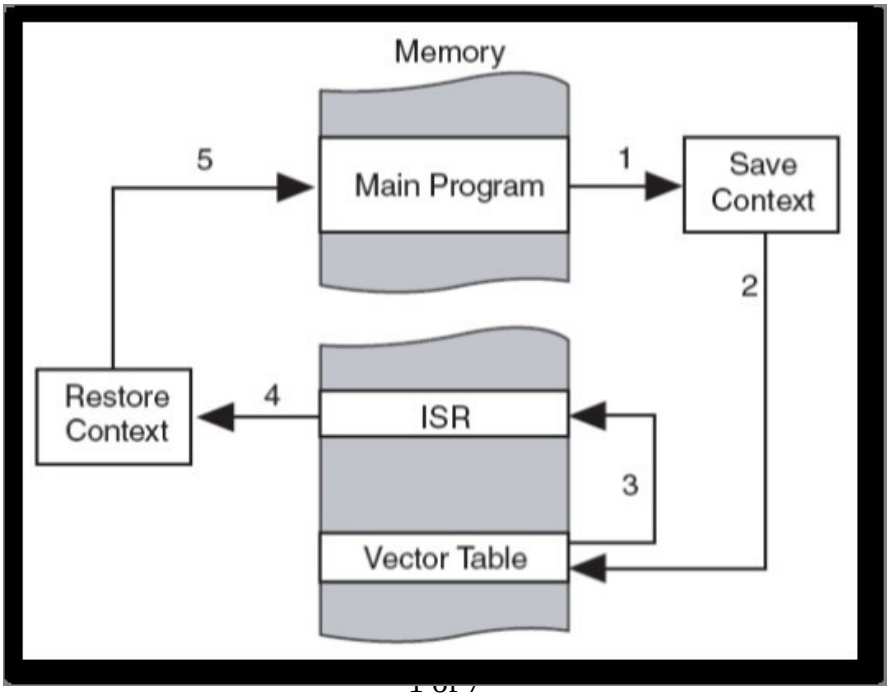
Title: Programming interrupts and using timers
Author: Craig Duffy 6/12/21
Module: Embedded Systems Programming
Awards: BSc CSI, Forensic Computing
Prerequisites: Basic computer architecture and some C

In our previous worksheet we saw how we could get input from a switch or button and send output to an LED. For such simple programs we could just loop round and continually test the state of the buttons to see what is going on and then do the required output. This method of operating is known as dedicated polling, which means going around asking the devices if they have in input or, if output devices, if they are ready to accept new data for output. This method of operating has the benefit of being simple to implement but is very limited and inefficient. If there are lots of devices then the polling loop could become very big and this might mean that devices won't get serviced when necessary. This can mean either delayed or lost input. For example, if a keyboard was an input device at the end of a long polling loop, then either it would appear unresponsive and then suddenly spew out data or data might get lost, leading to all sorts of problems and user frustrations. If it is crucially important that the data doesn't get lost or delayed then this system would be a real problem. An alternative method is busy-wait polling – in the keyboard example the system stalls waiting for data from the keyboard in order not to miss it. This then has the problem that other devices won't get serviced. As more and more devices were attached to computers this problem got more and more pressing. Alternatively, the devices can be polled on a timed schedule, called periodic polling.

Perhaps a better solution is to get the devices to 'tell' the processor when they are ready – either to receive or transmit data. This allows the processor to get on with other tasks and not to waste time polling, however this means that the processor must be able to support interrupts and have mechanisms for dealing sensibly with them. For example, what happens to the interrupted process? What happens when the interrupt itself gets interrupted? What happens when several interrupts happen simultaneously? The provision of interrupts requires extra hardware, interrupt lines into the processor and priority levels to decide which interrupt to respond to. Earlier systems had limited numbers of interrupts and programming them required some skill. For example, all Interrupt Services Routines (ISRs) would have to be coded in assembler and certain features and constructs were not allowed.

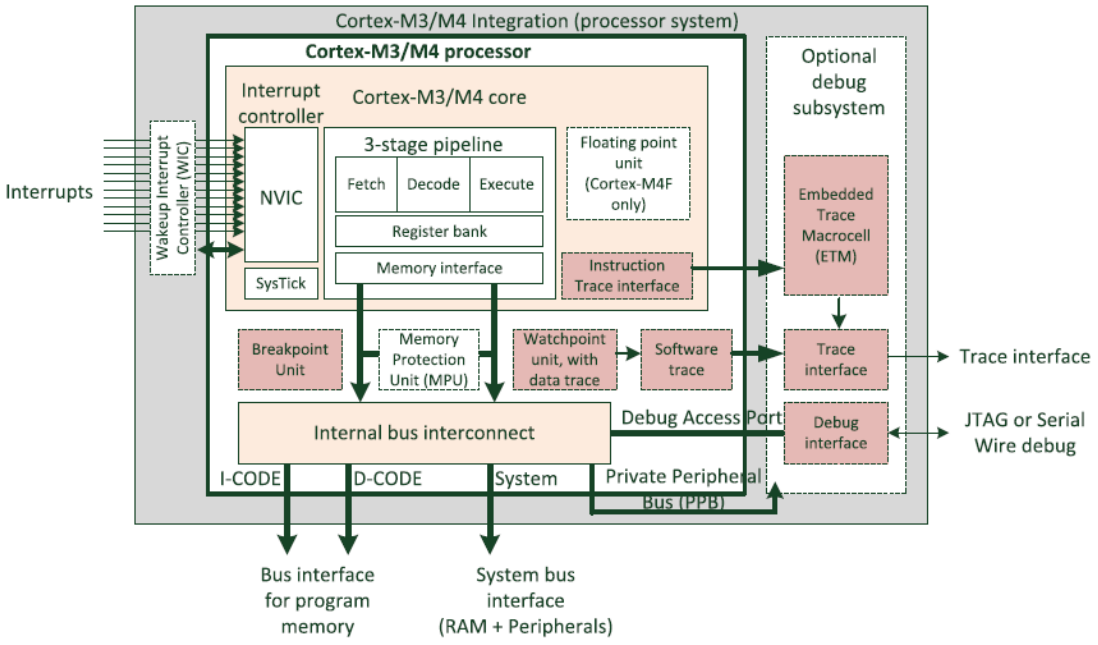


The system also has to decide what to do about the task that was interrupted. At some point the task will need to be resumed. In order to allow the resumption of interrupted tasks a certain amount of data needs to be saved, this data, known as the volatile environment or context of the task, is the minimum amount of data required to restart the it. This would normally include CPU registers such as the program counter (PC), stack pointer (SP), link register (LR) and frame counter (FP). Other registers and data might need saving too, for example if a task is manipulating a screen the X and Y co-ordinates will need saving. This data will differ from system to system and will be affected by the language the code is written in and the operating system the code is being executed upon.

The provision of interrupts allows us to run more complex software such as operating systems and to support more sophisticated hardware sub-systems such as direct memory access (DMA) or devices such as network interface or video cards. As such systems and the supporting hardware gets increasingly complex the interrupt support systems have had to evolve to support this. This means that systems have to cope with large number of interrupts, be able to allocate priorities to interrupts, block some interrupts and make sure that the correct code is matched with its interrupt.

Interrupts on the CORTEX M3/M4 Series

The Cortex M3 takes a quite different approach to interrupt handling from earlier ARM cores, now behaving much more like Intel processors. The interrupts are all handled by the **NVIC** (Nested Vectored Interrupt Controller). This handles all interrupts and allows code developed on one core to be reasonably portable to another. It also allows all of the ISRs, (Interrupt Service Routines), to be coded in C, saving the programmer from having to use assembler code, which to date was the normal method of coding ISRs.



In our previous worksheets we have already created and used ISRs as there are some that the system needs to get started. These were set up in the file **startup_stm32l475xx.c**. In the declaration **void (* const g_pfnVectors[])(void)** the list of vectors is set up and initialised. The majority of the vectors are set to the default handler (**default_handler**) which contains a

do-nothing loop. However, there are a number of entries that need to be filled with functional code.

1 Reset		-3	Reset_Handler
2 NMI		-2	NMI_Handler
3 Hard Fault	-1		HardFault_Handler
4 MemManage Fault	Prg		MemManage_Handler
5 Bus Fault	Prg		BusFault_Handler
6 Usage Fault		Prg	
UsageFault_Handler			
11 SVC		Prg	SVC_Handler
12 Debug Monitor	Prg		DebugMon_Handler
14 PendSV	Prg		PendSV_Handler
15 SYSTICK	Prg		SysTick_Handler
16 Interrupt #0		Prg	(device-specific)
17 Interrupt #1 - #239		Prg	(device-specific)

ARM Cortex interrupts

The first 15 are all fixed system interrupts - the lower the number the higher the priority. The ones without numbers (Prg), have programmable priorities and these can be set, to allow interrupting devices to have higher or lower priority dependent upon their function/importance. The vector which we have already initialised is the **Reset_Handler**, which is the code which sets up the program's execution environment and hands over to **main**.

In order to use interrupts, we need to include some header and C files in our source and Makefiles. Most of the NVIC code is in **stm32l4xx_hal_cortex.c/h** but we will also need some code from the files **stm32l4xx_hal_gpio.c/h** and **stm32l4xx_hal_exti.c/h**. This is because we will require some of the pins in the GPIO area to be used for interrupts and the code in **s stm32l4xx_hal_exti** is for external interrupts which we will use when we interface to buttons or serial ports. The set of functions provided for the NVIC are given in the table below. The main data structures for interrupts are **EXTI_InitTypeDef EXTI_InitStructure** and **NVIC_InitTypeDef NVIC_InitStructure**. These are described in the respective header files.

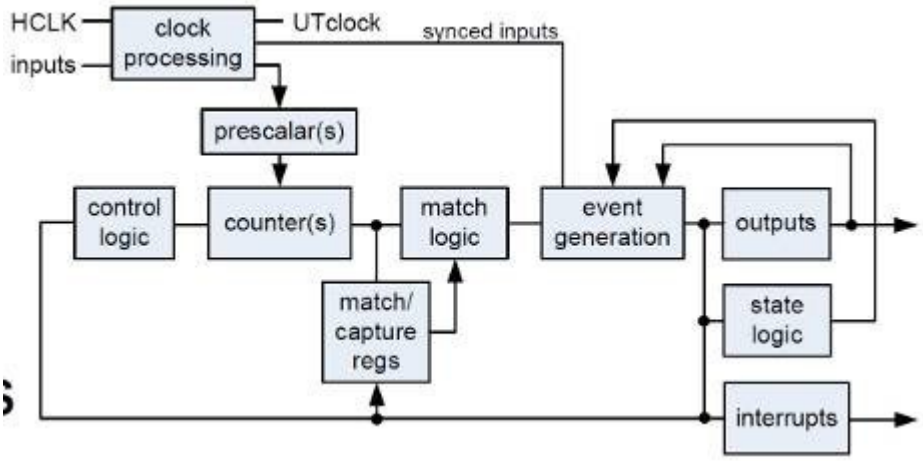
CMSIS function(1)	Description
void NVIC_EnableIRQ(IRQn_Type IRQn)	Enables an interrupt or exception.
void NVIC_DisableIRQ(IRQn_Type IRQn)	Disables an interrupt or exception.
void NVIC_SetPendingIRQ(IRQn_Type IRQn)	Sets the pending status of interrupt or exception to 1.
void NVIC_ClearPendingIRQ(IRQn_Type IRQn)	Clears the pending status of interrupt or exception to 0.
uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)	Reads the pending status of interrupt or exception. This function returns non-zero value if the pending status is set to 1.
void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)	Sets the priority of an interrupt or exception with configurable priority level to 1.
uint32_t NVIC_GetPriority(IRQn_Type IRQn)	Reads the priority of an interrupt or exception with configurable priority level. This function return the current priority level.

1. The input parameter IRQn is the IRQ number,

NVIC access functions

Timers

Before we look at some interrupt code we need to briefly look at the timers that the Cortex provides. As we have already seen timers are crucial for the operation of computer systems. The Cortex has a wide range of timers available. We are going to configure a timer to generate our interrupts and flash an LED (what else!). Timers require a clock input, they have a pre-scaler, which is used to divided the crystal clock signal, they normally have a target value that they want to achieve and a series of actions to be undertaken on reaching the target.



The Cortex timers can either get their clock signal from an external source or use the main CPU clock, which can run between *2 and 80Mhz* and this can be divided down by a 16 bit pre-scaler value (0..65535). The match logic can be count up, down or up/down. On reaching its target it can either reload and start again or stop. The timers can generate an event such as a signal or an interrupt, which is what we are going to do. In other circumstances the timers can be fed into another timer, so that the time period can be extended.

Our device – the STM32L475xx – has 16 timers, ranging from 2 advanced timers, through general purpose timers, basic tiers and low powered ones. They also provide the SYSTICK which allows an operating system to maintain an accurate time value. Our worksheet will use the general purpose timer, TIM2.

We next need to set up timer 2 and it's interrupt:

```
TIM_HandleTypeDef       TimHandle;
…...
…..

HAL_StatusTypeDef  configure_timer2_interrupt()
{
        /* Configure the TIM2 IRQ priority TickPriority=0 */
        HAL_NVIC_SetPriority(TIM2_IRQn, 0, 0U);
        /* Enable the TIM2 global Interrupt */
        HAL_NVIC_EnableIRQ(TIM2_IRQn);
        /* Enable TIM2 clock */
          __HAL_RCC_TIM2_CLK_ENABLE();
        /* Initialize TIM2 */
```

```
        TimHandle.Instance = TIM2;    /* Initialize TIMx peripheral  */
        TimHandle.Init.Period = 1000000U;
        TimHandle.Init.Prescaler = 100;    TimHandle.Init.ClockDivision = 0;
        TimHandle.Init.CounterMode = TIM_COUNTERMODE_UP;
        TimHandle.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
         if(HAL_TIM_Base_Init(&TimHandle) == HAL_OK)
                {
                /* Start the TIM time Base generation in interrupt mode */
                return HAL_TIM_Base_Start_IT(&TimHandle);
                }
                /* Return function status */
                return HAL_ERROR;
}
```

We are setting up the priorities very simply, as we don't have any other processes to worry about, but we will see later with FreeRTOS, that priorities can be used in very subtle ways to support system functionality.

Finally, we need an interrupt service routine – ISR.

```
void TIM2_IRQHandler()
{
  LED2_Toggle();
  //__io_putchar('i'); debug
  HAL_TIM_IRQHandler(&TimHandle);
}
```

The function name is fixed by the convention set up by ARM, in their header file **esp-lab-5.1/Drivers/CMSIS/Device/ST/STM32L4xx/Include/stm32l475xx.h.** These names are reflected in the names used in the vector table in the file **esp-lab-5.1/Lab5/startup_stm32l475xx.c**. There the names of all the vectors are given but they have the linker directive 'weak':

```
void TIM1_TRG_COM_TIM17_IRQHandler (void) __attribute__ ((weak, alias
("default_handler")));
void TIM1_CC_IRQHandler (void) __attribute__ ((weak, alias ("default_handler")));
void TIM2_IRQHandler (void) __attribute__ ((weak, alias ("default_handler")));
void TIM3_IRQHandler (void) __attribute__ ((weak, alias ("default_handler")));
```

This setting means that the name can be overridden in a later declaration. So, our declaration is taken in preference. You will notice that the other ISR are all set to the default handler.

As there are a number of different exercises you may need to copy both your main.c and Makefiles – so for the timer code is in Src/timer.c and the Makefile is Makefile_timer.

Exercise 1

Build the program and execute it. The program is called timer.c either change its name to main.c or change the Makefile to build it with that name.

Try changing the values so that the lights flash every second.

Button interrupts for LED

Now we are going to look at connecting an external event with an interrupt. We will try now to create an interrupt that is triggered by pressing one of the buttons. The code below does the initialisation of the button and sets up the interrupt. Notice how the GPIO event needs to be tied to the interrupt - via the EXTI_HandleTypeDef and EXTI_ConfigTypeDef – these structures are used to set up an external interrupt line - hexti.Line = EXTI_LINE_13 and pEXTiConfig.Line = EXTI_LINE_13, as well as saying how the interrupt will be triggered. The interrupt service routine, ISR, is initialised and given a priority with the NVIC calls.

```
//declaration for the GPIO pins for the wakeup Button


#define BLUE_BUTTON_PIN                             GPIO_PIN_13
#define BLUE_BUTTON_GPIO_PORT               GPIOC
#define BLUE_BUTTON_GPIO_CLK_ENABLE()     __HAL_RCC_GPIOC_CLK_ENABLE()
#define BLUE_BUTTON_GPIO_CLK_DISABLE()    __HAL_RCC_GPIOC_CLK_DISABLE()
```

```
EXTI_HandleTypeDef hexti;
EXTI_ConfigTypeDef pEXTiConfig;
```

```
……..

/* set up blue button ISR */
HAL_StatusTypeDef Blue_PB_EXT_Init()
{
  GPIO_InitTypeDef gpio_init_structure;

  /* Enable the BUTTON clock */
  BLUE_BUTTON_GPIO_CLK_ENABLE();
  __HAL_RCC_SYSCFG_CLK_ENABLE();

  /* Configure Button pin as input */
  gpio_init_structure.Pin = BLUE_BUTTON_PIN;
  gpio_init_structure.Mode = GPIO_MODE_INPUT;
  gpio_init_structure.Pull = GPIO_PULLUP;
  gpio_init_structure.Speed = GPIO_SPEED_FREQ_HIGH;
  HAL_GPIO_Init(BLUE_BUTTON_GPIO_PORT, &gpio_init_structure);

  hexti.Line = EXTI_LINE_13;

  pEXTiConfig.Line = EXTI_LINE_13;
  pEXTiConfig.Mode = EXTI_MODE_INTERRUPT;
  pEXTiConfig.Trigger = EXTI_TRIGGER_RISING;
  pEXTiConfig.GPIOSel = EXTI_GPIOC;
  /* Configure interrupts for button */
  if ( HAL_EXTI_SetConfigLine(&hexti, &pEXTiConfig) == HAL_ERROR)
    return HAL_ERROR;
  //set ISR priority
  HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0, 0U);
  //enable IRQ
  HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);

  return HAL_OK;
}
```

We also need an ISR for the button - this is pretty trivial as it is just toggling the LED:

```
//EXTI15_10_IRQHandler ISR
void  EXTI15_10_IRQHandler()

{
    LED2_Toggle();
    //clean up
    HAL_EXTI_IRQHandler(&hexti);

}
```

Obviously we will need to start-up the bus clock for both GPIOB and GPIOC, as we are using the button on GPIOC and the LED on GPIOB, also we need to set up the clock for the EXTI subsystem, using `__HAL_RCC_SYSCFG_CLK_ENABLE()`.

Exercise 2

Build the code – the program is called button.c, either rename or edit the Makefile as in the previous exercise.

Initialise the button and tie that to an ISR which alternates flashing the 2 green LEDs.