Politecnico di Torino

OSES Final Assignment: Tunable lamp

Professor:

Massimo Violante

Students:

Stefania Gabutto, 265481

Mohammadreza Beygifard, 257645

A.Y. 2019/2020

1) Introduction

The following report is an explanation of using the S32K NXP Evaluation Board for developing the final assignment of the Operating system for Embedded System course, supervised by Professor Massimo Violante. Although the code is commented and written clearly, we write this report to make a clear understanding of the project. As a starting point, we have used the example project Micrium_S32K144EVB-Q100_Blinky provided by Micrium®, based on a flat architecture operating system for the mentioned Board, which is called µC/OS-III.
The purpose of this project is to manage the RGB LED available on the board for implementing a lamp and to use the user buttons SW2 and SW3 to change its color according to specific color sequences. Moreover the light intensity of the LED has to vary according to the angular position, set by the user, of the S32K144 potentiometer.
The project files that we created for the sake of our project are:

1. *bsp_switch.h* and *bsp_switch.c*: they are necessary for initializing the pins connected to the switches that are responsible for changing colors and handling the related interrupt.
2. bsp_adc.h and bsp_adc.c: we create them to manage initialization, triggering, and the interrupt of the Analog to Digital Converter (ADC).
3. *bsp_flextimerled.h* and *bsp_flextimerled.c*: we use them to manage the FlexTimer Module and to switch on the channels of RGB LED with the signals of the PWM.
4. main.c: which is the main function of our application, and we create our task there.

2) Main file

First of all, the header files of the used libraries are included to the *main.c* file. Many global variables are declared as extern for the functioning of the application:
- *LED_COLOR* is a variable of type *LEDCOLOR_var* and it is used to store the current color of the RGB LED;
- *ADC0_conv* is a 16 bits variable used for the result of the ADC conversion (given by the interrupt handler);
- *ADC0sem* is used for the ADC semaphore.

Everything is developed inside the *StartupTask*. Here, after initializing the PWM, the ADC and the switches (see the relative part of the report for more details), the infinite main loop begins. Its first step is the start of the conversion on channel 12 of ADC0. For our purposes, the potentiometer available on the S32K144 board is mapped on the GPIO PTC14, which corresponds to ADC0 channel 12. Calling the function *BSP_ADC0_convertAdcChan_interrupt()* means moving the task to the pending list of the ADC semaphore *ADC0sem*. It will remain there waiting for the result of the conversion to be ready. When this result is available, the interrupt handler function *ADC0_IRQHandler()* posts the ADC semaphore so that the *StartupTask* is moved from the pending list to the list of ready tasks and can be executed. In this way the result of the conversion is stored in the variable *comparator* which is then passed to the function *BSP_FTM0_ChangeDutyCycle()*. This allows to update the duty cycle of the FTM channels to the value corresponding to the new position of the potentiometer.

3) Switches

In *bsp_switch.c* we can find the part of code which is responsible for correctly initializing the switches 2 and 3. First, PTC12 and PTC13 pins are connected to switch buttons SW2 and SW3 respectively, setting them as input pins. Interrupt is also enabled on port C in order to detect when a button is pressed. In particular, interrupt is configured only for rising edges on the two pins, so that it is related to the event of pushing the buttons and not to their status. The interrupt handler is *SW_IRQHandler().* Its purpose is to check from which pin the interrupt has been received (i.e. which button has been pressed) and, on the basis of this information, it performs a different call to function. If we press SW2, the function *BSP_FTM0_ToggleLEDColor2()* is called in order to perform the sequence of colors RED → GREEN → BLUE → RED; otherwise, if we press SW3, the function *BSP_FTM0_ToggleLEDColor3()* is called to obtain the sequence RED → BLUE → GREEN → RED.

4) Analog to Digital Converter

When we want to provide an analog input to our Microcontroller (MC), we shall convert it to a digital signal so the MC can work with it. Changing the light intensity of the LED by using the potentiometer is an example of Analog input for which we shall employ ADC. In our board, the potentiometer is connected to PTC14, channel 12, and we shall use ADC0-SE12. We implement a software approach for ADC, and we use a semaphore to prevent the task that needs input from ADC to waste CPU time.

In the file *bsp_adc.c* we have implemented *BSP_ADC0_Init()* to initiate ADC0. When initialization happens, it disables the interrupt for ADC, so the peripheral does not disturb the CPU unless the user changes the position of the selector.

When the user changes the angular position of the potentiometer, the ADC starts the conversion by calling the function *BSP_ADC0_TriggerChanConv()* which has as argument a 16-bit integer corresponding to the ADC channel. The code enables interrupt of the ADC, then *ADC0sem*, our semaphore variable for the ADC, gets pended and let the CPU be free from the tasks that need the data of the potentiometer, and let the ADC task to run.

When the ADC has finished its conversion and has produced the 12-bit result, *ADC0_IRQHandler()* is activated to store the result in *ADC0_conv*, change the duty-cycle of the PWM signals and post the semaphore *ADC0sem* in order to remove the task waiting for the conversion from the pending list.

As we want a proportional relation between the output voltage of the potentiometer and the light intensity of LED, we should write a map between the ADC output voltage and the duty cycle of the PWM. The duty cycle should be 0% at 0 volt and 100% at 5 volt, which corresponds to an output of the converter equal to 0x000 and 0xFFF, respectively. Moreover we should have the comparator set to 0 for 0xFFF and equal to 8000 for 0x000, so the final equation is:

$$Updated\ duty\text{-}cycle = 8000 - ADC\_conv \cdot \frac{8000}{0xFFF}$$

5) Flex Timer Module (FTM)

We select an hardware approach to generate the PWM signals for changing the light intensity of the RGB LED. Looking at the schematic of S32K144EVB, we can notice that the red channel connects to PTD15, the green channel to PTD 16, and the blue channel to PTD0. Consequently, there is a connection between the light intensity of each color and the voltage of each pin. Unfortunately, we can not regulate the voltage of the pins since in digital, when the pin is on, its voltage is always about 5 V. So, we use a trick, and we employ a PWM signal with tunable duty-cycle. In reality, by employing this trick, LED keeps blinking, but our eyes are not able to see that fast blinks and instead use a kind of integration on the whole light, which yields light intensity. For employing the mentioned trick, we connect each pin of RGB LED to a channel of FTM, and we generate three different PWM signals to manage the light intensity of the three colors. The connections are as follows:

1. FTM0_CH0 -> PTD 15 (Red)
2. FTM0_CH1 -> PTD 16 (Green)
3. FTM0_CH2 -> PTD 0 (Blue)

At the laboratory practice that we had, we always used the pins as General-purpose input-output (GPIO) which is ALT1 muxing, according to the datasheet of the NXP board. Now we should use ALT2 muxing and we initialize it using *BSP_FTM0_PWM_Init()* function in *bsp_flextimerled.c*. Also, we have modified the Channel status and Control register of the three channels to enable high-true edge alignment for PWM signals, and then FTM parameters are set. To get PWM signals with frequency 10kHz, we should calculate modulo as:

$$Modulo = \frac{f_{clock}}{Divider \cdot f_{PWM}} - 1$$

Having system clock frequency as 80 MHz and taking the default pre-scaler as 1, we set 7999 as modulo for the FTM0. We have set an arbitrary value for *comparator* for debugging purposes. The arbitrary value that we have set

is 4000, so at first, we produce a PWM signal with a 50% duty-cycle. It is replaced soon by the duty cycle that the user sets by ADC in the first iteration, but we need this initial value to check if PWM is initialized correctly. Finally, we always start with red color at start-up since this we only enable channel FTM0_CH0 by the initialization function.

The other function that we use is *BSP_FTM0_ToggleLEDColor()*, that we use one time for switch 2 and the other time with switch 3 to have the corresponding color sequences. We have defined a global variable as *LEDCOLOR_VAR* by enumeration of *bsp_color* values to use it in the mentioned function. In the end, by using a switch-case construct and taking the global variable that we have defined, we toggle LED color as the required sequence for each switch; this function performs it by changing the active channel of FTM0. The global variable that holds the LED color is defined as *volatile* to prevent the compiler from performing any unwanted optimization.

The final function that we use is *BSP_FTM0_UpdateDutyCycle().* This function is responsible for updating the duty-cycles of the three FTM0 channels, considering the given value of ADC. So, every time we change the potentiometer, we update all the duty cycles, and we change the light intensity of the corresponding LED.

6) Results

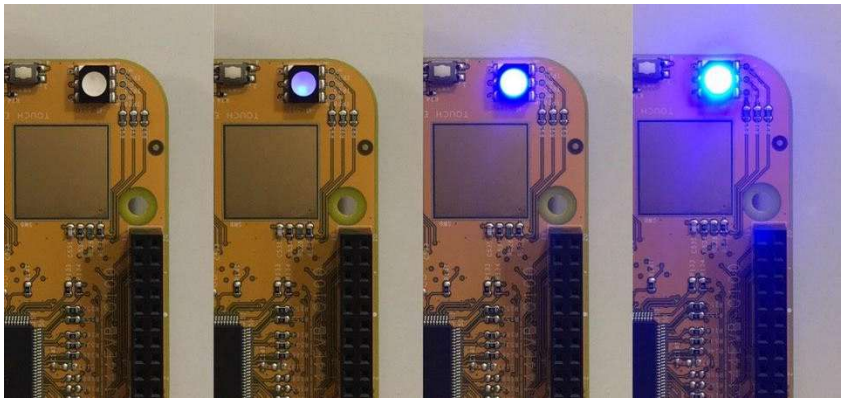

Figure 1: Variation of the LED light intensity basing on the angular position of the potentiometer

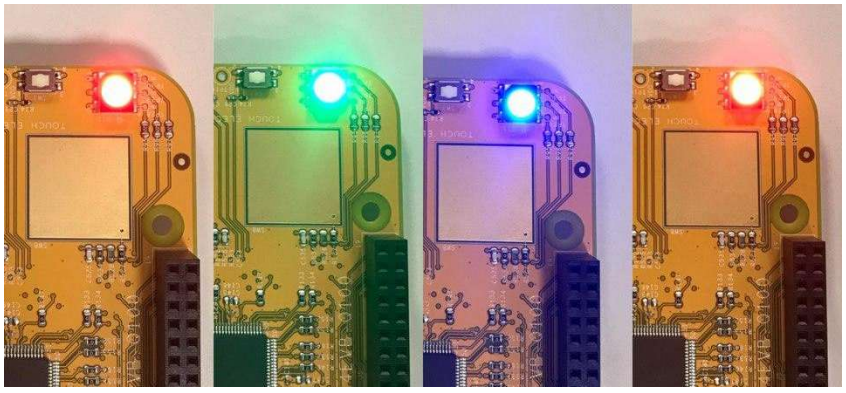

Figure 2: Sequence of colors for the LED when pushing switch 2