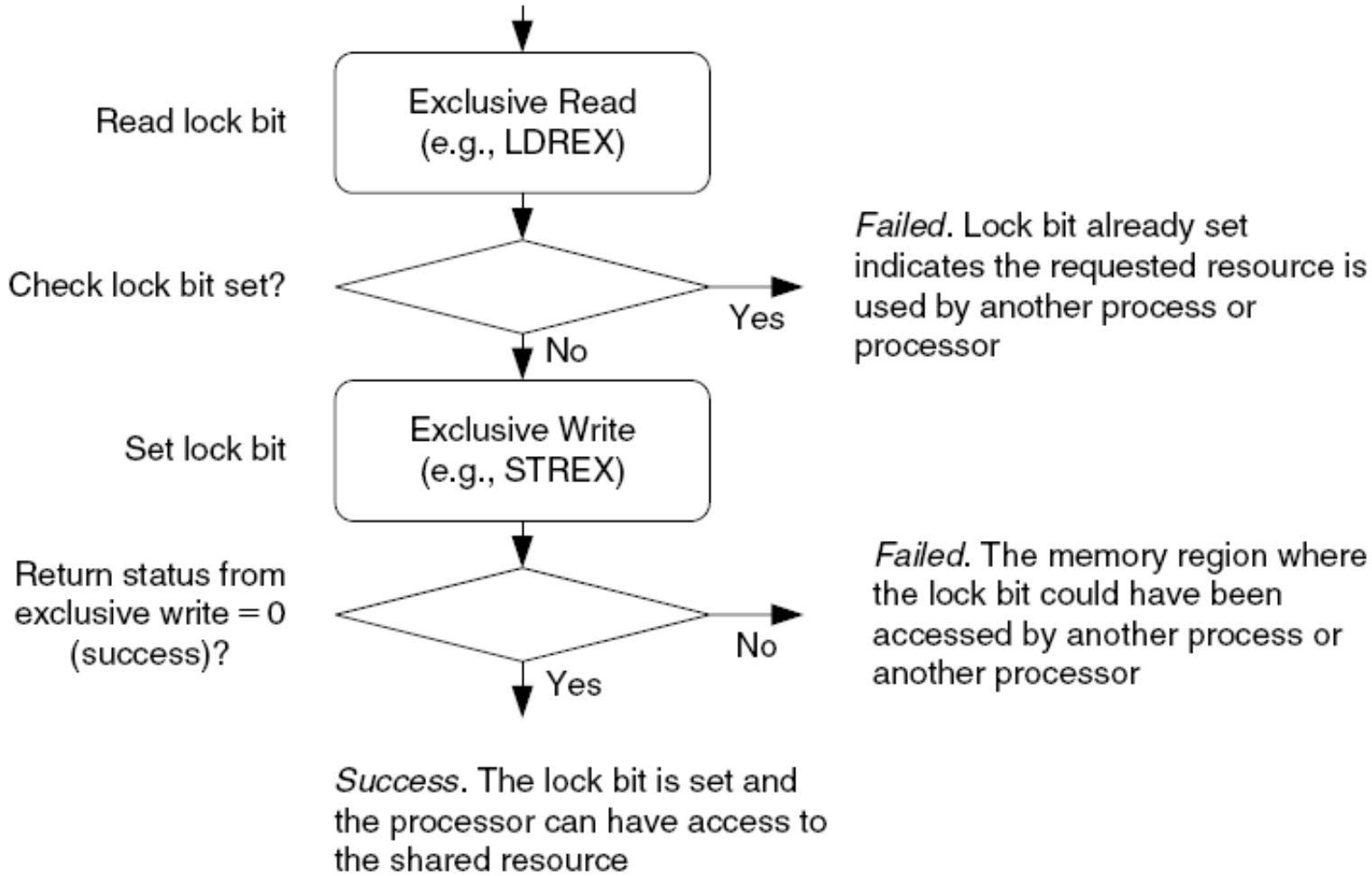
    ```
    ldrex rxf,[rn,offset]
    ```

  - Intentar escritura exclusiva, avisar si fue satisfactoria:

    ```
    strex rd,rxf,[rn,offset]
    ```

Diseño de Software Embebido

Verificar si un proceso está usando un recurso mediante un lock-bit:



Diseño de Software Embebido

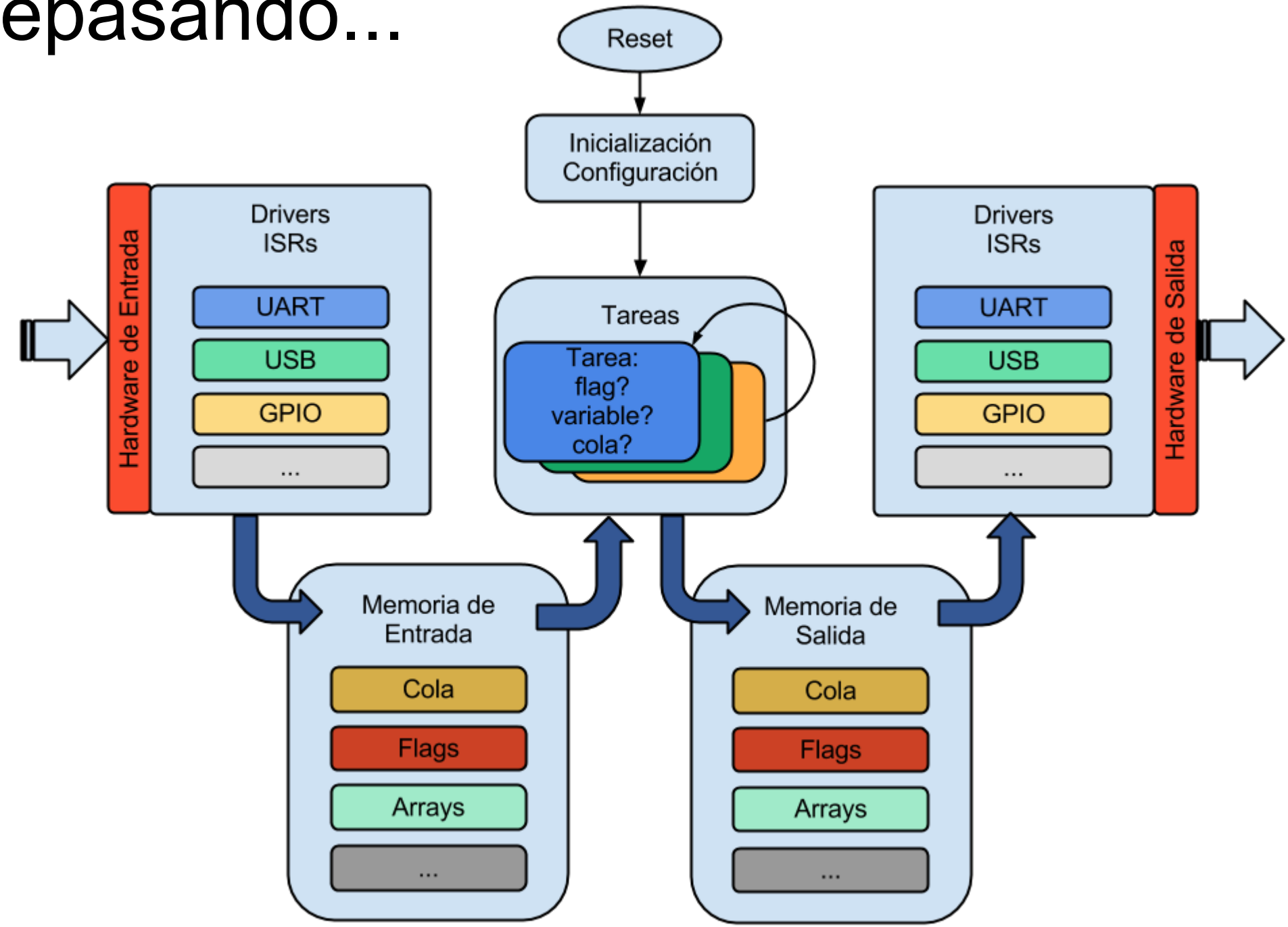
Verificar si un proceso está usando un recurso mediante un lock-bit:

```
verificarMutex:
    push {r1,r2,lr}
    ldr r1,=lockBit     //leo (exc.) el registro
    ldrex r2,[r1]
    cmp r2,0            //fue tomado?
    bne recursoTomado
    mov r0,1            //intento escribir (exc.)
    strex r2,r0,[r1]
    cmp r2,0
    bne recursoTomado //escritura satisfactoria?
    mov r0,0            //recurso libre (dev. 0)
    pop {r1,r2,pc}
recursoTomado:
    mov r0,1            //recurso ocupado (dev. 1)
    pop {r1,r2,pc}
```

Caso Práctico:

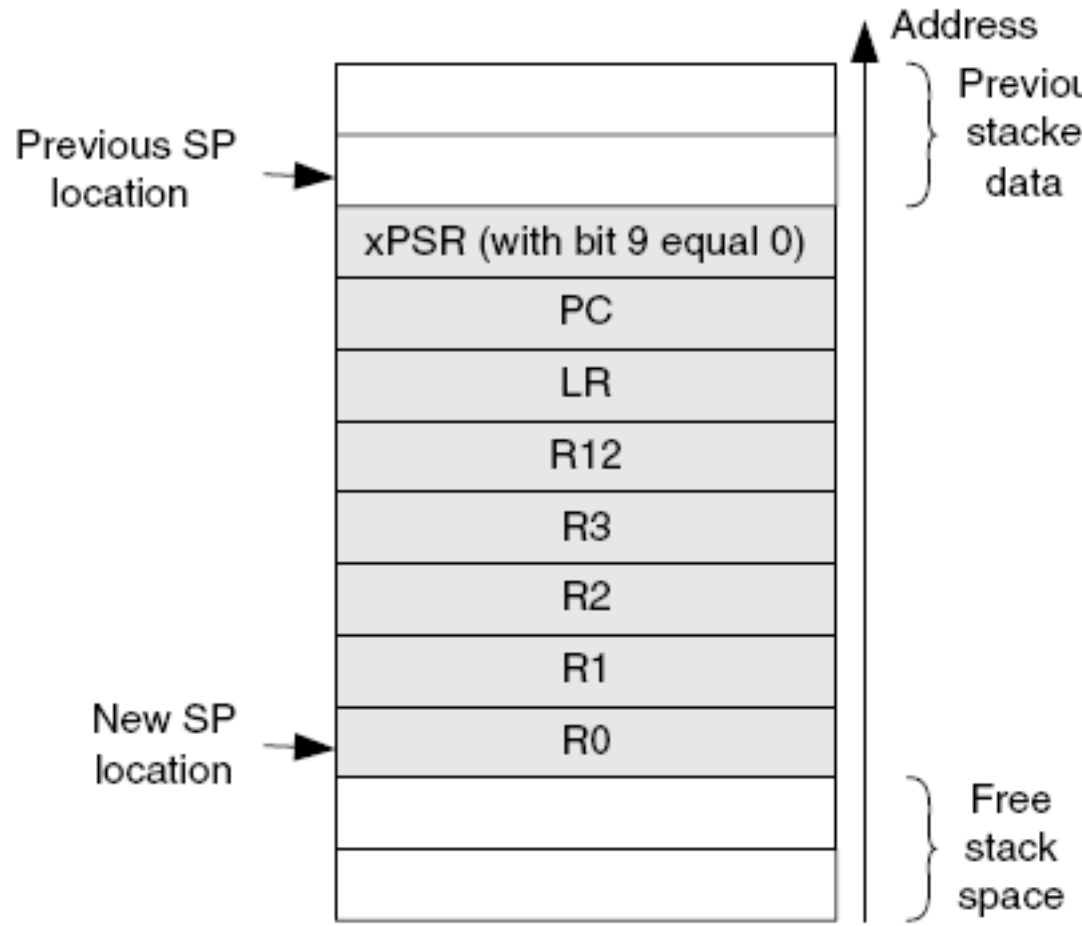
Implementación de un scheduler expropiativo con Cortex-M3

Repasando...


Reset
Inicialización
Configuración
Hardware de Entrada
Drivers
ISRs
UART
USB
GPIO
...
Tareas
Tarea:
flag?
variable?
cola?
Drivers
ISRs
UART
USB
GPIO
...
Hardware de Salida
Memoria de
Entrada
Cola
Flags
Arrays
...
Memoria de
Salida
Cola
Flags
Arrays
...

Cambio de contexto

- Sabemos que, al generarse una excepción/interrupción, el procesador guarda el contexto del programa actual en la pila:



Cambio de contexto

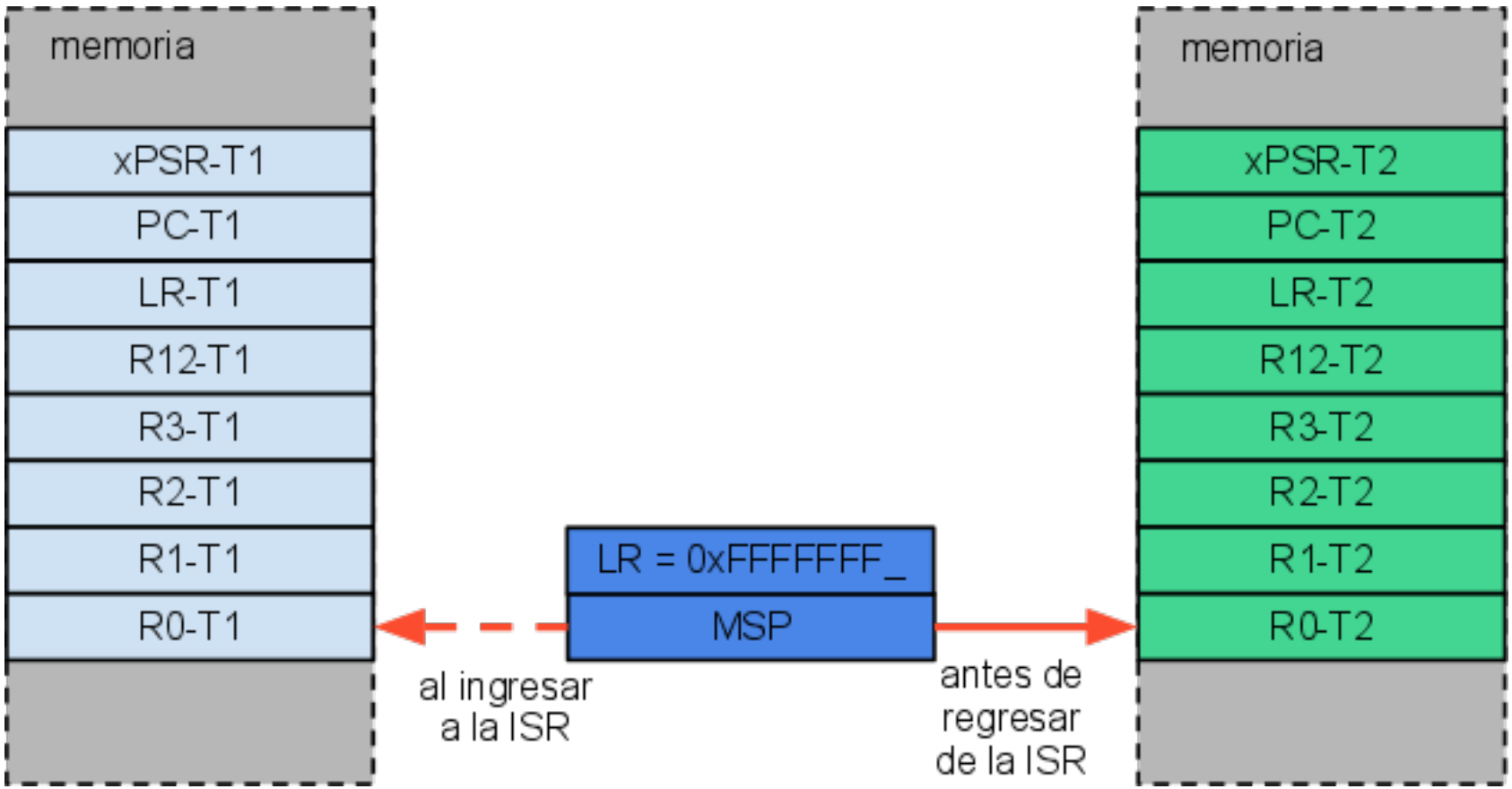
- Entonces, antes de regresar de la rutina de interrupción, podemos modificar el valor de **sp** para que apunte a otro contexto, distinto del de la tarea original.
- Para este ejemplo, usaremos solamente **msp** y sin pasar a modo no privilegiado, para simplificar el programa. FreeRTOS funciona así.
- Si deseamos salvar más registros en la pila además de los guardados por el procesador, deberemos usar **`push`** y **`pop`**.

- Cambiar a modo no privilegiado y el puntero de pila implica modificar **lr** antes de regresar de la ISR.

Value	Condition
0xFFFFFFF1	Return to handler mode
0xFFFFFFF9	Return to thread mode and on return use the main stack
0xFFFFFFFD	Return to thread mode and on return use the process stack

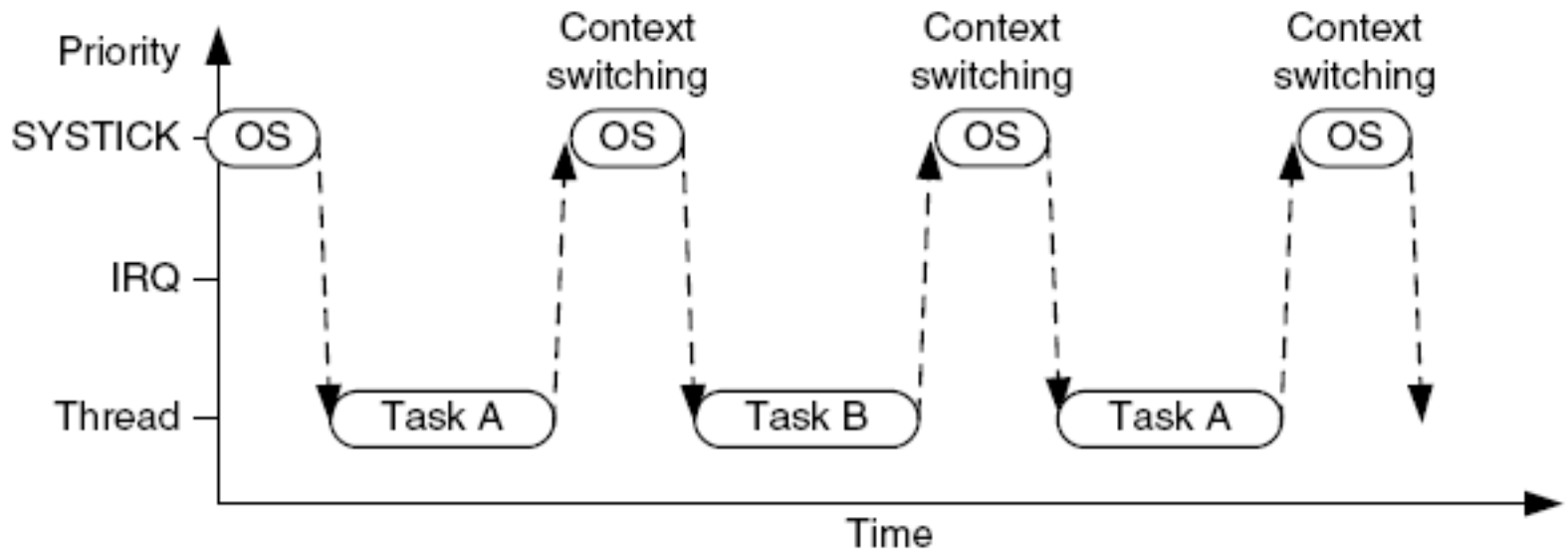
Cambio de contexto

Gráficamente:



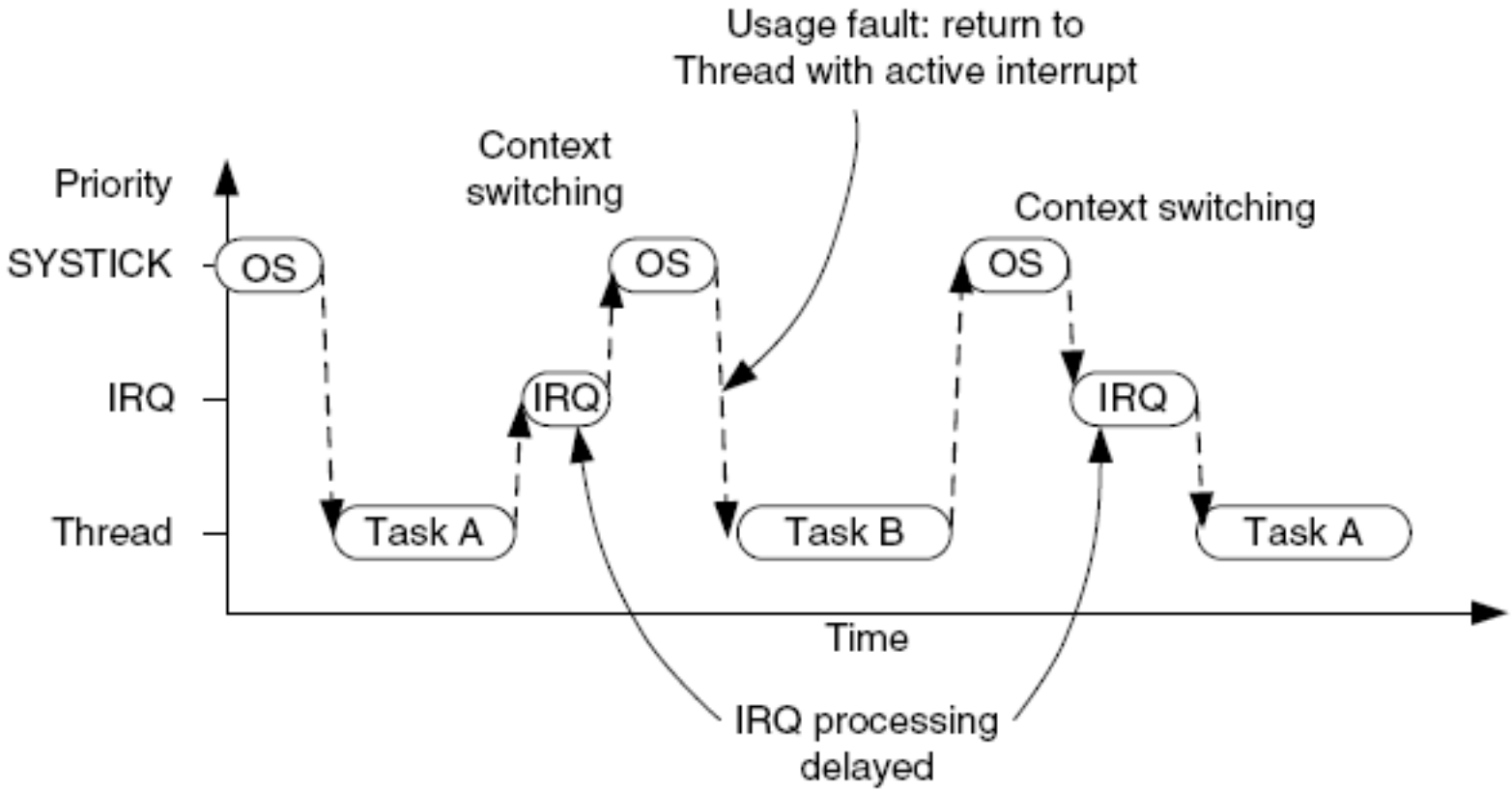
Cambio de contexto

- La interrupción que normalmente utilizaríamos para ejecutar el cambio de contexto, es la asociada a SysTick.



- Aunque esto podría traernos algunos inconvenientes:
  - ¿Puede la tarea forzar el cambio de contexto?
  - ¿Qué sucede si al interrumpir SysTick se está ejecutando otro handler de interrupción?

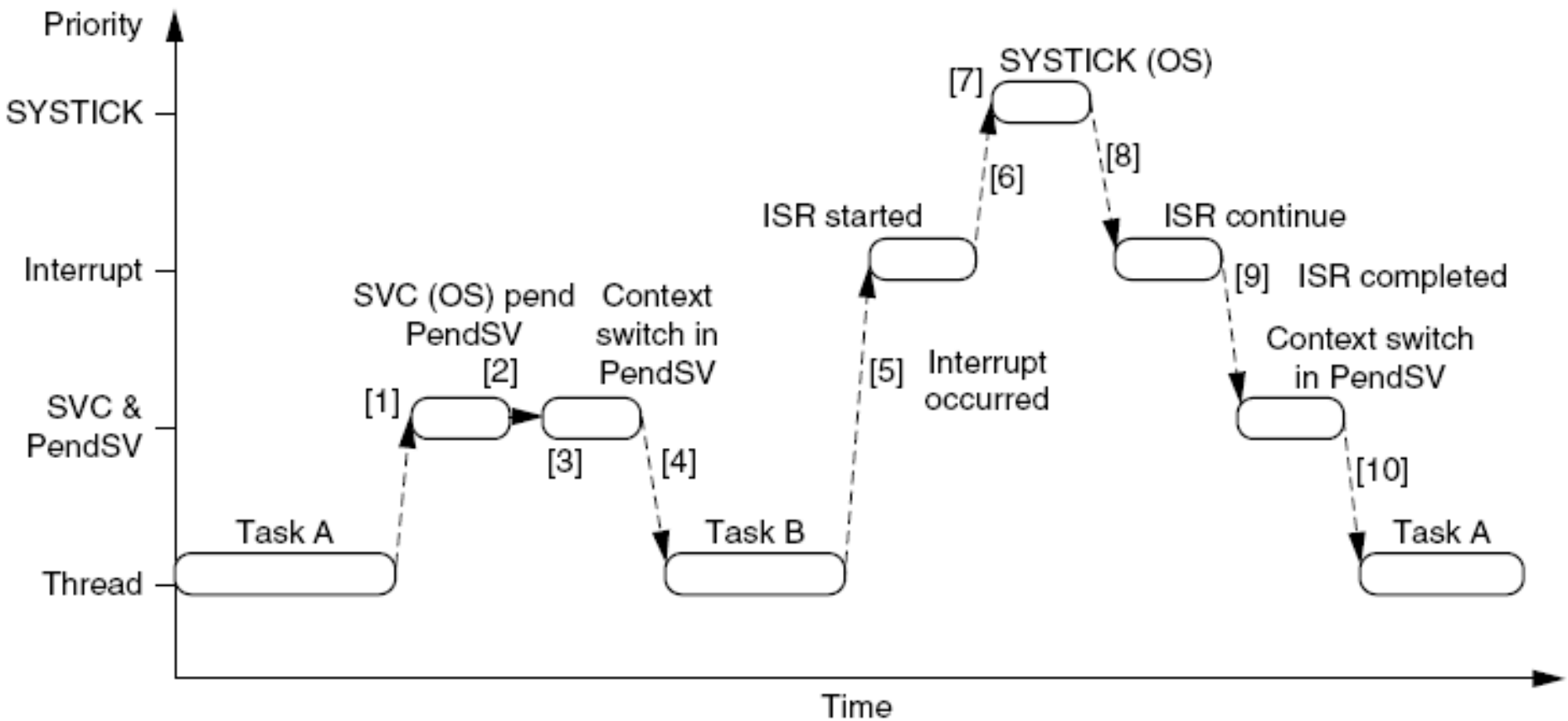
Cambio de contexto



- Tenemos que asegurar dos cosas:
  - Cambiar de contexto con ninguna ISR en espera.
  - Permitir a las tareas forzar el cambio de contexto para no desperdiciar tiempo del CPU.

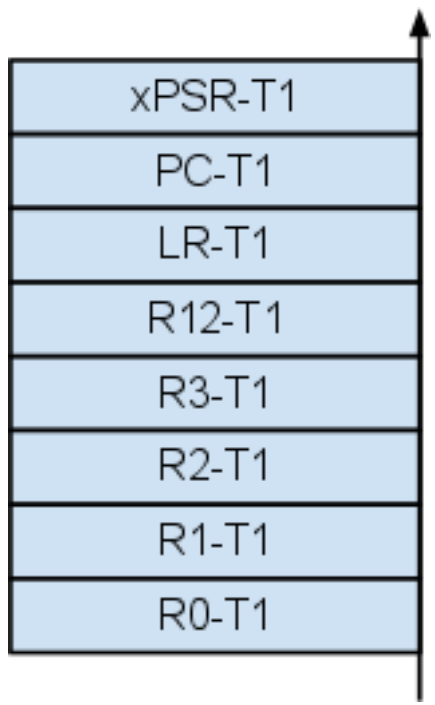
Cambio de contexto

- Para lograrlo haremos el cambio de contexto en el handler de PendSV (Pendable Service Call).
  - La programamos para que tenga la prioridad más baja, así los otros posibles handlers finalizarán su ejecución antes de que se lleve a cabo el cambio de contexto [5].
  - A su vez, si la tarea desea finalizar, basta con generar una interrupción por software para que el sistema operativo active PendSV [1].



Inicialización

- Recordar que codificaremos sólo el scheduler.
  - Para empezar, la lista de tareas será estática (no se agregarán nuevas tareas durante la ejecución).
  - Las tareas tendrán la misma **prioridad**: A,B,C,A,B,C,A,....
  - Configurar PendSV (context switch).
  - Configurar SysTick (monitoreo del sistema, llamar a PendSV).
  - Configurar una syscall SVC para ceder el CPU llamando a PendSV. En inglés se denomina "CPU yield".
- Inicialización de las pilas de cada tarea:
  - xPSR.24 = 1 (modo Thumb).
  - PC = taskMain (dirección de inicio).
  - Los registros restantes pueden ser cero.
  - Guardaremos en un vector los SP de cada tarea.



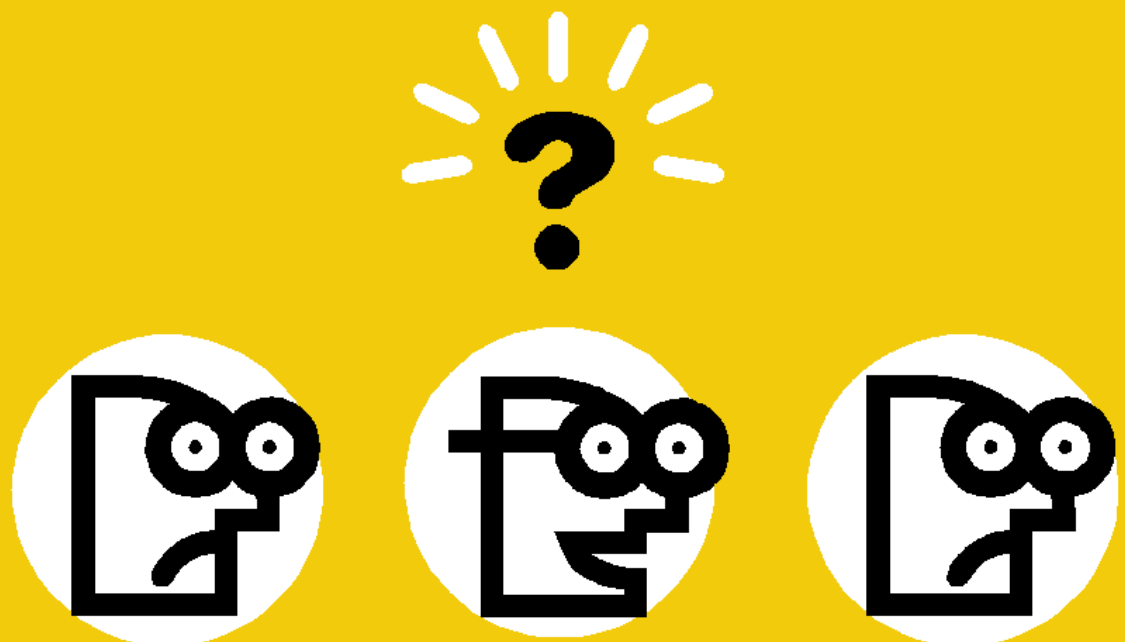
Ejemplo:

Inicialización de la pila de cada tarea:

```
void initTask(
    uint32_t * stack, /* vector reservado */
    uint32_t * sp, /* puntero (por referencia) */
    size_t stack_len,/*long. del vector (words)*/
    void (*task_main)(void) ) /* entry point */
{
    *sp = (uint32_t)(stack+stack_len-8);
    stack[stack_len-1] = 1<<24;
    stack[stack_len-2] = (uint32_t)task_main;
}
```

```
        .global initTask
#define pStack      r0
#define pSp         r1
#define stack_len   r2
#define stack_main r3
        .thumb_func
initTask:
        push {r4}
        lsl stack_len,stack_len,2
        mov r4,pStack
        add r4,stack_len
        sub r4,8*4
        str r4,[pSp]
        mov r4,1<<24
        sub stack_len,1*4
        str r4,[pStack,stack_len]
        sub stack_len,1*4
        str stack_main,[pStack,stack_len]
        pop {r4}
        bx lr
```

- **Cuidado** al utilizar funciones de librerías cuyo código desconocemos. Las mismas deben ser **reentrantes**.
  - Se considera que una función soporta reentrancia cuando no posee variables estáticas ni devuelve punteros a datos estáticos.
  - Recordar que las variables estáticas se almacenan en el área de variables globales. Al reentrar en la función en cuestión pero desde otra tarea, si la función posee datos estáticos, se sobreescriben los que estaba utilizando la tarea anterior, corrompiendo el contexto de la misma.
  - Las funciones que escribamos para nuestras tareas **no deben usar variables estáticas ni globales**.



It's QUESTION TIME!!

¡Manos a la obra!

Veamos un ejemplo usando LPCXpresso.