Operating for Embedded Systems

Final Assignment

Students	Castagneri Dario	277967
	Stochino Damiano	269745

Goal

The goal of the final assignment is the development of a tuneable lamp, for which it should be able to adjust the light intensity and the colour, using the S32K144 evaluation board and Micrium µC/OS3. In particular, the light emitted by the led must be regulated using the potentiometer available on the board, and the colour changed by pressing S2 and S3 buttons.

Peripherals overview

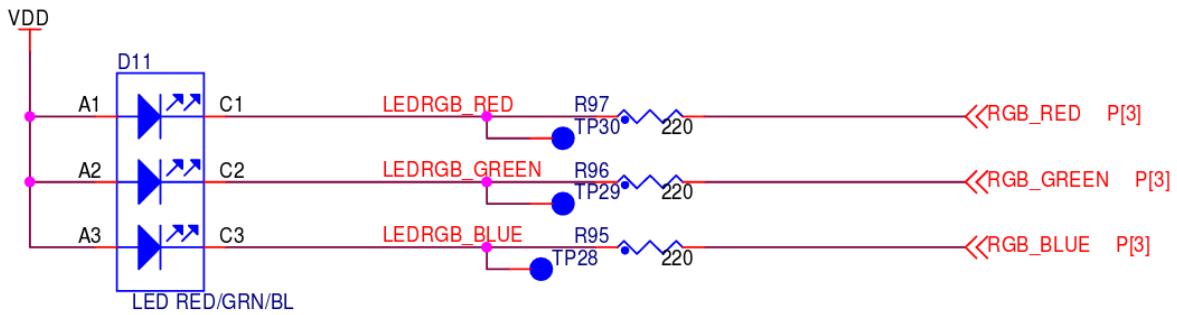
A good way of regulating the light emitted from a led through a digital circuit is to use a PWM signal. Driving a PWM signal via software is not feasible, due to the jitter caused by interrupts, scheduling latency and so on. Fortunately, we can avoid indeterministic behaviour since a hardware PWM peripheral is available on the board. In order to read the value of the potentiometer, we have to use and configure the Analog to Digital Converter (ADC) to which the potentiometer is connected. Finally, we need to configure interrupts in order to wait for both the ADC conversion to be completed and a button to be pressed, either S2 or S3.

PWM configuration

In BSP_PWM_LED_INIT() function, we set up the PWM peripheral. After activating the timer, we set the association between the RGB colours and the board pins and how to control those through FlexTimerModule0 channels, according to the following table:

FTM0_CH0	PTD15	RGB_RED
FTM0_CH1	PTD16	RGB_GREEN
FTM0_CH2	PTD0	RGB_BLUE

Correspondences between FTM0, board pins and RGB LEDs



Since the LED other terminal is connected to Vdd, it is ON when the output of the circuit is 0, and vice-versa. For this reason, the canonical definition of duty cycle is inverted: if the signal is a constant 0, it is equal to 100%, if the signal is a constant 1, it is equal to 0%. If we want to be more precise:
$d_c = t_0/(t_0+t_1)$
Assigning the mask PORT_PCR_MUX(2) to the corresponding PCR register of PORTD ([15] for RED, [16] for green and [0] for BLUE), we connect the PWM signals to the LEDs.
We decide for the edge-align PWM mode, according to which, the FTM counter cyclically counts from the FTM0->CNTIN, initially set to 0, to the FTM0->MOD values. For what concerns the MOD register, we need to do the following considerations: in order to get a 10 kHz pulse frequency, we have to set $Modulo = \frac{f_{clock}}{Divider * f_{PWM}} - 1$. Considering the 80 MHz system clock, we can use a pre-scaler (Divider) equal to 1, therefore we obtain a modulo of 7999. The previous value is written in the MOD register. The duty cycle is set acting on FTM0->CONTROLS[i].CnV, where i is replaced with 0, 1 and 2 in order to set the duty cycle for RED, GREEN and BLUE LEDs respectively. The PWM peripheral internal counter is periodically compared with the CnV value. If we set ELSA bit to 1, no matter what the ELSB bit is, the output is 0 until the value in the counter overflows the CnV threshold, then it becomes 1. Therefore, we need to set CnV to 0 for a duty cycle of 0% and to 8000 for 100%, according to the inverted definition of the duty cycle due to the LEDs wiring. During initialisation, we set a 0% duty cycle for each colour. Since CnV registers can be written directly only in the initialisation phase, we need to enable PWM synchronisation in order to access them in the future. We can choose among several updating methods. We opt to use the *software triggered updating, a*ccording to which "a software trigger event occurs when 1 is written to the SYNC[SWSYNC] bit". This means that the value in the CnV register will be effectively written in the internal register only when SWSYNC bit becomes 1. In order to update the duty cycle smoothly, the new value is not written immediately, but the update is scheduled for the next *synchronisation point*. In our up-counting case "the synchronisation points are when the FTM counter changes from MOD to CNTIN", i.e, when count terminates. Summing up, the following actions must be taken in the initialisation stage to update the CnV registers:

- Enable PWM synchronisation by writing 1 in SYNCENn bits (n = 0,1,2) of the FTM0->COMBINE register.
- In the FTM0->SYNCONF register, set the following bits to 1
  - SYNCMODE in order to enable *enhanced PWM synchronisation*
  - SWWRBUF to enable *software triggered update*
- Write 1 either in CNTMIN or CNTMAX bits of the FTM0->SYNC register in order to enable the synchronisations points.

Finally, with the last FTM->SC register configuration, we make the PWM generation starts by specifying the system clock source and enabling the three FTM0 channels (PWMENn bits).

ADC configuration
By using BSP_ADC0_init_interrupt(), we set up the Analog to Digital converter.

We decide to use ADC0 because, by looking at the board schematic, we discover that it is the only one directly connected to the potentiometer. Through the function BSP_ADC0_convertAdcChan_interrupt(), we can start AD conversion on 12 bits.
In order to reduce time latency, we choose an interrupt approach over a polling one for waiting the conversion to be completed. The ADC can be configured to rise an interrupt request when the conversion is completed. Because of that, we need to include a specific handler, ADC0_IRQHandler(), that is called every time an ADC interrupt occurs, or rather, a conversion is completed. The handler contains the instructions needed for reading the result. Furthermore, it includes a semaphore, initialised to 0, to notify whether the ADC completes the conversion and to take the next task among those waiting. The read value is re-scaled with an up-bound of 8000 (modulo+1) and saved in the global variable *ADC0_adc_chx*.

ADC reading and duty cycle update
In *main.c*, the StartupTask() initialize all the peripherals, i.e, ADC, PWM and buttons (see later). Then, an infinite loop continuously asks the ADC to make a conversion, waits on the semaphore shared with the ADC handler, and, when the conversion is completed, uses the *ADC0_adc_chx* value to update the CnV register and the duty cycle. We need to set CnV = ADC0_adc_chx.
When the PWM is running, in order to update the CnV register, we have to proceed as follows

- Write the desired new value in the CnV[i] register
- Trigger a software event by writing 1 in SWSYNC bit of the FTM0->SYNC register
- The SWSYNC bit is reset at the next synchronisation point by the peripheral, so that a new event can be detected

Colour switch
In order to keep track of the current turned on colour, a global variable current_color is used. The variable is initialised to 0 (red LED). All the duty cycle updates act on the FTM0->CONTROLS[current_color].CnV register, while other duty cycles are set to 0%. The possible current_color values {0,1,2} (red, green, blue) are cyclically assigned. We configure a unique interrupt handler for both S2 and S3, to detect whether a button is pressed.
In order to distinguish which of the 2 buttons has raised the interrupt, we must check the IFS (Interrupt Status Flag) bit in PORTC->PCR[12] (button 2) and PORTC->PCR[13] (button 3) registers.
If S2 is pressed, current_color is set as (actual_color+1)%3. If S3 is pressed, current_color is set as (current_color+2)%3. After either S2 or S3 is pressed and current_color updated, the CnV registers related to the other two colours are set to 0%. Finally, CONTROLS[current_color].CnV is set to curent_CnV, a global variable that keeps track of the current duty cycle. IFS bit must be reset manually.