**CG2271 Real-Time Operating Systems**

**Assignment 2 – Analog Systems**

1. Introduction

In this assignment, we will be going through a series of activities to teach you how to program a timer, set timer-based interrupts and use various timer-based peripherals, namely Pulse-width modulation (PWM) and Analog-to-digital converters (ADC).

The first activity will teach you how to program a simple timer, which you **do not** need to demonstrate. It is for your own learning, and you will need to modify it for your next activities.

The second activity will teach you how to use PWM to control your RGB LEDs, which will be used in conjunction with your third activity, which teaches you how to use an ADC to take in analog inputs and convert it into digital outputs to control your RGB LEDs as well.

1. Submission

Please fill your answers in the provided answer book, and rename it to AS2-B0x-yy.docx. So if you are in sub-group 8 of the Tuesday group (B01), your answer book should be named AS2-B01-08.docx.

Submit only one copy of the answer book per sub-group, either in DOCX or PDF format. **Submit to Canvas by 2359 hours on Friday 13 February 2026.** Late assignments will receive a heavy penalty.

1. Lab Assignment

You will be given 3 files with code snippets (as02\_1.c, as02\_2.c and as02\_3.c), which are templates to help you complete these activities. You need to create a new project in MCUXpresso, and copy over the entire code into the .c file in order to compile them, as we have several headers and enums that we will be using in the code.

In your groups, you are required to create **one** working project to demonstrate the usage of a timer-based interrupt, as well as the use of PWM and ADC. The lab demo is only necessary for activity 2.3 (ADC). However, you are still required to complete activities 2.1 and 2.2, as they will help you complete 2.3.

**Lab 2.1 – Timers**

In this activity we will program the timer TPM0 on the MCXC444 to flash the RGB LEDs on the FRDM-MCXC444 board in sequence. To save you some time, a few functions for controlling the LEDs will be given to you. However, you will still be required to figure out the full timer code, as well as handle the logic for deciding how and when to toggle the LEDs. As this is a self-guided exercise, we do not require you to demonstrate this part to the TAs.

The low frequency internal clock for MCG-Lite has a 2 MHz clock and an 8 MHz clock. For this activity we will use the **2 MHz clock**.

We will clock timer TPM0 using the MCG Internal Reference Clock (MCGIRCLK), but we reduce the clock speed to **1 MHz**.

Write the function setMGCIRClk(). To configure MCGIRCLK appropriately:

* Use the 2 MHz internal reference clock.
* Use LIRC DIV2 to scale it down to 1 MHz.

**Question 1. 1**. (3 marks)

Complete the following function:

void setMCGIRCLK() {

//clear CLKS

//set IRCLKEN to enable LIRC

//choose the 2MHz Clock

//set FRCDIV and LIRC\_DIV2

}

**Question 1.2**. (2 marks)

When reducing the internal clock from 2 MHz to 1 MHz, we should set a division factor of 2 for LIRC DIV2, and set the division factor for LIRC DIV1 at 1. Why can’t we do it the other way around? (I.e. set division factor of LIRC DIV1 to 2, and LIRC DIV2 to 1). **Hint:** Look at Figure 27-1 in the Reference Manual or page 13 of Lab Lecture 4.

We will now configure TPM0 to trigger an interrupt every 500ms. Choose the right prescalar and modulo (TPM0->MOD) values that gives the best accuracy. (**Note**: prescalar and prescale counter mean the same thing.)

**Question 1.3**. (2 marks)

State and explain why your chosen prescalar and modulo values will give the best accuracy.

We will now set up TPM0 timer. Set the TPM0\_IRQn interrupt to the **highest** priority level.

**Question 1.4**. (4 marks)

Complete the following function:

void initTimer() {

//disable TPM0 Interrupt

//setMCGIRClk(); //uncomment this after question 2

//Turn on clock gating for TPM0

//Clear TPM clock source and select MCGIRCLK

//Turn off TPM0 and clear Prescale counter

//Set Prescale Counter

//Initialize counter to 0

//Initialize modulo

//Set priority to highest and enable IRQ

}

Complete the code for the ISR. Fill in the correct vector name, and the test condition for the TOF. Note: Zero TPM0->CNT and clear the TOF flag.

**Question 1.5**. (2 marks)

Complete the following function and copy it into CG2271Lab2.c:

void TPM0\_IRQHandler() {

//clear pending IRQ

//Check if TOF is set

//IF TOF is set, 1. count = (count + 1) % 6

// 2. reset counter and 3. Clear TOF bit

}

Finally, complete the functions to start and stop the timer.

**Question 1.6**. (2 marks)

Cut and paste your code for startTimer() and stopTimer() below.

void startTimer() {

//set CMOD bits

}

void stopTimer() {

//mask CMOD bits

}

Now that you’ve successfully written a simple timer, create a small function to test your timer. Your LEDs can be programmed in many ways, but the main point here is for you to know how to configure and modify the various aspects of a timer (counter, frequency etc.). A simple example has been given to you, that toggles the LEDs on and off in various colours every 500ms.

Progress Check: Run the program. You **do not** need to demo this part to the TAs. This is simply a soft check to ensure that you are able to configure and program a timer successfully.

**Lab 2.2 – PWM Programming**

In this activity we will look at using PWM to program the RGB LEDs on the FRDM-MCXC444 board.

As we have seen in the lab lecture, PWM generation heavily rely on the use of timer. The MCXC444 have a total of 3 timer/PWM modules (TPMs). We will be using TPM0 to generate PWM signals on all 3 LEDs to cycle through (almost) all the colours the RGB LED can produce. TPM1 will be used as a periodic interrupt timer that updates the duty cycle of TPM0. This will achieve

Using a similar timer setup from the previous part, you should be able to set up the PWM.

For this activity, we will be using the **8MHz** clock, with a divisor of 1.

1. Programming the RGB LEDs

Fill in the following table to determine which channels of which TPM to use, and which MUX ALT value to use. You can find this information in the Reference Manual.

**Question 2.1**.(2 marks)

Fill in the details below about the GPIO MUX ALT values, which TPM to use and which channels to use for each of the LEDs.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **LED** | **PIN** | **ALT** | **TPMx (TPM0, 1 or 2)** | **Channel** |
| RED | PTE31 |  |  |  |
| GREEN | PTD5 |  |  |  |
| BLUE | PTE29 |  |  |  |

In this case, we no longer control the LEDs via PTE31/PTD5/PTE29 respectively, but rather using the respective TPMx channel.

**Question 2.2**. (2 marks)

Recall that the RGB LEDs are active low:

A diagram of a led

AI-generated content may be incorrect.

Based on this, should the PWM signal be High-True or Low True? Explain your answer.

After completing lab 2.2, we will achieve the following:

1. The red LED will fade on, then fade off. This is followed by green and blue and the cycle repeats itself.
2. When we press SW3, the PWM frequency will change between 250Hz and 20Hz

For question 2.3, we will start off with **PWM frequency of 250Hz**. In question 2.5, we will explore the effect of different PWM frequency on the behaviour of the LED. For question 2.3, we will use the pre-scalar of 128 for setting up of TPM0.

Complete the following functions in your code:

**Question 2.3.** (7 marks)

void initPWM() {

//turn on clock gating to TPM0

//turn on clock gating to PORTD and PORTE

//configure the RGB LED MUX to be for PWM

//set pins to output (good practice)

//the following code is used to setup TPM0

//turn off TPM0 by clearing the clock mode

//clear and set the prescalar

//set centre-aligned PWM mode

//initialize count to 0

//choose and initialize modulo

//Configure TPM0 channels.

//IMPORTANT: Configure a REVERSE PWM signal!!!

//i.e. it sets when counting up and clears when counting

//down. This is because the LEDs are active low.

}

void startPWM() {

//set CMOD for TPM0

}

void stopPWM() {

//mask CMOD for TPM0

}

void setPWM(int LED, int percent) {

//convert percent into a value

//switch(LED) {

// case(RED):

// set TPM0 control value

//repeat for BLUE and GREEN

// default: printf(“invalid LED.\r\n”);

//}

}

1. Programming the TPM1 to control duty cycle of TPM0

We will now program timer TPM0 which will be used to control the duty cycle of TPM1 (This will affect how fast/slow the LED changes colour.) The timer interrupt should trigger every 10ms (Using PS of 128). While you do not need to write the interrupt, it is important for you to understand the function TPM1\_IRQHandler() to be able to understand how the entire system works. (**Note**: These code snippets are similar to activity 2.1 but are not entirely the same.)

**Question 2.4.** (3 marks)

void initTimer() {

//disable TPM1 interrupt

//setMCGIRClk();

//turn on clock gating to TPM1

//Choose 8MHz MCGIRCLK

//Turn off TPM1 and mask the prescalar

//Set prescalar of 128, and enable TOIE

//Initialize the counter to 0

//Initialize modulo

//Set the priority to the highest & enable IRQ

}

void startTimer() {

//Set TPM Clock Mode to increment on every TPM counter clock.

}

void stopTimer() {

//Turn off the timer

}

We will now experiment with using a very low PWM frequency to examine its effect.

**Question 2.5**.(1 mark)

Modify your code to set the PWM frequency to 20 Hz. Copy and paste the function: PORTA\_IRQHandler() to this box.

Compare with a frequency of 250 Hz. What difference do you see in the way the LEDs light up?

Progress check: Congratulations on making it this far! At this point, your LEDs should be able to increase and decrease in brightness on its own, change colours and toggle between 250Hz and 20Hz using a button. Again, you **do not** need to demonstrate this part to the TAs, but you will need this part for the next activity, so do not skip this.

**Lab 2.3 – Analog-Digital Conversion**

In Lab 2.2 we saw how to generate “analog” outputs using PWM, and in this activity we will look at the opposite side of things; we will see how to take analog inputs and convert them to digital values.

In this activity, how ADC can be used to control various peripherals on our board, we will be using the joystick provided in our 45-in-1 sensor kit. It looks like this:

A black object with a round black object on it

AI-generated content may be incorrect.

The 5V and GND lines supply power to the joystick, which use variable resistors to function as voltage dividers. We can read the resulting voltages from VRX and VRY.

Connect the 5V pin to 3.3V (**Note**: **Do not** connect it to the 5V pin!), and GND to any of the GND pins on the MCXC444. We will be exploring how to connect VRX and VRY later.

**Question 3.1**.(2 marks)

With a reference voltage of 3.3v and 12-bit resolution, what is the ADC’s accuracy in volts? This is defined to be the smallest voltage that the ADC can measure to produce a non-zero result. Give your answer to 5 decimal places.

Repeat the calculation for 8-bit resolution. Give your answer to 5 decimal places.

**Question 3.2**.(2 marks)

Based on your calculations in Question 1, state ONE advantage of using 12-bit resolution over 8-bit.

State ONE disadvantage of using 12-bit resolution over 8-bit resolution (Hint: See Table 23-7 in your Reference Manual.

In the Reference Manual, you can find that there are multiple modes of ADC output. One such option is averaging, which we have disabled in this example.

**Question 3.3**.(2 marks)

Consult the Reference Manual and state how many samples the ADC can average over.

State ONE advantage and ONE disadvantage of doing averaging.

Now, we will be completing the wiring for the Joystick, which can be done in multiple ways.

**Question 3.4**.(2 marks)

By examining the pin assignment table, determine 2 suitable pins to use for ADC inputs, and write down the corresponding ADC Channels (e.g. ADC0\_SE2a, ADC1\_SE5, etc).

In the file given for this activity (as02\_3.c), you will be given some template code, and are required to fill in initADC and startADC, as well as the IRQHandler. In order to handle multiple channels, you need to switch off continuous conversion.

The idea behind converting multiple channels is:

* + Start converting the current channel
  + Wait for the interrupt. Within the ISR:
    - Read the result of the conversion
    - Start converting the next channel

Since we are interested in converting between multiple channels, we create a 2-element array to store the results (this has been done for you)

**Question 3.5**.(2 marks)

In this case, we do not turn on continuous conversion. If we want to continuously convert samples as quickly as possible, where is the best place to turn it back on?

**Question 3.6**.(5 marks)

Complete the following functions:

void initADC() {

    // Disable & clear interrupt

    // Enable clock gating to relevant configurations

    // Set pins from Q3 to ADC

    // Configure the ADC

    // Enable ADC interrupt

    // Select single-ended ADC

    // Set 12 bit conversion

    // Select software conversion trigger

    // Configure alternate voltage reference

    // Don't use averaging

    // Switch off continuous conversion.

    // Set highest priority

}

void startADC(int channel) {

//mask and set the channel

}

int result[2];

//IMPORTANT: Change the n below to the right ADC channel

//ADC\_XXX and ADC\_YYY are the ADC Channels identified in Q3

void ADCn\_IRQHandler() {

//static int turn = 0;

//clear pending IRQ

//if conversion is complete, result[turn] = result register

//turn = 1 – turn;

//if (turn) {

// startADC(ADC\_XXX);

//} else {

// startADC(ADC\_YYY);

//}

}

**Question 3.7**. (2 marks)

Why is turn declared as “static”? What does “static” mean?

**Demo. (3 marks)**

Create a project to showcase the RGB LEDs changing with the inputs from the joystick. Demonstrate your code to your TA. We will mark **solely** on the following requirements, and any extension is not required, although fun.

Demo requirements:

* Joystick should be able to finely control values of 2 out of 3 LEDs (R/G/B)
* Button interrupt to toggle between 20Hz and 250Hz