Department of ACES, SHU

Embedded Systems Design

Semester 2 academic year 2017/18

**Lab B: Using STM32CubeMX for operating peripherals and interrupts**

Lab B consists of three assignments  
- using the SysTick timer to improve responsiveness of the button;

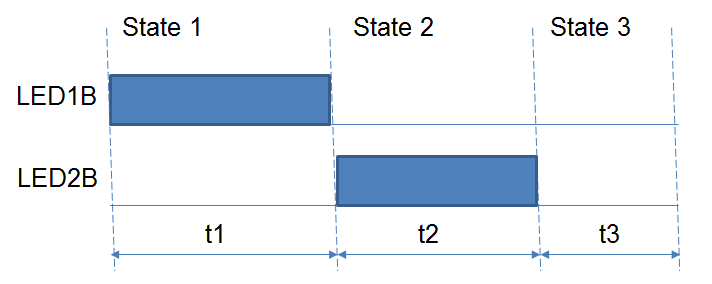
- using an additional timer to trigger the ADC at the required rate;

- using interrupts for the perfect ADC timing and ultimate button's responsiveness.

***Assignment 1*** - using the SysTick timer to improve responsiveness of the button

As you have observed in LabA, you need to press the button and wait for a while until the relevant LED goes ON. This is because there are several delays inserted into the superloop to provide for your specified timing. The delays are implemented using the ***HAL\_Delay*** delay function that is ***blocking*** - i.e., when it is executed the MCU cannot do anything else (unless the interrupts are enabled and operate). The button is checked once in the superloop thus sometimes the response can be fast but in the worst case you may need to wait for up to \_\_\_\_ s (calculate for how many from your variant for your CW report). You may want to comment out the section of the LabAa2 code that controls the LED1B and LED2B to check that the response on the button press becomes almost instant.

There is a way of improving this by using a code structure somehow similar to that of the state machine. There are three states for the LEDs (first ON, second OFF; first OFF, second ON; both OFF) that needs to be set depending on the amount of time elapsed since the start of the blinking cycle as shown on the diagram below:



(here the LEDs LED1B and LED2B, and the delays t1, t2 and t3 are determined by your student ID number in LabA1; another, more detailed, time diagram is presented in the appendix A).

In this assignment you will need to replace the piece of code that you developed to control the LED1B and LED2B with the two pieces of code - one will check the time elapsed since you completed the previous blinking cycle and set the state variable; and the other piece of code will control the LED1B and LED2B according to the present state.

***Please clone the LabA2 Cube and MDK projects into the LabB/B1 folder ones using the appropriate section of the Reference Manual on BB.***

Please declare within the appropriate placeholder for user code  
**uint32\_t state=1; // present state, start from the state 1**

**uint32\_t OldSysTick;// variable to keep track of time**

**uint32\_t** is the platform independent declaration of 32 bits unsigned integer. This data type is native to ARM microcontrollers thus it takes the least time to operate. (You may want to use **uint8\_t** or **uint16\_t** instead to save some memory but the memory is ample for this MCU thus go for simplicity and efficiency instead.)

Before entering the superloop take note of the SysTick value (it counts time starting from the CPU reset in milliseconds, 24 bits wide):

**OldSysTick = HAL\_GetTick(); state=1;**

Within the superloop instead of the old code controlling the LED1B and LED2B have the code like

**switch (state) {**

**case 1:   
 <control the LEDs for the state 1>;  
 // check whether the present System Timer**

**// value exceeds the state 1's end time and  
 // change to state 2 if true**

**if ( HAL\_GetTick() > (OldSysTick + t1) )   
 state = 2;**

**break;**

**// case for the state 2 that, when the time is right, // switches to the state 3**

**// case for the state 3 that, when the time is right, // switches to the state 1 and updates OldSysTick**

**default: <print an error message here and terminate>**

**}**

Compile and run the code and check responsiveness of the button - any delays should be unnoticeable. Please feel free NOT to calculate **t1** in the code above, just use your specific number.

However the toggling LED does not toggle properly anymore. (As it toggles at a very high rate, it looks like it is ON all the time.) Please try to fix this yourself, taking into the account that you need to toggle the LED at every start of the new blinking cycle.

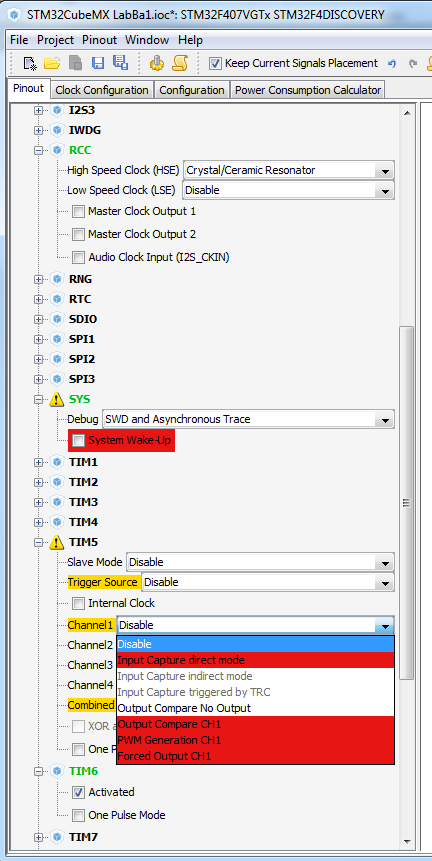
***In order to complete this assignment, you may find useful to read the appendix A.***

***Assignment 2*** - using an additional timer to trigger the ADC at the required rate

***Clone your completed project B1 into B2 before you start.***

You may have evidenced from the LabB1 that if you have operations running at various speeds it might be cumbersome to use a single clock source for all of them. Additionally you might need to have finer resolution than the one set by the STM32CubeMX to SysTick by default (1 ms per every tick of the 24 bit timer). For this reason most MCUs include additional peripherals called timers that are discussed at a lecture. We will set a timer in STM32CubeMX and use it without and with interrupts.

Peripherals in STM32CubeMx are enabled on the left of the "Pins" tab:



If a particular peripheral was enabled and configured, it is shown in green colour. For example, in the screenshot above the RCC (Reset and Clock Control), SYS (System) and TIM6 (timer 6) peripherals are enabled.

Unfolding a peripheral enables setting its parameters using checkboxes, dropdown lists and/or direct input of numerical values.

Yellow warning triangle means that some of the options related to the peripheral are no longer available to use. For example, PORTA pin 0 was configured as GPIO\_input to be used with the pushbutton. Therefore it is no longer possible to use this pin as System Wake Up, which is indicated by showing this option on a red background. The same applies to some of the options for Timer 5 because its Channel 1 is now disabled by this setting of the above mentioned pin.

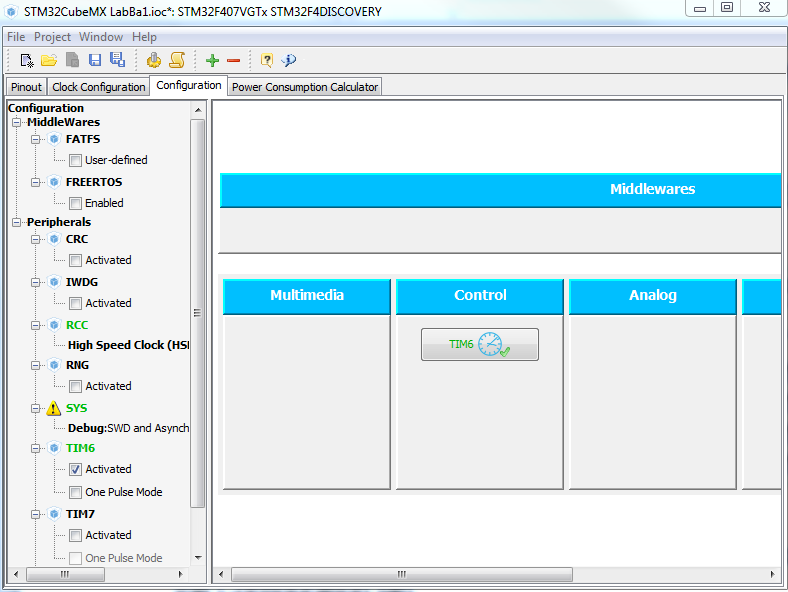
Sometimes you might find that you cannot use a particular peripheral because the pins it has to use are already taken in your design for another purpose(s). STM32CubeMX pinning/unpinning facility may help you to automatically re-allocate pins in order to allow this particular peripheral to function (see STM32CubeMX help, Appendix A, A.1 Block consistency for an example).

Use two digits of your student ID number (bXXABCDE) to determine a line in the table for the timer you will use from the formula (C+E)%6. For example, student with the ID number b1234567 will calculate (5+7) % 6 = 0 thus use the timer 6.

|  |  |  |
| --- | --- | --- |
| (C+E) % 6 | Timer to use | Advanced peripheral bus |
| 0 | T6 | APB1 |
| 1 | T7 | APB1 |
| 2 | T10 | APB2 |
| 3 | T11 | APB2 |
| 4 | T13 | APB1 |
| 5 | T14 | APB1 |

Note your timer's number, corresponding APB number and its frequency for the report.

Enable the required timer in the STM32CubeMX and proceed to its configuration using the Configuration tab. Clicking on the timer button will open the timer's configuration dialog where you need to set the prescaler and counter period values, both of 16 bits.



You will need to achieve the required timer's overflow period by finding some suitable values that satisfy the following equation (please also consult Appendix B):

Required period = (prescaler value +1) \* (counter period + 1) / FAPB,Hz

Your required period is to be found as (1+B) + 0.1\*D seconds. For example, student with the ID number b1234567 will set the period as close as possible to   
(1+4) + 0.1 \* 6 = 5.6 seconds.

Note the required period and your working for determining the ARR and PSC settings for the report.

Set the required values only and leave all the other intact. Generate the project and open it in the Keil MDK-ARM. Do not forget to enable **printf** (set the correct core frequency and enable Trace in the Debug/Settings/Trace tab of the "Target options.." window as discussed in the Reference Manual).

Find the following programming constructs added by the STM32CubeMX:

- one private variable ***htimX*** (***X*** is the number of your timer, this will be used to access the timer's data);

- one private function prototype like ***static void MX\_TIMx\_Init(void);*** that is called by the Cube generated code in the section /\*Initialize all configured peripherals\*/ (this function initialises the timer).

In the section

**/\* USER CODE BEGIN 2 \*/**

add

**HAL\_TIM\_Base\_Start\_IT(&htimX);**

in order to actually start the timer.

Delete the code that you have developed to toggle your particular LED( as you will use this LED for monitoring the ADC operation).

Instead check the overflow flag of the timer (the least significant bit of the relevant timer control register) before the state machine's code

**(htimX.Instance->SR & 0x1)**

If only this flag is set,   
 print the present counter value **htim6.Instance->CNT**,   
 clear the timer overflow flag ( **htim6.Instance->SR &= ~0x1; )** to make sure all the other bits in the register stay intact),   
 and toggle the LED.

(If you want to find what other data are available through the htim6 pointer, you may want to look into the definition of the TIM\_TypeDef structure in the file stm32f407xx.h and section 20.4.9 TIM6&TIM7 register map in the STM32F4 family reference manual).

Compile and run the project, count how many times the LED will toggle in 3 minutes and note this value for the report.

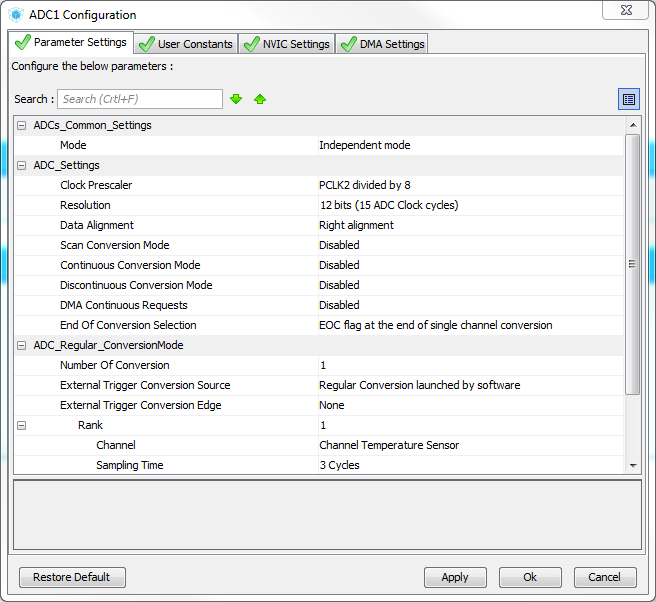
Enable the temperature channel for the ADC1 (tab Pins in the STM32CubeMX) then configure this ADC (tab Configuration as it was discussed at the lecture).

Find your configuration parameters from the following table using your student ID number (bXXABCDE) from the formula (C+D+E)%10 (i.e. using the last digit of the sum C+D+E). For example, the student b1234567 will use 5+6+7 = 18 thus line 8 from the table below.

|  |  |  |
| --- | --- | --- |
| (C+D+E) % 10 | Sampling time | ADCCLK prescaler |
| 0 | 84 | 2 |
| 1 | 144 | 4 |
| 2 | 480 | 6 |
| 3 | 84 | 8 |
| 4 | 144 | 2 |
| 5 | 480 | 4 |
| 6 | 84 | 6 |
| 7 | 144 | 8 |
| 8 | 480 | 2 |
| 9 | 84 | 4 |

Please note that for some PCLK2 values /2 prescaler might not be available (greyed out in the STM32CubeMX). If this happened, use /4 prescaler instead.

Your ADC configuration window should look similar to this (make sure you have set the sampling time and ADC prescaler to your required values:



In your already developed and tested code add the following actions completed using the HAL functions discussed at the lecture at every timer overflow (i.e. in the same piece of code where you toggle the ADC's LED):  
- to start the ADC  
- to wait for the end of conversion  
- to read out and print the ADC value.

Calculate the single conversion time and present your working in the report.

***Assignment 3*** - using interrupts for the perfect ADC timing and ultimate button's responsiveness

***Clone your completed project B2 into B3 before you start.***

When you observe operation of the design completed in assignment 2, you will find that the LED seems toggling fine but some students will observe that the counter readings are not zeros and even can vary from time to time. Therefore the timing of toggling and taking ADC samples is not perfect.

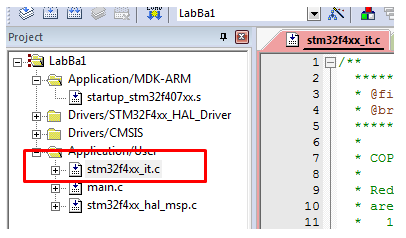
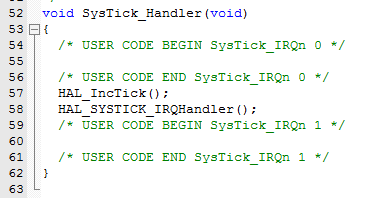
Additionally, operation of the button can be slowed down when, e.g., the ADC readings are printed out what may not be noticeable by human senses but can be recorded using a digital oscilloscope.

To fix these two imperfections we are going to use interrupts - a mechanism applicable to any MCU that equates to calling a subroutine by hardware.

At this moment please review the lecture handouts, "Interrupts" lecture from semester I and lecture 5 from this semester because you cannot complete the exercise without understanding of what you are doing according to your variant. I will only guide you through the STM32Cube MX specifics below.

***3.1. Familiarisation with the stm32f4xx\_it.c file - do not skip***

STM32CubeMX places all the ISR related content into the file **stm32f4xx\_it.c** found in the Application/User project tree as it is shown below:

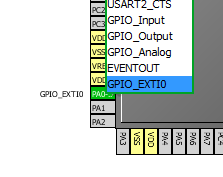
 

At this very moment you should have only one ISR in this file that processes interrupts from **SysTick** timer which increments the **SysTick** variable ( **HAL\_IncTick()**; you have accessed this variable by calling **HAL\_SysTick()**  function already), clears the interrupt flag and does other essential things **HAL\_SYSTICK\_IRQHandler()** , and gives two placeholders for the user's code as shown above on the right.

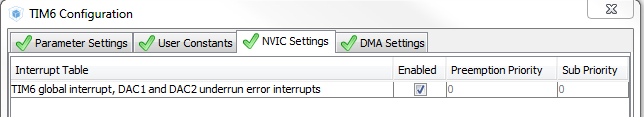
One places additional code depending on whether it is to be executed before or after the interrupt flag is cleared and essential actions took place. The choice might be application specific but for the coursework just keep your code consistently placed after the IRQHandler - there will be less chance to get the MCU accidentally hang.

***3.2. Enabling the required interrupts in the STM32CubeMX***

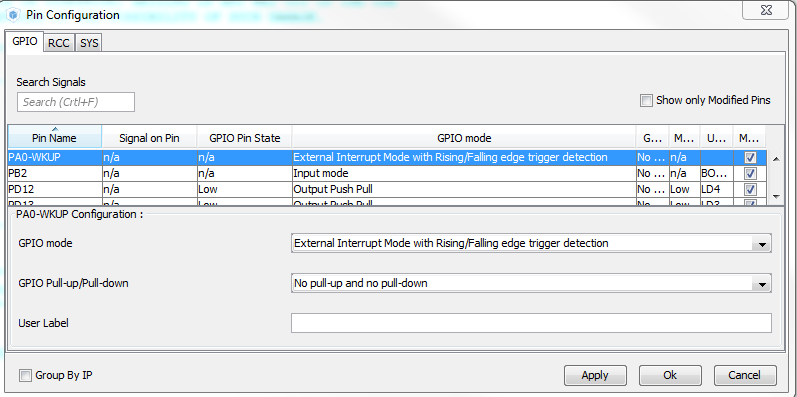
Start from reconfiguring the PA0 (the pushbutton pin) to be used for external interrupt by left clicking on it in the ***Pinout*** tab and selecting the GPIO\_EXTI0 option as shown below



Then select the "Configuration" tab and click on the icon for your timer. Select the NVIC tab and enable the interrupt by checking the box as shown



Close the configuration window for the timer, and click on the GPIO icon to configure the button pin for the interrupt use. In the GPIO configuration window one can find many more options comparing to these available on the pinout diagram. Please select only line for PA0 and, if necessary, amend the parameters as shown below:

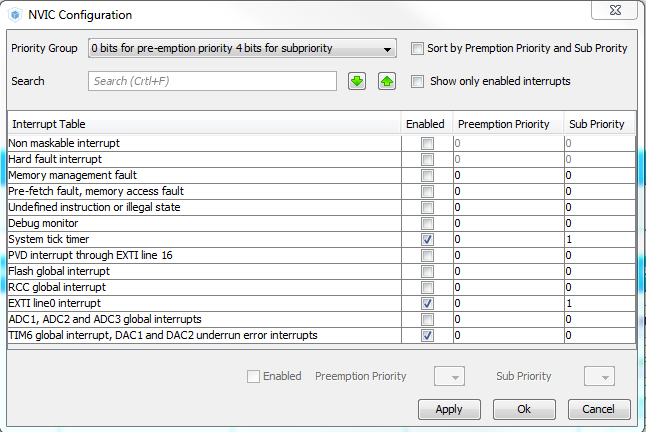


Self-check point:

- do you know what the rising and falling edges are?  
- why do they are to be used together for the button?

- what are pull-up and pull-down?  
- when does an electronic designer need to enable one of these?  
- why are any of pulls not required for the STM32F4Discovery board? (hint: schematic)

Close the configuration window for the GPIO, and click on NVIC icon to configure it as shown below



Note settings of the priorities - we want the ADC timing to be perfect thus the timer interrupt should be capable of interrupting the other interrupts.

Self-check: which letter in the abbreviation NVIC describes this capability?

Generate the project and open it in Keil MDK ARM.

***3.3. Adding the user code to the ISRs***

You will now find two additional ISRs generated by the STM32CubeMX in the file

**stm32f4xx\_it.c**.

**EXTI0\_IRQHandler** is the ISR for the button; please test the state of the button in it and control the relevant LED appropriately (simply copy your relevant code into the appropriate placeholder).

Please note: this subroutine (EXTI0 - external interrupt on line 0) will be called for every pin 0 irrespectively of its port when the interrupts on this pin are enabled. If you do need to enable interrupts from two (or more) pins on the same line, read these pins in your ISR first to determine which one caused the interrupt, and then proceed with the required functionality.

**TIMx\_IRQHandler**  is the ISR for your timer.

You will need to move here the following actions from the superloop:

- toggle the LED

- start the ADC.

In the superloop you no longer need to check whether the timer overflew; and clear the overflow flag as it is now handled by the ISR. Also remove printing of the CNT value as this is no longer makes sense.

Just check that the ADC conversion is finished, and print the value if yes.

There is a little complication with the second action as the handle for the ADC (generated by the STM32CubeMX and placed in the **main.c** ) is not visible from this file. This can be fixed by declaring variable hadc1 as external in the ISR (copy the hadc1 declaration from the main.c into the ISR and add extern in front of it). If you never came across external variables before, here are two references describing their practical use:

<http://www.geeksforgeeks.org/understanding-extern-keyword-in-c/>

<http://stackoverflow.com/questions/1433204/how-do-i-use-extern-to-share-variables-between-source-files-in-c>

Why using these external variables at all? To compile the amended source files only in a big project.

**Appendix A**

Some additional considerations that will help for completion of the assignment 1

1. Some students tried to type statements like

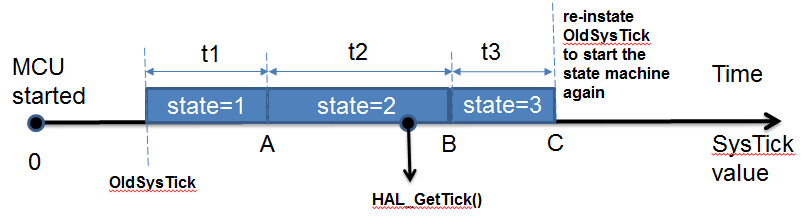
**case 1:**

**<control the LEDs for the state 1>;**

**break;**

into their code directly and have got a lot of compiler errors. This and some other lines were written using ***pseudocode*** that one needed to replace with some valid C code.

2. After discussions with various students I found that the additional graph below was understood better by many students:



The axis shows the time direction; over it there are states and below there are values of the SysTick timer.

When the MCU starts, the SysTick timer value is 0 and it is incremented every millisecond until the timer overflows (after around 4000 seconds (1ms\*224) - more than one hour - we can ignore overflow for our purposes). Before entering the superloop we sample the SysTick value into OldSysTick by calling the HAL function (HAL\_GetTick).

In the superloop, in order to establish the correct state, we sample SysTick as shown with the down arrow in the graph. If the outcome at state 1 is smaller than A (A=OldSysTick+t1, where t1 is the required time from your variant when LED1B is on and LED2B is off), **state** should be kept to be 1; otherwise it needs to be changed to 2.

If the outcome at state 2 is greater than B (B=OldSysTick+t1+t2), change **state** to 3.

If the outcome at state 2 is greater than B (B=OldSysTick+t1+t2+t3), change **state** to 3 and reload the OldSysTick.

**Appendix B**

Some additional considerations that will help for completion of the assignment 2

Some students were uncomfortable with calculations for setting PSC (prescaler) and ARR (autoreload) registers of their timer because there was a need to derive some values from the equations provided.

In order to calculate the overall division factor for the clock frequency of your timer you first need to find YOUR timer's master clock in the STMCubeMX project FAPB,Hz. Please consult relevant lecture handout for that.

Then calculate the required timer period out of your student ID number Tr,s (in seconds).

You will be getting FAPB,Hz pulses at the inpt of the timer every second thus you need to divide the input frequency by the factor of N = FAPB,Hz \* Tr,s (quite a substantial number for most students). This N is to be ideally equal to the product (PSC+1) \* (ARR+1) and there are very many ways to get close to the required N. If you set ARR reasonably high (say, to 59,999) then you can calculate  
PSC = [ N / (ARR+1) ] - 1   
where [..] takes the integer part of the number.

To see that your error is not substantial, work out the actual period you will get with your chosen PSC and ARR

Ta,s = (PSC +1) \* (ARR + 1) / FAPB,Hz