Sistemas Electrónicos Digitales

Tema #3

4. Interrupciones

Temario

1. Introducción
2. GPIO: General Purpose Input/Output
3. Arquitectura Arm Cortex-M4
4. Interrupciones
5. C en ensamblador
6. Temporizadores (Timers)
7. Direct Memory Access
8. Comunicaciones Serie
9. Conversores A/D y D/A

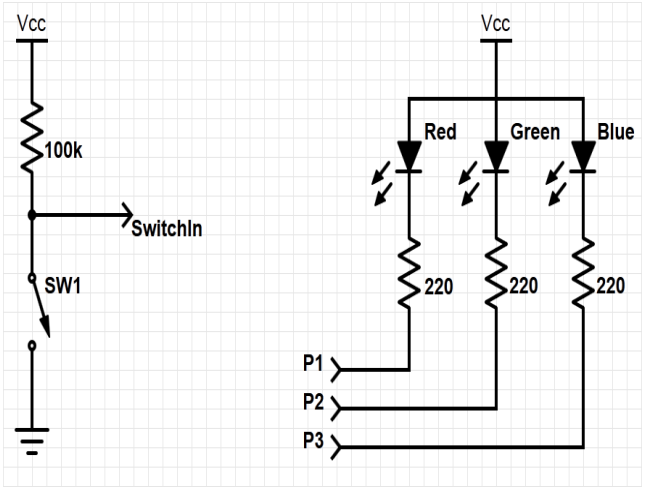
INDICE ESPECÍFICO

- **Conceptos de Excepción e Interrupción**
  - Entrando en una Excepción
  - Saliendo de una Excepción
- **El núcleo de las Interrupciones**
- **Diseño de programa con Interrupciones**
  - Compartiendo datos de forma segura entre ISRs y otros hilos.

INDICE ESPECÍFICO

- **Conceptos de Excepción e Interrupción**
  - Entrando en una Excepción
  - Saliendo de una Excepción
- **El núcleo de las Interrupciones**
- **Diseño de programa con Interrupciones**
  - Compartiendo datos de forma segura entre ISRs y otros hilos.

Estudio mediante un ejemplo



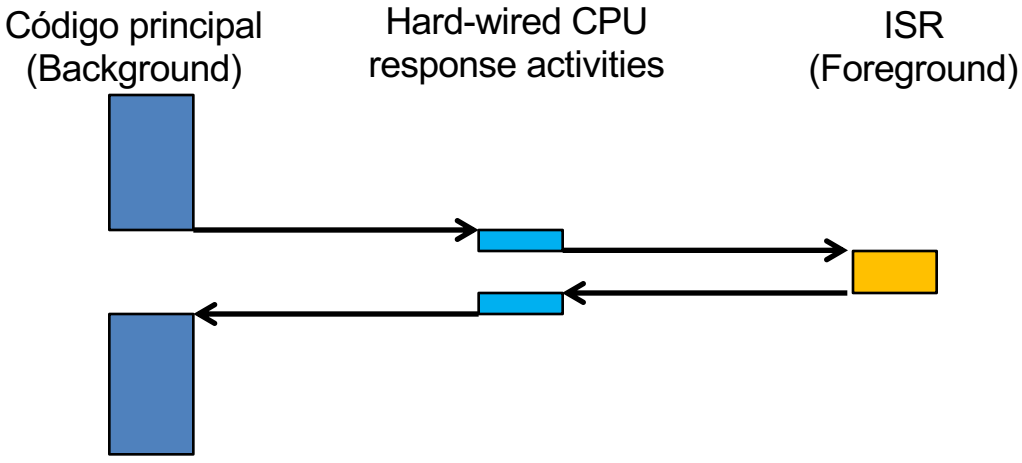
- Objetivo del ejemplo: cambiar el color del LED RGB de la placa de desarrollo cuando se presiona el pulsador sw1
- Los detalles de cómo implementarlo al completo se vieron en el apartado GPIO

¿Cómo detectar que el pulsador se ha presionado?

- **Polling** – comprobar en el bucle principal el estado del botón
  - **Lento**: es necesario comprobar explícitamente si el botón ha sido apretado.
  - **Desperdicio de tiempo de CPU:** Cuanto mas rápido necesitemos la respuesta más rápido tenemos que chequear el estado.
  - **Dificultad de escalado:** Es difícil construir un Sistema que tenga muchas actividades que tengan que ser atendidas con rapidez. El tiempo de respuesta depende del tiempo perdido en el resto de las tareas.

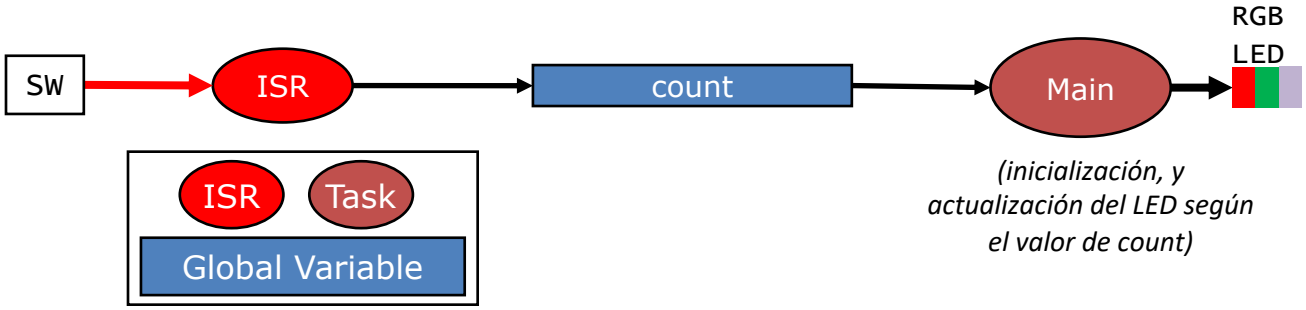
- **Interrupciones** – se usa hardware específico en el microcontrolador para detectar eventos y ejecutar código específico (**Interrupt Service Routine - ISR**) como respuesta
  - **Eficiente**: el código se ejecuta solo cuando es necesario
  - **Rápido**: es un mecanismo realizado por hardware.
  - **Fácilmente escalable**
    - El tiempo de respuesta del ISR no depende del resto de los procesos.
    - Los módulos se pueden programar con cierta independencia

Secuencia de procesado de una interrupción



- El código principal (background) se está ejecutando
- Se dispara la interrupción
- El procesador realiza algunas tareas “hard-wired” (digamos que son automáticas)
- El procesador ejecuta la ISR (foreground), incluyendo el procesamiento de vuelta de la interrupción al final.
- El procesador vuelve a la ejecución del código principal donde lo dejó

Ejemplo: requisitos y diseño



- Req1: Cuando se presiona el botón (SW), ISR incrementará la variable de cuenta (*count*)
- Req2: el código principal iluminará los LED de acuerdo con el valor de *count* en secuencia binaria (Azul: 4, Verde: 2, Rojo: 1)
- Req3: ISR tendrá prioridad (sobre el código principal) cuando se esté ejecutando

Procesado de una interrupción

- Vamos a estudiar el comportamiento del procesador ante una excepción

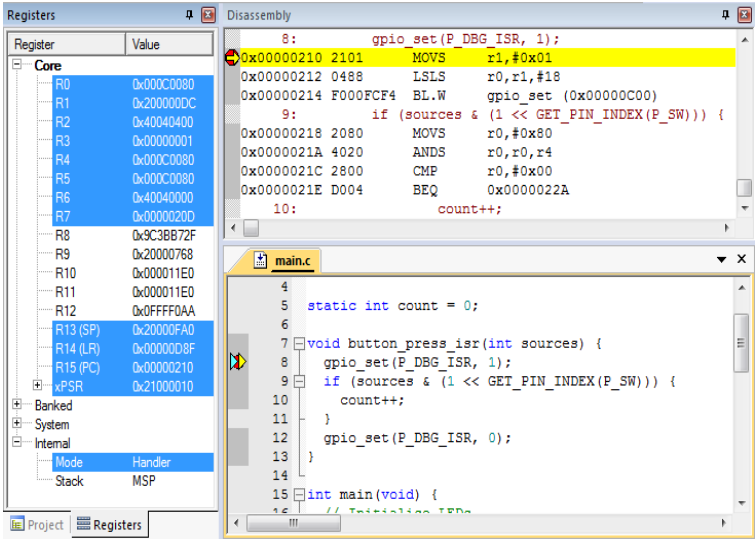
main.c

```
4
5   static int count = 0;
6
7   void button_press_isr(int sources) {
8     gpio_set(P_DBG_ISR, 1);
9     if (sources & (1 << GET_PIN_INDEX(P_SW))) {
10      count++;
11    }
12    gpio_set(P_DBG_ISR, 0);
13  }
14
15  int main(void) {
16    // Initialise LEDs.
17    leds_init();
18    leds_set(0, 0, 0);
19
20    // Set up debug signals.
21    gpio_set_mode(P_DBG_ISR, Output);
22    gpio_set_mode(P_DBG_MAIN, Output);
```

Uso del Debugger para el estudio

- Esta herramienta permite ver los valores de los registros, la pila, el código fuente y el código objeto (traducción del fuente a lenguaje ensamblador)

- Notas:
  - El compilador genera código para la entrada a la función de excepción.
  - Se ha de colocar el “breakpoint” en la declaración de la función, no en la primera línea.



INDICE ESPECÍFICO

- **Conceptos de Excepción e Interrupción**
  - **Entrando en una Excepción**
  - Saliendo de una Excepción
- **El núcleo de las Interrupciones**
- **Diseño de programa con Interrupciones**
  - Compartiendo datos de forma segura entre ISRs y otros hilos.

Entrando en una Interrupción

Secuencia del proceso de entrada de una interrupción:

1. Terminar la instrucción actual (excepto para instrucciones largas)
2. Salvar el contexto (8 registros de 32-bit) en la pila actual (MSP o PSP):
   - xPSR, Dirección de retorno, LR (R14), R12, R3, R2, R1, R0
3. Cambiar a modo privilegiado, usar MSP
4. Cargar el PC con la dirección de la rutina de excepción
5. Cargar LR con el valor EXC_RETURN
6. Cargar IPSR con el número de excepción
7. Empezar la ejecución de la rutina de excepción

Usualmente, este proceso (que es automáticamente ejecutado por hardware) lleva 16 ciclos desde la petición de interrupción hasta la ejecución de la primera sentencia de la rutina de interrupción

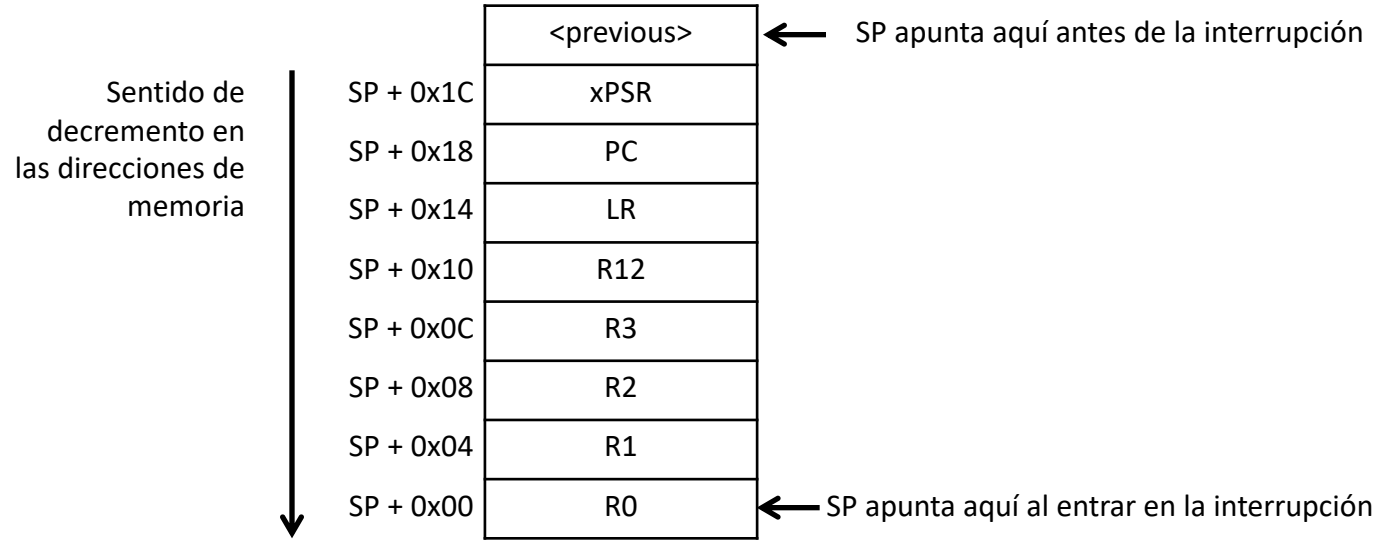
[Entrando en una Interrupción]

1. Terminar la instrucción actual

- La mayoría de las instrucciones son cortas y terminan rápidamente
- Sin embargo hay instrucciones que necesitan muchos ciclos de ejecución
  - Load Multiple (LDM), Store Multiple (STM), Push, Pop, MULS (32 ciclos en algunas CPUs basadas en la arquitectura)
- Estas instrucciones retrasarían la respuesta a la interrupción
- Si una de estas instrucciones se está ejecutando cuando entra una petición de interrupción, el procesador:
  - *abandona* la instrucción
  - responde a la interrupción
  - ejecuta la rutina de interrupción
  - retorna de la interrupción
  - *reinicia* la instrucción abandonada

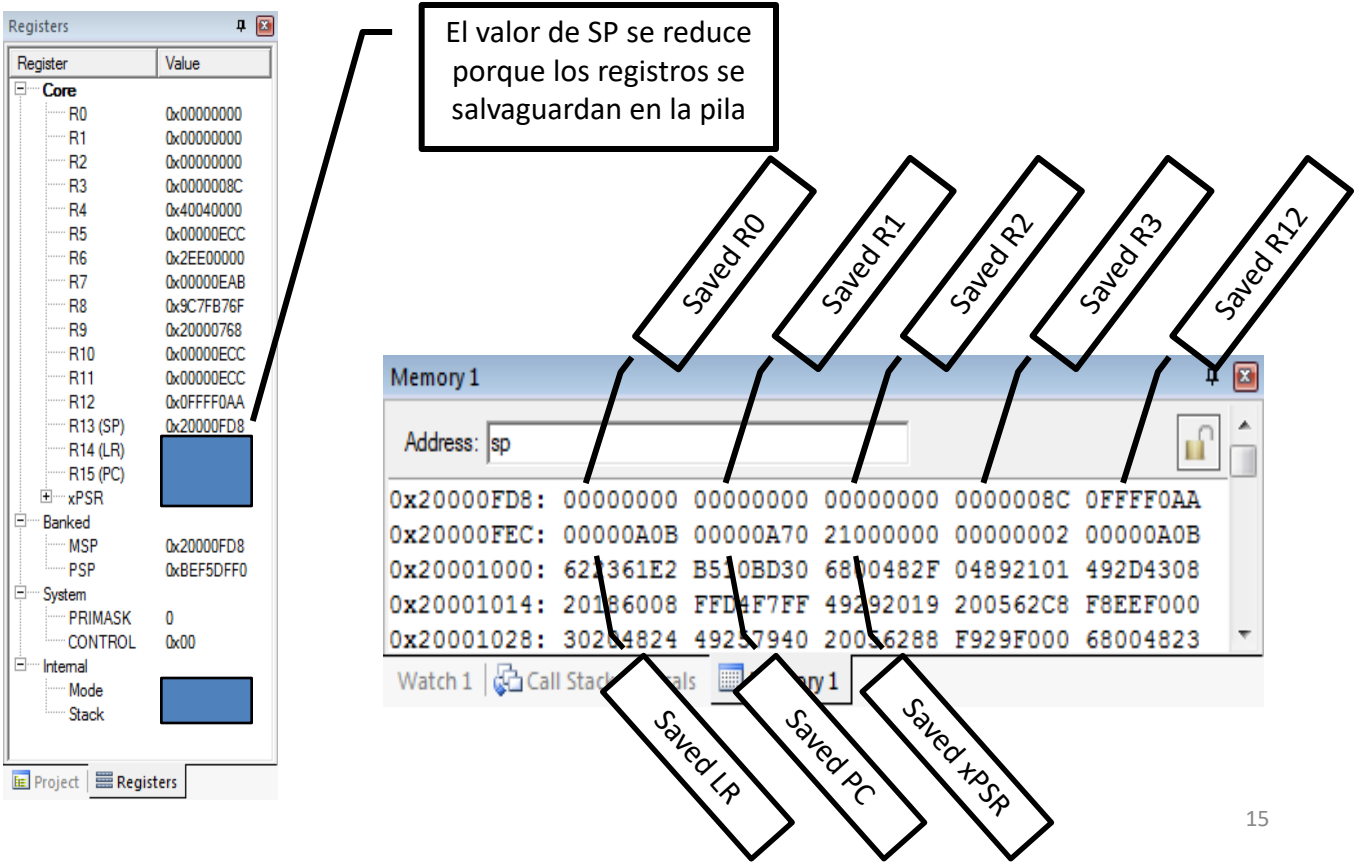
[Entrando en una Interrupción]

2. Salvar el contexto en la pila



La pila crece hacia direcciones inferiores de memoria

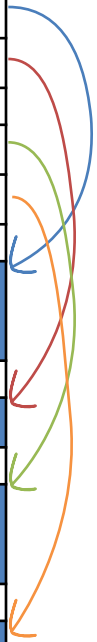
3. Cambiar a modo privilegiado



[Entrando en una Interrupción]

4. Cargar PC con dirección de rutina de excepción

Memory Address	Value
0x0000_0000	Initial Stack Pointer
0x0000_0004	Reset
0x0000_0008	NMI_IRQHandler
…	
	IRQ0_Handler
	IRQ1_Handler
…	
Reset:	
…	
NMI_IRQHandler:	
…	
IRQ0_Handler:	
…	
IRQ1_Handler:	

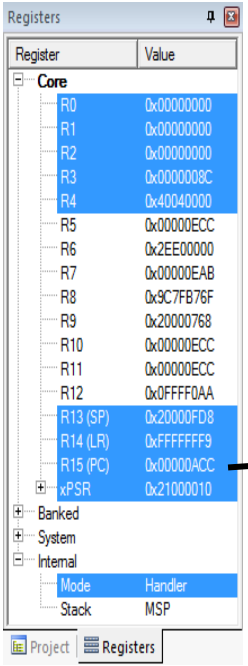


- El contador de programa se carga con la dirección de comienzo de la excepción correspondiente
- En memoria se muestra la tabla de vectores de excepción, en función de la interrupción concreta.

Tabla de excepciones

Exception number	IRQ number	Priority	Vector	Offset
			Initial SP	0x00
1		-3 (Highest)	Reset	0x04
2	-14	-2	NMI	0x08
3	-13	-1	HardFault	0x0C
4		Programable/ No disponible	Reserved	0x10
5				
6				
7				
8				
9				
10				
11	-5	Programable	SVCall	0x2C
12		Programable/ No disponible	Reserved	
13				
14	-2	Programable	PendSV	0x38
15	-1	Programable	SysTick	0x3C
16	0		IRQ0	0x40
17	1		IRQ1	0x44
18	2		IRQ2	0x48
.			.	
16+n	n		IRQn	0x40+4n

4. Cargar PC con dirección de rutina de excepción (II)

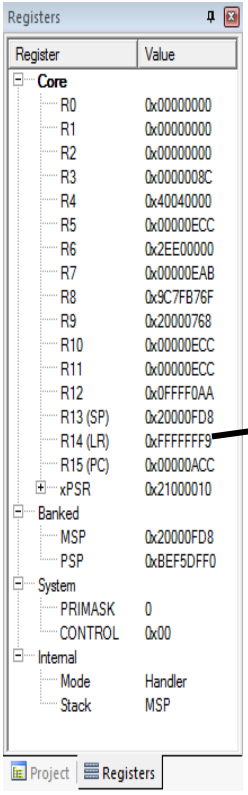



```
    42: void switch_isr(void) {
0x00000ACC B510      PUSH     {r4,lr}
```

PC se carga con la dirección inicial de la rutina de interrupción

[Entrando en una Interrupción]

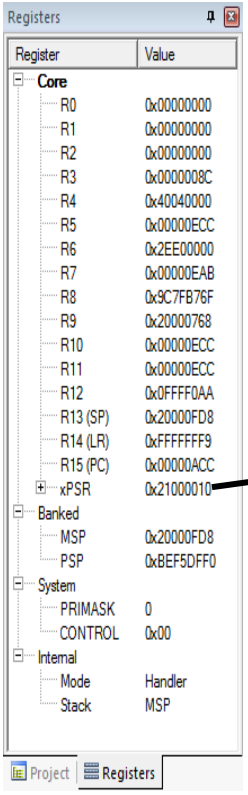
5. Cargar LR con el retorno de excepción



El valor 0xFFFFFFF9 es un caso especial

[Entrando en una Interrupción]

6. Cargar IPSR con el número de excepción



Exception number 0x10
(interrupt number + 0x10)

[Entrando en una Interrupción]

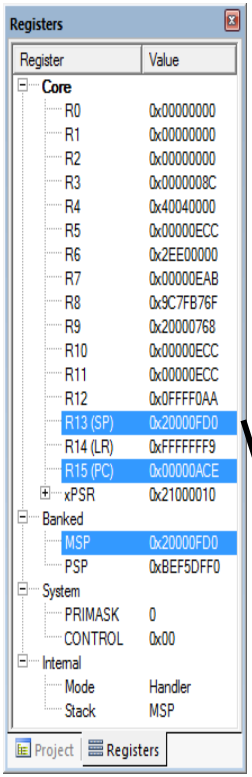
7. Empezar la ejecución de la rutina

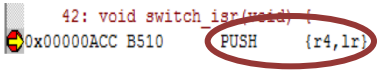
- La rutina de excepción se inicia, a no ser que se vea interrumpida por una excepción de mayor prioridad.

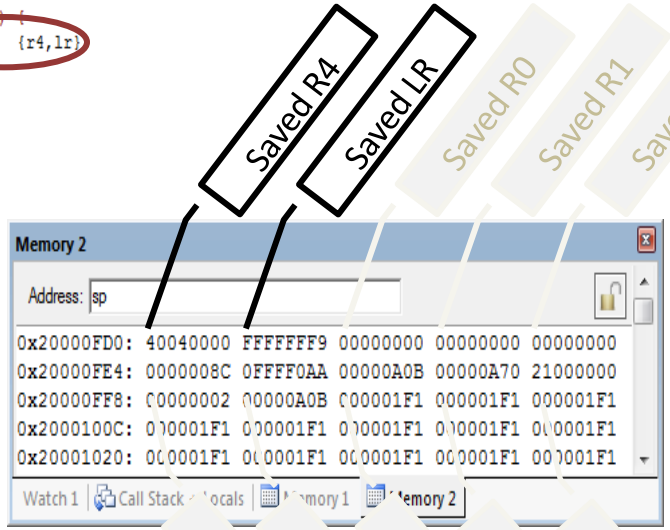
- La rutina puede salvaguardar registros adicionales en la pila
  - Por ejemplo, si la rutina puede llamar a una subrutina, LR y R4 deben salvaguardarse

```
    42: void switch_isr(void) {
0x00000ACC B510      PUSH     {r4,lr}
```

Salvaguardando los registros.







SP value reduced since registers have been pushed onto stack

INDICE ESPECÍFICO

- **Conceptos de Excepción e Interrupción**
  - **Entrando en una Excepción**
  - **Saliendo de una Excepción**
- **El núcleo de las Interrupciones**
- **Diseño de programa con Interrupciones**
  - Compartiendo datos de forma segura entre ISRs y otros hilos.

Saliendo de una Interrupción

Secuencia del proceso de salida de una interrupción:

1. Ejecutar la instrucción que dispara el proceso de retorno de excepción.
2. Seleccionar la pila de retorno (MSP o PSP); restaurar el contexto (8 32-bit words) de la pila actual.
3. Reanudar la ejecución del código interrumpido cuando se lanzó la excepción.

[Saliendo de una Interrupción]

1. Inicio del proceso de retorno

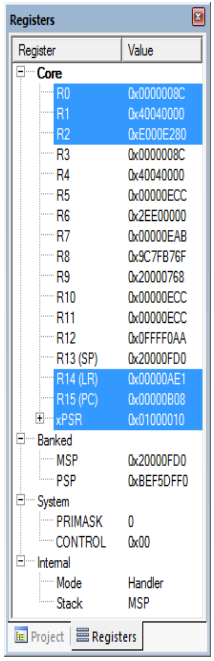
- No existe una instrucción tal como “return from interrupt”.
- En su lugar se usa:
  - **BX LR :** Salto a la dirección almacenada en LR (carga el PC con el contenido de LR)
  - **POP …, PC** – Restaura los valores desde la pila, entre ellos el PC
- … con un valor especial **EXC_RETURN**, que al cargarse en el PC dispara el proceso de retorno de la excepción
  - BX LR se usa si EXC_RETURN está en LR
  - Si EXC_RETURN se ha guardado en la pila, entonces se usa POP

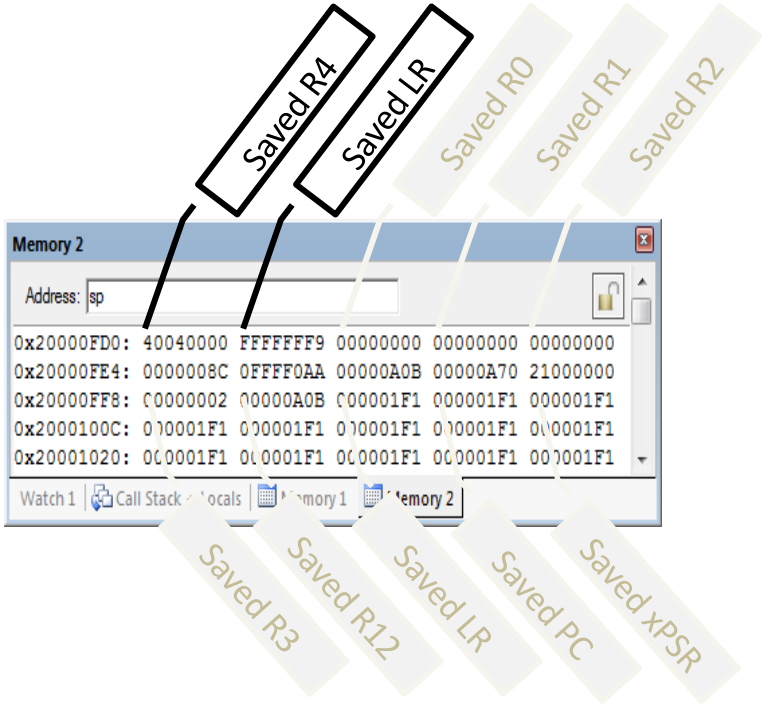
```
    51: }
0x00000B08 BD10      POP      {r4,pc}
```

1. Proceso de retorno. PC desde la pila

- R4: 0x4040_0000
- PC: 0xFFFF_FFF9

```
    51: }
0x00000B08 BD10      POP      {r4,pc}
```





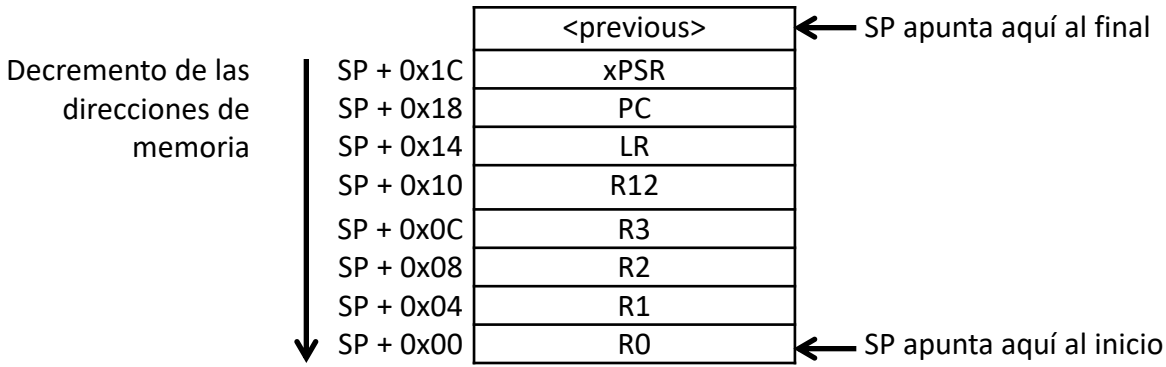
[Saliendo de una Interrupción]

2. Selección de la pila, restaurar contexto

- Dependiendo del valor de EXC_RETURN se determina la pila de trabajo y el proceso de retorno

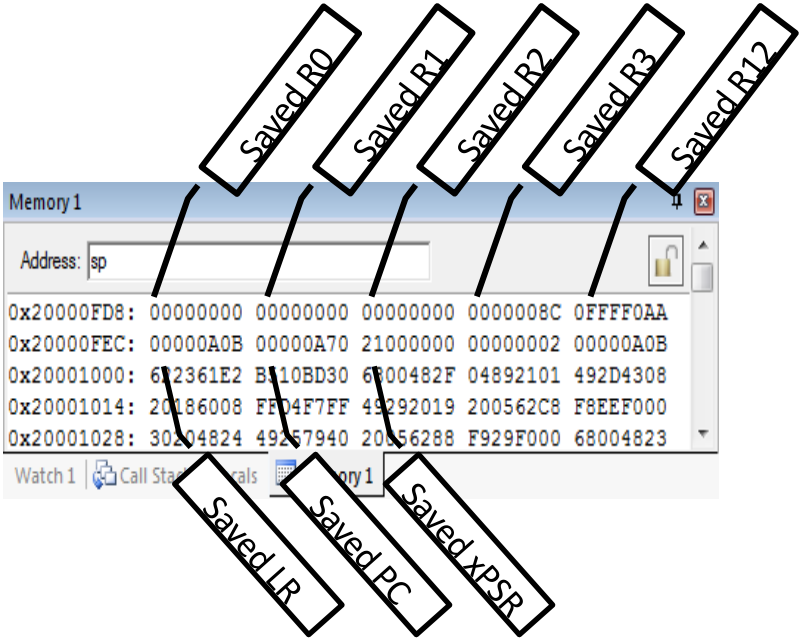
EXC_RETURN	Return Stack	Description
0xFFFF_FFF1	0 (MSP)	Return to exception handler with MSP
0xFFFF_FFF9	0 (MSP)	Return to thread with MSP
0xFFFF_FFFD	1 (PSP)	Return to thread with PSP

- Y se restauran los valores correctos de la pila seleccionada



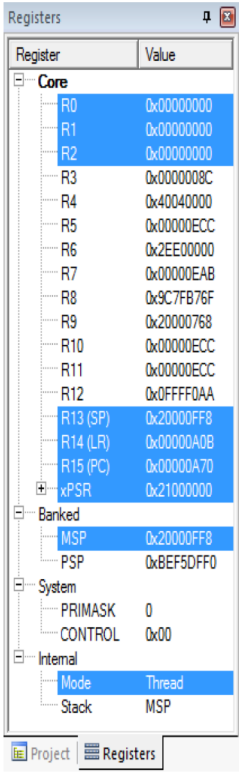
2. Siguiendo con el ejemplo…

- PC=0xFFFF_FFF9, luego se utiliza la pila principal (MSP) y se retorna al hilo principal del programa de donde se salió.
- Se restauran automáticamente los vaolres almacenados en la pila a los registros.



[Saliendo de una Interrupción]

3. Reanudar la ejecución interrumpida



- Los registros que salvaguardó el procesador de forma automática al iniciarse la interrupción se restauran con su valor: R0, R1, R2, R3, R12, LR, PC, xPSR
- SP vuelve a tener el valor original
- Se vuelve al modo de trabajo hilo
- La siguiente instrucción a ejecutar es la situada en 0x0000_0A70

INDICE ESPECÍFICO

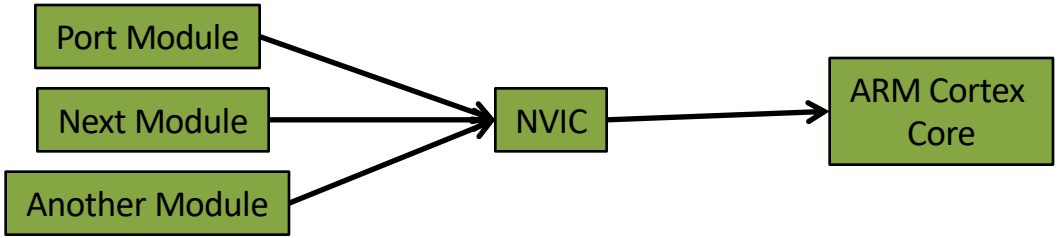
- **Conceptos de Excepción e Interrupción**
  - Entrando en una Excepción
  - Saliendo de una Excepción
- **El núcleo de las Interrupciones**
- **Diseño de programa con Interrupciones**
  - Compartiendo datos de forma segura entre ISRs y otros hilos.

Tipos de interrupciones

- **Interrupciones Hardware**
  - Asíncronas, no están relacionadas con el código que el procesador ejecuta en ese instante.
  - Ejemplos: se recibe un carácter por el puerto serie, o se finaliza una conversión A/D.
- **Excepciones, fallos e interrupciones software**
  - Síncronas, son el resultado de la ejecución de una instrucción
  - Ejemplos: se produce desbordamiento en una operación, acceso a una posición de memoria inexistente.
- Podemos habilitar y deshabilitar la mayoría de las interrupciones (enmascarables), aunque algunas no (no enmascarables).
- Después de completar la rutina de interrupción, el procesador retorna al código interrumpido.

El Controlador de interrupciones (NVIC Nested Vectored Interrupt Controller)

- NVIC es el subsistema que maneja y prioriza las interrupciones externas
- Las interrupciones son un tipo de excepciones
  - Excepciones <16, Interrupciones: 16 a 16+N
- Modos
  - Modo hilo: entra con un Reset
  - Modo rutina: entra al ejecutar una excepción
- Se entra en un modo de ejecución privilegiado
- Punteros de pila
  - Main Stack Pointer, MSP
  - Process Stack Pointer, PSP
- Estados de excepción: Inactivo, Pendiente, Activo, Activo y pendiente



Registros y estado del Controlador de interrupciones

- **Habilitado** – Permiten que se reconozcan las interrupciones
  - Se acceda a través de dos registros (bits a “1” para habilitar)
    - *Set enable con* NVIC_ISER, *clear enable* con NVIC_ICER
  - Funciones con CMSIS:
    NVIC_EnableIRQ(IRQnum),
    NVIC_DisableIRQ(IRQnum)

- **Pendiente** – la interrupción ha sido solicitada pero aún no se ha permitido su comienzo
  - Funciones con CMSIS:
    NVIC_SetPendingIRQ(IRQnum), NVIC_ClearPendingIRQ(IRQnum)

Registros de máscara de interrupciones (Core Exception Mask Register)

- Similar al bit “Global interrupt disable” en otros micros.
- PRIMASK - Exception mask register (núcleo CPU)
  - Bit 0: PM Flag
    - “0” permite la activación de todas las excepciones
    - “1” evita la activación de todas las excepciones con prioridad configurable
  - Se modifica con las instrucciones CPS, MSR y MRS
  - Se usa para prevenir condiciones de carrera
- Funciones en CMSIS
  - void __enable_irq() – pone a cero el PM flag
  - void __disable_irq() – pone a uno el PM flag
  - uint32_t __get_PRIMASK() – devuelve el valor de PRIMASK
  - void __set_PRIMASK(uint32_t x) – pone PRIMASK a x

Priorización

- Las excepciones son priorizadas con un número para poder arbitrar la respuesta ante peticiones de atención simultaneas
  (menor número = mayor prioridad)
- La prioridad de algunas excepciones es fija
  - Reset: -3, la mayor prioridad
  - NMI: -2
  - Hard Fault: -1
- La prioridad de otras excepciones (periféricos) es ajustable
  - Su valor se guarda en el “interrupt priority register” (IPR0-7)
  - Ej.: 0x00, 0x40, 0x80, 0xC0

Casos aplicados de priorización

- **Caso A**: Peticiones simultaneas de excepciones
  - Se atiende primero a la excepción con menor número de prioridad (interrupción más prioritaria)
- **Caso B:** Nueva petición de excepción mientras se está ejecutando la rutina de atención de otra
  - La prioridad de la nueva es **mayor** que la de la antigua
    - La nueva rutina se ejecuta y la antigua ha de esperar a que la nueva acabe
  - La prioridad de la nueva es **menor o igual** que la de la antigua
    - La nueva excepción queda en estado pendiente
    - La antigua rutina se ejecuta y acaba
    - Se restaura el anterior nivel de prioridad
    - La nueva excepción se ejecuta si el nivel de prioridad lo permite

Cómo priorizar las interrupciones

- El uso de las librerías **CMSIS** (Cortex Microcontroller Software Interface Standard) proporcionadas por ARM es la forma recomendada para programar los microcontroladores Cortex-M de forma que pueda portarse el programa entre dispositivos.

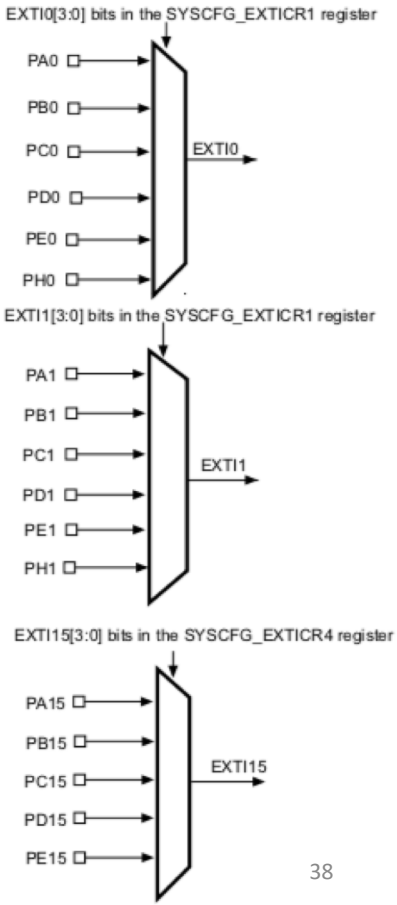
- CMSIS proporciona la función:
  NVIC_SetPriority(IRQn, priority)
  para establecer la prioridad de las interrupciones.

- STM32 HAL proporciona llamadas para esta función

```
/* EXTI interrupt init*/
HAL_NVIC_SetPriority(EXTI0_IRQn, 0, 0);
```

Tipos de interrupciones externas (STM32F4)

- Podemos usar 16 líneas de interrupción externas (línea 0 a línea 15) para detectar eventos externos de los pines GPIO. Cada pin de cada puerto GPIO está conectado a una línea de interrupción externa con el mismo número.
- Ejemplo:
  - PA0, PB0, PC0, etc., se multiplexan a line0. Por lo tanto, solo puede configurar uno de estos pines para conectarse a la línea de interrupción 0.
  - PA0 y PA8 se multiplexan en diferentes líneas. Por lo que puede configurar estos pines para utilizar interrupciones externas al mismo tiempo. Porque PA0 está conectado a line0 y PA8 está conectado a line8.



Tipos de interrupciones externas (STM32F4) (II)

- Para los manejadores (**handlers**) de interrupciones, STM32F4 tiene solo 7 manejadores de interrupciones para manejar las interrupciones externas de la línea 0 a la línea 15.
- Solo interrupciones externas de la line0 a line4 tienen su propio controlador IRQ.
- Line5 - line9 y line10 - line15 tienen el mismo controlador IRQ.

IRQ	IRQ Handler
EXTI0_IRQn	EXTI0_IRQHandler
EXTI1_IRQn	EXTI1_IRQHandler
EXTI2_IRQn	EXTI2_IRQHandler
EXTI3_IRQn	EXTI3_IRQHandler
EXTI4_IRQn	EXTI4_IRQHandler
EXTI9_5_IRQn	EXTI9_5_IRQHandler
EXTI15_10_IRQn	EXTI15_10_IRQHandler

INDICE ESPECÍFICO

- **Conceptos de Excepción e Interrupción**
  - Entrando en una Excepción
  - Saliendo de una Excepción
- **El núcleo de las Interrupciones**
- **Diseño de programa con Interrupciones**
  - Compartiendo datos de forma segura entre ISRs y otros hilos.

INDICE ESPECÍFICO

- **Conceptos de Excepción e Interrupción**
  - Entrando en una Excepción
  - Saliendo de una Excepción
- **El núcleo de las Interrupciones**
- **Diseño de programa con Interrupciones**
  - **Compartiendo datos de forma segura entre ISRs y otros hilos.**

Cuestiones a tener en cuenta

- ¿Cuanta carga de trabajo asignar a una rutina de interrupción?¿Puede ser muy extensa?
- ¿Debería la rutina habilitar las interrupciones?
- ¿Cómo comunicar una rutina de interrupción con otros hilos del programa?
  - Datos compartidos (“data buffering”)
  - Integridad de los datos y condiciones de carrera (“race conditions”)

¿Cuanta carga de trabajo asignar a una rutina de interrupción?

Importante: la atención inmediata de las interrupciones retrasa la ejecución del resto del programa

- Cuando tenemos varias rutinas de interrupción, o una rutina se ejecuta repetidamente, es conveniente realizar el trabajo crítico en la rutina de atención a la interrupción y almacenar los resultados parciales para un procesado posterior

Compartiendo datos de forma segura entre ISRs y otros hilos.

- Tipo "**Volatile data**" – se trata de un tipo de dato que puede ser actualizado en cualquier momento y sin control del programa que se ejecuta

- "**Non-atomic shared data**" – se trata de un tipo de dato que contiene múltiples campos. En ejecución se puede interrumpir su actualización, escritura o lectura, por lo que es vulnerable a las condiciones de carrera

Datos volátiles (“Volatile”)

- **Los compiladores asumen que las variables en la memoria no cambian espontáneamente y se optimizan en base a esa creencia**
  - No vuelven a cargar una variable de la memoria si la función actual no la ha cambiado
  - Leen variable de memoria en registro (acceso más rápido)
  - Escriben de nuevo en la memoria al final del procedimiento, o antes de una llamada a un procedimiento, o cuando el compilador se quede sin registros libres
- **Esta optimización puede fallar**
  - Ejemplo: leer un puerto de entrada, detectando una pulsación de una tecla
    - while (SW_0); leerá de SW_0 una vez y reutilizará ese valor
    - Generará un bucle infinito activado por SW_0 siempre verdadero
- Razones por las que puede fallar:
  - Registro de periféricos mapeado en memoria -> **registros cambian por sí solos**
  - Variables globales modificadas por un ISR -> **ISR cambia la variable**
  - Variables globales en una aplicación multiproceso -> **otro subproceso** o ISR **cambia la variable**

The Volatile Directive

- Hay que decirle al compilador qué variables pueden cambiar fuera de su control. Para ello:
  - Utilizar la palabra reservada "**volatile**" para forzar al compilador a recargar esta variable de la memoria en cada uso:

```
volatile unsigned int num_ints;
```

❖ Puntero a un volatile int

```
volatile int * var; // or
int volatile * var;
```

  - Ahora, cada lectura de una variable (por ejemplo, registro de estado) dará como resultado una instrucción LDR en lenguaje ensamblador

Buena explicación en "Volatile" de Nigel Jones, Programación de sistemas integrados, julio de 2001

Non-Atomic Shared Data

- Queremos hacer un seguimiento de la fecha y hora
- Usamos interrupciones con temporizador de 1Hz
- Sistema
  - La estructura current_time rastrea el tiempo y los días desde algún evento de referencia
  - los campos de current_time se actualizan mediante la ISR y el temporizador periódico de 1 Hz

```
void GetDateTime(DateTimeType * DT){
 DT->day = current_time.day;
 DT->hour = current_time.hour;
 DT->minute = current_time.minute;
 DT->second = current_time.second;
}
```

```
void DateTimeISR(void){
 current_time.second++;
 if (current_time.second > 59){
   current_time.second = 0;
   current_time.minute++;
   if (current_time.minute > 59) {
     current_time.minute = 0;
     current_time.hour++;
     if (current_time.hour > 23) {
      current_time.hour = 0;
       current_time.day++;
       … etc.
     }
```

Ejemplo: comprobando la hora/fecha

- **Problema:**
  - Una interrupción en el momento equivocado provocará datos semi actualizados en DT
- Ejemplo de fallo:
  - current_time es {10, 23, 59, 59} (día 10, 23:59:59)
  - El código de la tarea llama a GetDateTime (), que comienza a copiar los campos current_time en DT: día = 10, hora = 23
  - Se produce una interrupción del temporizador, que actualiza current_time a {11, 0, 0, 0}
  - GetDateTime () reanuda la ejecución, copiando los campos restantes de current_time a DT: minuto = 0, segundo = 0
  - DT ahora tiene una marca de tiempo de {10, 23, 0, 0}.
  - ¡El sistema piensa que el tiempo acaba de saltar una hora!
- Problema fundamental: ”**race condition**"
  - La prioridad permite a la ISR interrumpir otro código y posiblemente sobrescribir datos
  - **Se debe garantizar el acceso atómico (indivisible) al objeto.**
    - El tamaño del objeto atómico nativo depende del conjunto de instrucciones del procesador y del tamaño de la palabra.
    - Tiene 32 bits para ARM

El problema en más detalle

- Se debe proteger cualquier objeto de datos que:
  - Requiere múltiples instrucciones para leer o escribir (acceso no atómico), y
  - Puede ser potencialmente escrito por una ISR

- ¿Cuántas tareas / ISR pueden escribir en el objeto de datos?
  - ¿Una? Entonces tenemos comunicación unidireccional
    - Debe asegurarse de que los datos no se sobrescriban a la mitad de la lectura.
      - El escritor y el lector no se interrumpen
  - ¿Más de una?
    - Debe asegurarse de que los datos no se sobrescriban a la mitad de la lectura.
      - El escritor y el lector no se interrumpen
    - Debe asegurarse de que los datos no se sobrescriban en parte a través de la escritura
      - Los escritores no se interrumpen

Definiciones

- **Condición de carrera (Race Condition)**: Comportamiento anómalo debido a una dependencia crítica inesperada de la temporización de los eventos. El resultado del código de ejemplo depende de la sincronización relativa de las operaciones de lectura y escritura.

- **Sección crítica (Critical section)**: una sección de código que crea una posible condición de carrera. La sección de código solo puede ser ejecutada por un proceso a la vez. Se requiere algún mecanismo de sincronización en la entrada y salida de la sección crítica para garantizar el uso exclusivo.

Solución: Desactivar prioridades (preemption)

- Prevenir las prioridades dentro de la sección crítica
- Si un ISR puede escribir en el objeto de datos compartidos, se deben **deshabilitar las interrupciones**
  - ✓ guardar el estado de máscara de interrupción actual en m
  - ✓ desactivar interrupciones
- Restaurar el estado anterior posteriormente (es posible que las interrupciones ya se hayan deshabilitado por otro motivo)
- Utilice CMSIS-CORE para guardar, controlar y restaurar el estado de máscara de interrupción
- Evitar si es posible:
  - Deshabilitar interrupciones retrasa la respuesta a todas las demás solicitudes de procesamiento
  - Haga este tiempo lo más corto posible (por ejemplo, unas pocas instrucciones)

```
void GetDateTime(DateTimeType * DT)
{
 uint32_t m;

 m = __get_PRIMASK();
 __disable_irq();

 DT->day = current_time.day;
 DT->hour = current_time.hour;
 DT->minute = current_time.minute;
 DT->second = current_time.second;
 __set_PRIMASK(m);
}
```

Resumen para datos compartidos

- En threads (hilos) y ISR, identifique los datos compartidos
- Determine qué datos compartidos son demasiado grandes para ser manejados atómicamente por defecto
  - Esto debe protegerse de la priorización (por ejemplo, deshabilitar las interrupciones, usar un mecanismo de sincronización RTOS)
- Declarar (e inicializar) las variables compartidas como volátiles en el archivo principal (o globals.c)
  - `volatile int my_shared_var=0;`
- Actualice extern.h para que estas variables estén disponibles para funciones en otros archivos.
  - `volatile int my_shared_var;`
  - `#include “extern.h”` en cada archivo que use las variables compartidas
- Cuando utilice datos compartidos largos (no atómicos), guarde, deshabilite y restaure el estado de máscara de interrupción
  - CMSIS-CORE interface: `__disable_irq(), __get_PRIMASK(), __set_PRIMASK()`

Recursos recomendados

- Architecture Reference Manual:

  http://infocenter.arm.com/help/index.jsp

- Cortex-M4 Technical Reference Manual:

  http://infocenter.arm.com/help/topic/com.arm.doc.ddi0439d/DDI0439D_cortex_m4_processor_r0p1_trm.pdf

- Cortex-M4 Devices Generic User Guide:

  http://infocenter.arm.com/help/topic/com.arm.doc.dui0553a/DUI0553A_cortex_m4_dgug.pdf