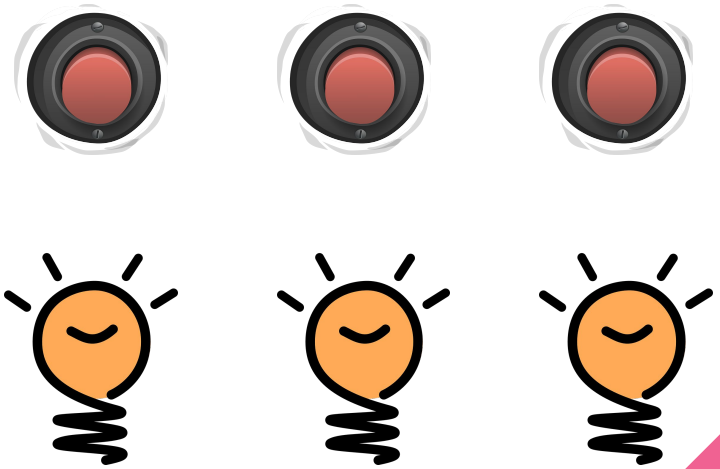
Interrupciones

Informática II - R2004
2021

Recordando...

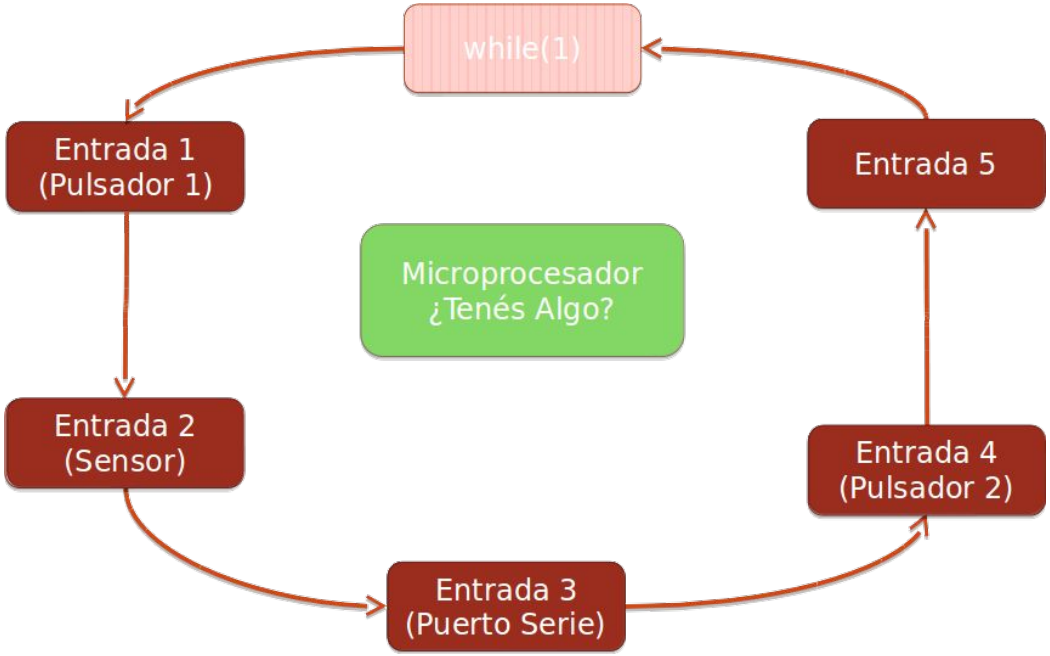
¿Cómo hacemos para que el microcontrolador esté siempre pendiente de lo que está pasando?

```
while ( 1 )
{
        if (LeerBoton1() == 1)
                EncenderLampara1();
        else
                ApagarLampara1();


         if (LeerBoton2() == 1)
                 EncenderLampara2();
         else
                 ApagarLampara2();
}
```

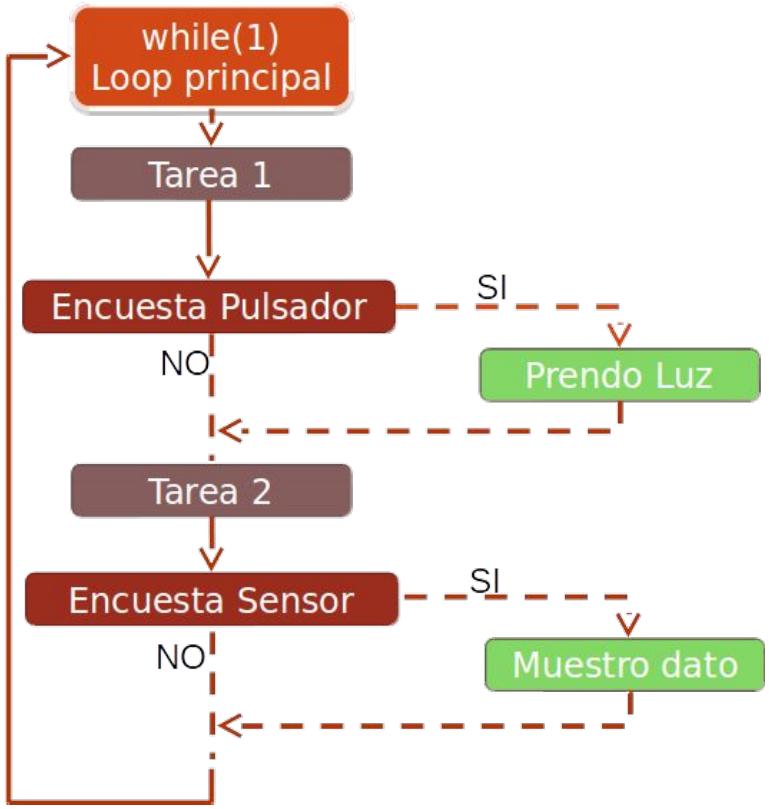


Entonces… ¿Cómo venimos programando?

```
void main ( void )
{
        Inicializar();

        while ( 1 ) {

                /* ... */

                if ( Pulsador() == TRUE )

                /* ... */

                if ( Sensor() == FALSE )

                /* ... */

                if ( Temp() == TEMP_MAX )

                /* ... */

                if ( DatoSerie() == ERROR )

                /* ... */

        }
}
```



Pooling - Atención SINCRÓNICA de las E/S



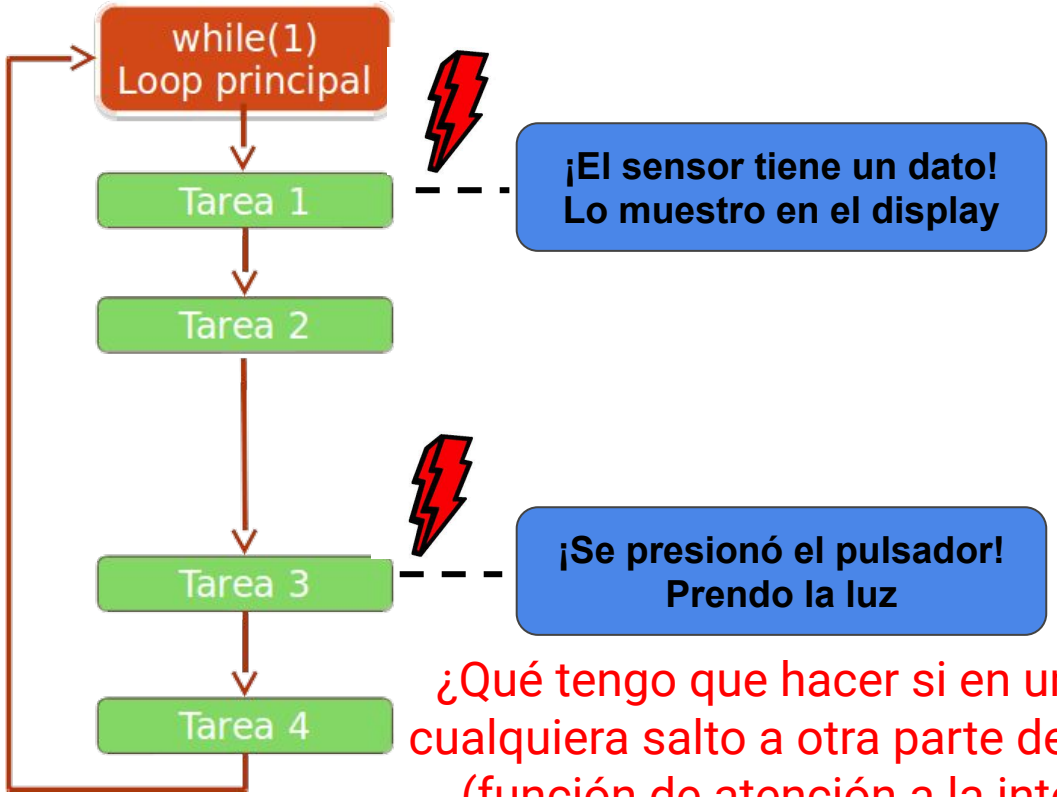
En este esquema el microcontrolador sabe exactamente en qué momento va a ver la entrada (el sensor, pulsador, etc.).

Esto hace que la atención a esa entrada sea ***SINCRÓNICA*** con el programa (el programa le destina un determinado momento para atender a ese evento).

Esto en general no es un problema (porque el microcontrolador va tan rápido que la atención a las diferentes tareas parecerían simultáneas), pero bajo ciertas circunstancias es útil que algunos eventos se atiendan EN EL MOMENTO que suceden, y no cuando se terminen de analizar el resto de las tareas.

Algunos ejemplos de atenciones inmediatas podrían ser: El vencimiento de un timer, una parada de emergencia, una medición de ancho de pulso, etc.

Interrupciones - Atención ASINCRÓNICA



En este esquema la entrada **INTERRUMPE** la ejecución del programa para indicar que hay algún dispositivo que necesita atención.

Esto hace que la atención a esa entrada sea ***ASINCRÓNICA*** con el programa (no sabemos en qué momento puede llegar una INTERRUPCIÓN).

Cuando se utilizan interrupciones, es importante considerar que una interrupción puede llegar EN CUALQUIER PUNTO de mi programa, y tomar los resguardos necesarios.

¿Qué tengo que hacer si en un momento cualquiera salto a otra parte del programa? (función de atención a la interrupción)

¿Qué tengo que hacer antes de atender la interrupción? (I)

```
switch ( estado ){

        /*...*/
        case MAQUINA_ON:

        if ( Pulsador() == TRUE ){

                if ( luz )
                SetPIN( LED1 , OFF );
                else
                SetPIN( LED1 , ON );
        }
        break;
        /*...*/
```

Al momento que llega una interrupción, debería:

- Terminar lo que estaba haciendo.
- Guardar el “estado” del microcontrolador al momento de la interrupción (como estaban sus registros más importantes).
- Guardar el lugar a donde tengo que volver luego de atender a la interrupción (punto del programa)
- **Atender la interrupción**
- Recuperar el estado del micro, volver al punto del programa desde donde partí y continuar la ejecución.

1. Terminar lo que estaba haciendo

```
    if (luz)
0x00000c9c <EINT3_IRQHandler+28>:  movw  r3, #516   ; 0x204
0x00000ca0 <EINT3_IRQHandler+32>:  movt  r3, #4096  ; 0x1000
0x00000ca4 <EINT3_IRQHandler+36>:  ldr   r3, [r3, #0]
0x00000ca6 <EINT3_IRQHandler+38>:  cmp   r3, #0
0x00000ca8 <EINT3_IRQHandler+40>:  beq.n 0xcc0 <EINT3_IRQHandler+64>
        SetPIN(LED1,ON);
0x00000caa <EINT3_IRQHandler+42>:  movw  r0, #49216 ; 0xc040
0x00000cae <EINT3_IRQHandler+46>:  movt  r0, #8201  ; 0x2009
0x00000cb2 <EINT3_IRQHandler+50>:  mov.w r1, #0
0x00000cb6 <EINT3_IRQHandler+54>:  mov.w r2, #1
0x00000cba <EINT3_IRQHandler+58>:  bl    0xb80 <SetPIN>
0x00000cbe <EINT3_IRQHandler+62>:  b.n   0xcd4 <EINT3_IRQHandler+84>
        SetPIN(LED1,OFF);
0x00000cc0 <EINT3_IRQHandler+64>:  movw  r0, #49216 ; 0xc040
0x00000cc4 <EINT3_IRQHandler+68>:  movt  r0, #8201  ; 0x2009
0x00000cc8 <EINT3_IRQHandler+72>:  mov.w r1, #0
0x00000ccc <EINT3_IRQHandler+76>:  mov.w r2, #0
```

Siempre que llegue una interrupción el micro va a estar haciendo ALGO (ejecutando alguna instrucción).

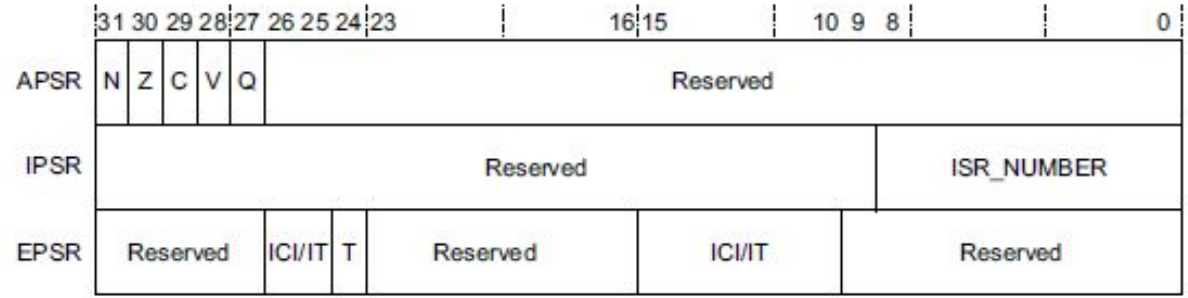
Las instrucciones de ASSEMBLER no pueden ser detenidas por la mitad, por lo que en primer lugar se debe completar la instrucción en curso.

En la imagen vemos como 3 instrucciones en C son en realidad muchas instrucciones de assembler. Cuando decimos “finalizo lo que estaba haciendo”, nos referimos a la instrucción de assembler que se estaba ejecutando en el momento en que llego la interrupción

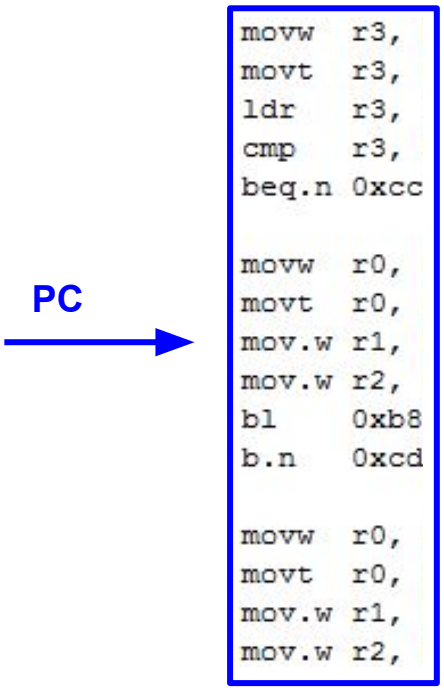
2. Guardar el estado del micro al momento de la interrupción

En todo momento el procesador está ejecutando una instrucción. Muchas veces el resultado de esta instrucción puede depender de operaciones anteriores (por ejemplo si la última operación tuvo o no carry, o si el resultado fue cero o distinto de cero), de la misma manera que las instrucciones futuras dependerán de las presentes (por ejemplo, la próxima instrucción que se vaya a buscar a memoria depende de la dirección de la instrucción que se esté ejecutando en este momento.

Todos estos parámetros se guardan en una serie de registros que se llaman “Status Registers”, y el LPC845 tiene 3: Application Status Register, Interruption Status Register y Execution Status Register. Estos 3 registros deben ser guardados al momento de invocarse una interrupción, y deben ser restaurados para que el microcontrolador “siga en el estado que estaba” cuando finalice la interrupción.



3. Guardar el lugar a donde tengo que volver luego de atender la interrupción



El Program Counter (PC) es un registro de 32 bits que guarda la dirección de la instrucción que se está ejecutando en cada momento.

Salvar el punto de retorno del programa implica, entonces, guardar el valor del registro PC

Y… ¿Cómo hago todo esto?

¡¡¡Lo hace automáticamente el procesador!!!

Como programadores, no nos debemos encargar de guardar el estado del procesador, restaurarlo, etc.

Si nos debemos encargar de configurar que interrupciones permitimos que generen este proceso, y cuáles no.

Podemos habilitar y deshabilitar las interrupciones generales o particulares de cada periférico en cualquier momento de la ejecución de un programa, o elegir atender a ciertos periféricos con estrategias de pooling o de interrupción.

¿Cómo configurar el comportamiento de las interrupciones?

El **microprocesador** tiene un periférico encargado de manejar las interrupciones, llamado NVIC (Nested Vectored Interrupt Controller).

Este periférico es el encargado de recibir los pedidos de interrupción, manejarlos (habilitarlos o deshabilitarlos), darles prioridad y hacer llegar la señal a la CPU para que se genere la interrupción.

El NVIC puede manejar hasta 32 **fuentes de interrupciones**, puede darles 4 diferentes **niveles de prioridad**, y **enmascarar** cada una individualmente.

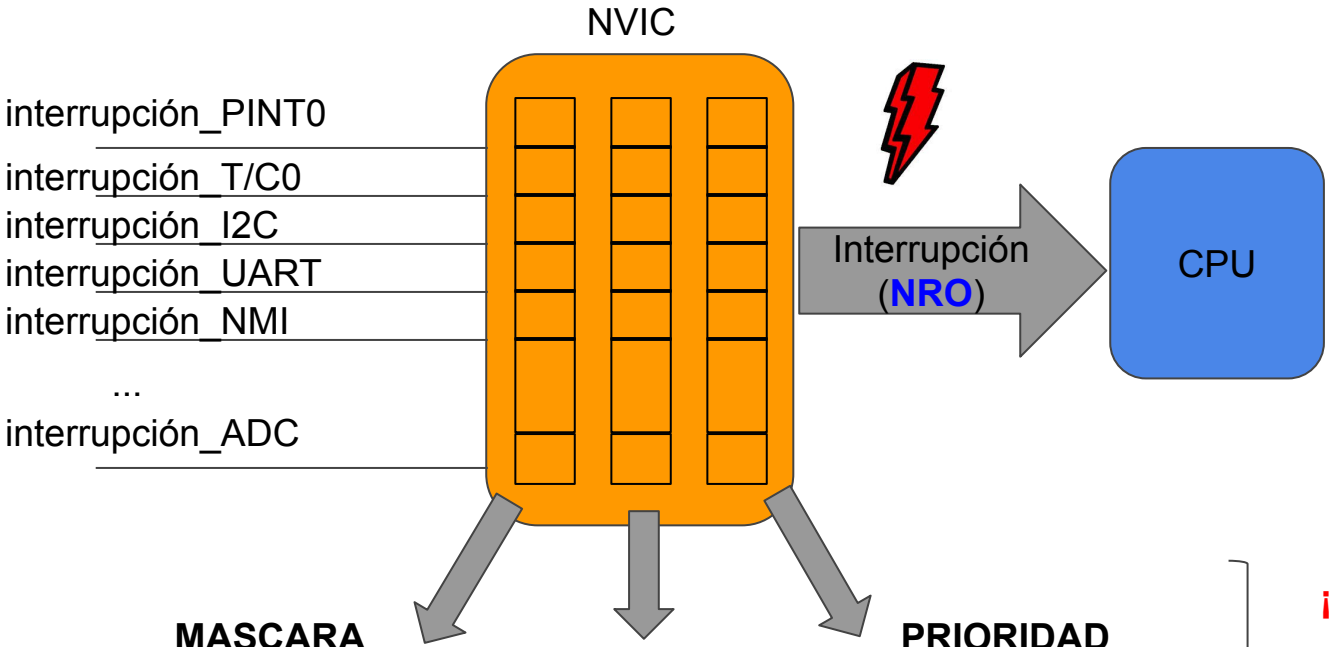
Funcionamiento del NVIC

El NVIC recibe las señales de interrupción generadas por los diferentes periféricos (o sea, en primer lugar, deberemos configurar el periférico para que genere esta señal de interrupción cuando consideremos oportuno).

Luego, cada señal puede enmascararse (habilitarse o deshabilitarse), y puede además tener un cierto nivel de prioridad (en caso de que llegue una interrupción en el momento en que se está ejecutando otra).

El NVIC resuelve las señales que recibe en función de estas dos configuraciones (habilitación y prioridad), y en caso de que se tenga que generar una interrupción al procesador le envía una señal, indicando el número correspondiente al periférico que generó la interrupción.

Comunicación NVIC - CPU



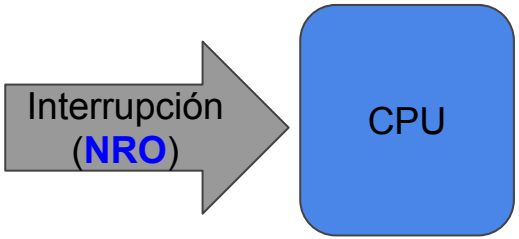
Termina la instrucción (de assembler)

Guarda **xPSR** y **PC**

Salta a la ISR **NRO**

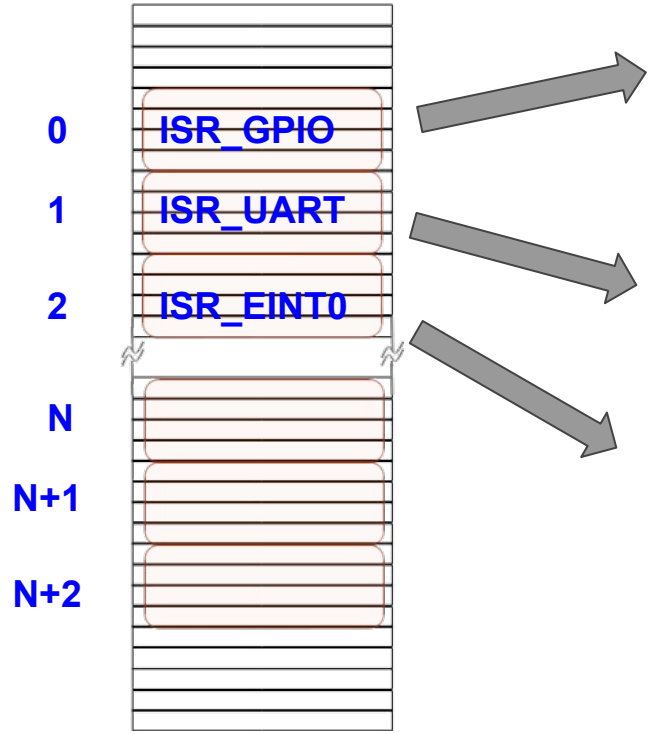
¡¡ESTOS SON REGISTROS QUE TENEMOS QUE CONFIGURARLOS NOSOTROS!!

Y… ¿Cómo sabe la CPU qué tiene que hacer?



Cada interrupción tiene entonces un número, que se condice con la posición del vector de interrupciones donde está la dirección a la función de atención a la interrupción

VECTOR DE INTERRUPCIONES




```
void ISR_GPIO ( void )
{
        /*...*/
}

void ISR_UART ( void )
{
        /*...*/
}

void ISR_EINT0 ( void )
{
        /*...*/
}
```

Fuentes de interrupción - Hoja de datos

Interrupt number	Name	Description	Flags
3	UART0_IRQ	USART0 interrupt	See Table 327 "USART Interrupt Enable read and set register (INTENSET, address 0x4006 400C (USART0), 0x4006 800C (USART1), 0x4006C00C (USART2), 0x4007 000C (USART3), 0x4007 400C (USART4)) bit description"
4	UART1_IRQ	USART1 interrupt	Same as UART0_IRQ
5	UART2_IRQ	USART2 interrupt	Same as UART0_IRQ
6	-	Reserved	-
7	I2C1_IRQ	I2C1 interrupt	See Table 357 "Interrupt Enable Clear register (INTENCLR, address 0x4005 000C (I2C0), 0x4005 400C (I2C1), 0x4003 000C (I2C2), 0x4003 400C (I2C3)) bit description".
8	I2C0_IRQ	I2C0 interrupt	See Table 357 "Interrupt Enable Clear register (INTENCLR, address 0x4005 000C (I2C0), 0x4005 400C (I2C1), 0x4003 000C (I2C2), 0x4003 400C (I2C3)) bit description".
9	SCT_IRQ	State configurable timer interrupt	EVFLAG SCT event
10	MRT_IRQ	Multi-rate timer interrupt	Global MRT interrupt. GFLAG0 GFLAG1 GFLAG2 GFLAG3

Como vemos, cada fuente de interrupción tiene un número (fijo) asignado, por lo que cuando se genere una señal de interrupción de este periférico, el NVIC podrá identificar cuál es el periférico que interrumpió a partir de su número, y sabrá a que función de atención a interrupción (ISR) deberá saltar.

Resumen: ¿Cómo configurar las interrupciones?

1. Inicializar el vector de interrupciones, poniendo la dirección de las ISR correspondientes en cada posición.
2. Configurar el NVIC (Habilitación y prioridad de las IRQs).
3. Configurar el periférico correspondiente y los pines asociados (Que cada periférico genere o no una señal ante un determinado evento).
4. Escribir la rutina de servicio de interrupción (ISR).

Inicializar la Tabla de vectores (I)

El fabricante lo resuelve en el archivo *cr_startup_lpc84x.c*

```
//*****************************************************************************
//
// The vector table.
// This relies on the linker script to place at correct location in memory.
//
//*****************************************************************************
extern void (* const g_pfnVectors[])(void);
__attribute__ ((used,section(".isr_vector")))
void (* const g_pfnVectors[])(void) = {
    // Core Level - CM0plus
    &_vStackTop, // The initial stack pointer
    ResetISR,                               // The reset handler
    NMI_Handler,                            // The NMI handler
    HardFault_Handler,                      // The hard fault handler
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    __valid_user_code_checksum,             // LPC MCU Checksum
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    SVC_Handler,                            // SVCall handler
    0,                                      // Reserved
    0,                                      // Reserved
    PendSV_Handler,                         // The PendSV handler
    SysTick_Handler,                        // The SysTick handler
```

Vector de punteros a función

Nombre de la funciones

Abramos el archivo para ver el vector completo.

Inicializar la Tabla de vectores (II)

```
extern void (* const g_pfnVectors[])(void);
__attribute__ ((used,section(".isr_vector")))
void (* const g_pfnVectors[])(void) = {
    // Core Level - CM0plus
    &_vStackTop, // The initial stack pointer
    ResetISR,                               // The reset handler
    NMI_Handler,                            // The NMI handler
    HardFault_Handler,                      // The hard fault handler
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    __valid_user_code_checksum,             // LPC MCU Checksum
    0,                                      // Reserved
    0,                                      // Reserved
    0,                                      // Reserved
    SVC_Handler,                            // SVCall handler
    0,                                      // Reserved
    0,                                      // Reserved
    PendSV_Handler,                         // The PendSV handler
    SysTick_Handler,                        // The SysTick handler
    I2C2_IRQHandler,                        // I2C2 controller
    I2C3_IRQHandler,                        // I2C3 controller
    CTIMER0_IRQHandler,                     // Timer0
    PININT0_IRQHandler,                     // PIO INT0
    PININT1_IRQHandler,                     // PIO INT1
```

El fabricante ya les pone nombre a las funciones. NO PUEDO ELEGIR QUE NOMBRE PONERLE

Prototipos de las funciones de interrupción

Para poder asignarlas al vector de funciones de interrupción se deben definir y declarar las funciones. También está en el archivo *cr_startup_lpc84x.c*

Prototipos de funciones de interrupción para excepciones Core

```
void ResetISR(void);
WEAK void NMI_Handler(void);
WEAK void HardFault_Handler(void);
WEAK void MemManage_Handler(void);
WEAK void BusFault_Handler(void);
WEAK void UsageFault_Handler(void);
WEAK void SVCall_Handler(void);
WEAK void DebugMon_Handler(void);
WEAK void PendSV_Handler(void);
WEAK void SysTick_Handler(void);
WEAK void IntDefaultHandler(void);
```

Prototipos de funciones de interrupción para interrupciones Cortex M3

```
...
void PWM1_IRQHandler(void) ALIAS(IntDefaultHandler);
void I2C0_IRQHandler(void) ALIAS(IntDefaultHandler);
void I2C1_IRQHandler(void) ALIAS(IntDefaultHandler);
void I2C2_IRQHandler(void) ALIAS(IntDefaultHandler);
void SPI_IRQHandler(void) ALIAS(IntDefaultHandler);
void SSP0_IRQHandler(void) ALIAS(IntDefaultHandler);
void SSP1_IRQHandler(void) ALIAS(IntDefaultHandler);
void PLL0_IRQHandler(void) ALIAS(IntDefaultHandler);
void RTC_IRQHandler(void) ALIAS(IntDefaultHandler);
void EINT0_IRQHandler(void) ALIAS(IntDefaultHandler);
...
```

Prototipos de las funciones de interrupción (II)

```
void PININT0_IRQHandler(void) ALIAS(IntDefaultHandler);
```

ALIAS significa que el nombre de la función definida refiere a otra función. Se usa para hacer que muchas funciones se refieran a un misma función. Es una función default que debe estar definida si o si por seguridad. El modificador **ALIAS** está definido para que incluya también al atributo weak, por lo tanto las funciones ISR (Rutinas de servicio de interrupciones) están declaradas por default como **ALIAS** lo que implica que todas se refieren a otra común y que son válidas siempre y cuando no se defina otra con el mismo nombre que no sea **WEAK**.

```
WEAK void SysTick_Handler(void);
```

WEAK significa que esa función será pisada por otra con el mismo nombre. Si no hay otra con el mismo nombre, entonces es válida. Se usa esto para definir funciones default.

Para escribir nuestras funciones de interrupción se debe redefinir las que ya existen, debido al atributo WEAK, simplemente escribimos otra función **con el mismo nombre y ya tenemos NUESTRA función de interrupción**

funciones de interrupción default

```
__attribute__ ((section(".after_vectors")))
void SysTick_Handler(void)
{ while(1) {}
}
```

```
__attribute__ ((section(".after_vectors")))
void IntDefaultHandler(void)
{ while(1) {}
}
```

IMPORTANTE: Las funciones de interrupción por defecto realizan un while(1), si no redefino **las que voy a usar** la ejecución se va a quedar bloqueada en la función.

Configurar el NVIC (Registro ISER0)

Write — Writing 0 has no effect, writing 1 enables the interrupt.

Read — 0 indicates that the interrupt is disabled, 1 indicates that the interrupt is enabled.

Table 110. Interrupt Set Enable Register 0 register (ISER0, address 0xE000 E100) bit description

Bit	Symbol	Description	Reset value
0	ISE_SPI0	Interrupt enable.	0
1	ISE_SPI1	Interrupt enable.	0
2	ISE_DAC0	Interrupt enable.	0
3	ISE_UART0	Interrupt enable.	0
4	ISE_UART1	Interrupt enable.	0
5	ISE_UART2	Interrupt enable.	0
6	-	Reserved	0
7	ISE_I2C1	Interrupt enable.	0
8	ISE_I2C0	Interrupt enable.	0
9	ISE_SCT	Interrupt enable.	0
10	ISE_MRT	Interrupt enable.	0
11	ISE_CMP or ISE_CAPT	Interrupt enable for both comparator and Capacitive Touch.	0
12	ISE_WDT	Interrupt enable.	0
13	ISE_BOD	Interrupt enable.	0
14	ISE_FLASH	Interrupt enable.	0

Colocando un 1 en el bit correspondiente del registro ISER0 o ISER1 habilito al NVIC para controlar esa interrupción

¿Cómo llego al registro ISER0? Bloque de registros del NVIC

Table 109. Register overview: NVIC (base address 0xE000 E000)

Name	Access	Address offset	Description	Reset value
ISER0	RW	0x100	Interrupt Set Enable Register 0. This register allows enabling interrupts and reading back the interrupt enables for specific peripheral functions.	0
-	-	0x104	Reserved.	-
ICER0	RW	0x180	Interrupt Clear Enable Register 0. This register allows disabling interrupts and reading back the interrupt enables for specific peripheral functions.	0
-	-	0x184	Reserved.	0
ISPR0	RW	0x200	Interrupt Set Pending Register 0. This register allows changing the interrupt state to pending and reading back the interrupt pending state for specific peripheral functions.	0
-	-	0x204	Reserved.	0
ICPR0	RW	0x280	Interrupt Clear Pending Register 0. This register allows changing the interrupt state to not pending and reading back the interrupt pending state for specific peripheral functions.	0
-	-	0x284	Reserved.	0
-	-	0x304	Reserved.	0
IPR0	RW	0x400	Interrupt Priority Registers 0. This register allows assigning a priority to each interrupt. This register contains the 2-bit priority fields for interrupts 0 to 3.	0
IPR1	RW	0x404	Interrupt Priority Registers 1 This register allows assigning a priority to each interrupt. This register contains the 2-bit priority fields for interrupts 4 to 7.	0

//Habilitar la interrupción de la UART0:

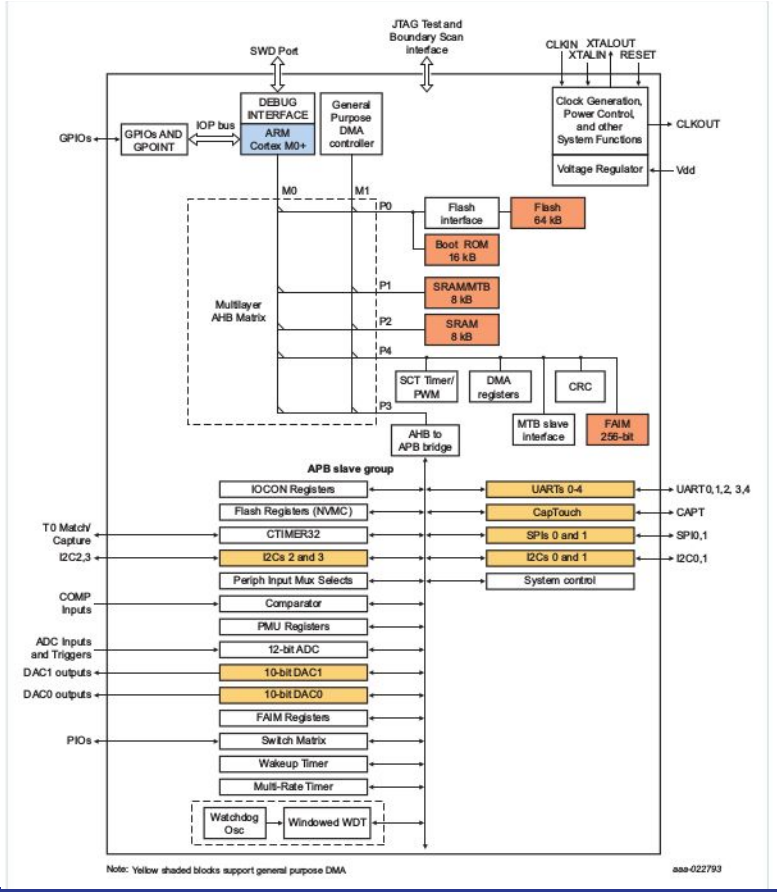
```
NVIC->ISER0 |= 1<<3;
```

De la misma manera que el registro ISER sirve para habilitar una interrupción (Set), el registro ICER sirve para deshabilitar una interrupción (Clear)

Registros de prioridad
(no los usamos por ahora)

Configurar el periférico y los pines asociados

- Cada periférico posee sus fuentes de interrupción.
- Se deben habilitar en los registros del periférico que deseo utilizar las interrupciones que queramos utilizar.
- Se debe implementar la función de interrupción asociada a la fuente elegida.



Escribir la rutina de servicio de interrupción (IRQHandler o Handler)

Para la fuente de interrupción que deseamos utilizar buscamos en el archivo startup el nombre que debemos utilizar y redefinimos la función.

IMPORTANTE

- Nuestro código de una función de interrupción debe ser CORTO y SIN LOOPs BLOQUEANTES.
- Las funciones de interrupción son ASINCRONICAS y las invoca el CPU, NO podemos recibir NI devolver nada.
- Para comunicarnos con nuestro programa principal debe utilizar variables globales

```
void SysTick_Handler(void)
{

    //Nuestro código

}
```

Comunicación entre interrupción y programa principal

```
int main(void)
{
    Kit_Init();
    HW_Init();

    while(1)
    {
        if(g_flag)
        {
            if(GetPIN(0, 22))
                SetPIN(0, 22, 0);
            else
                SetPIN(0, 22, 1);

            g_flag = 0;
        }
    }

    return 0 ;
}
```

Se produce la interrupción. No sabemos cuando!

Para avisarle al programa principal cambiamos un flag GLOBAL.

```
uint8_t g_flag = 0;

void PININT0_IRQHandler(void)
{
    //Limpio el bit de status:
    PINT->IST |= 1;

    g_flag = 1;
}
```

Comunicación entre interrupción y programa principal

En caso de que la acción a realizar por la función de interrupción sea corto y conciso se puede realizar dentro de la función. NUNCA tengo que realizar tareas largas o loops bloqueantes

```
int main(void)
{
    Kit_Init();
    HW_Init();

    while(1)
    {

    }

    return 0 ;
}
```

```
void PININT0_IRQHandler(void)
{
    //Limpio el bit de status:
    PINT->IST |= 1;

    if ( GetPIN ( 0 , 4 ) );
        SetPIN( 1,0,OFF );
    else
        SetPIN( 1,0,ON );

}
```