CG2271 Real Time Operating Systems

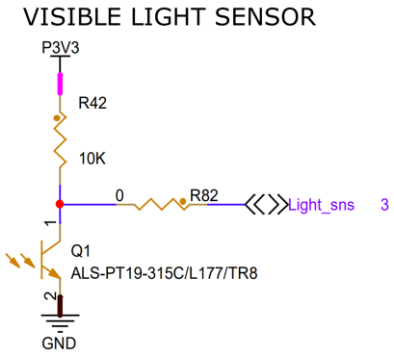
Lab 5 – Analog-Digital Conversion

Introduction

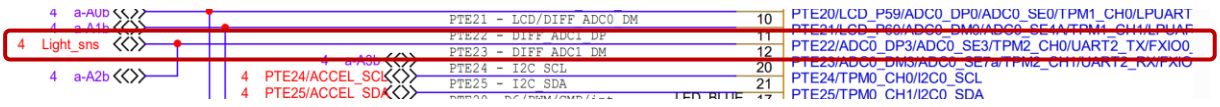
In Lab 4 we saw how to generate "analog" outputs using PWM, and in this lab we will look at the opposite side of things; we will see how to take analog inputs and convert them to digital values.

1. Running the Sample Code

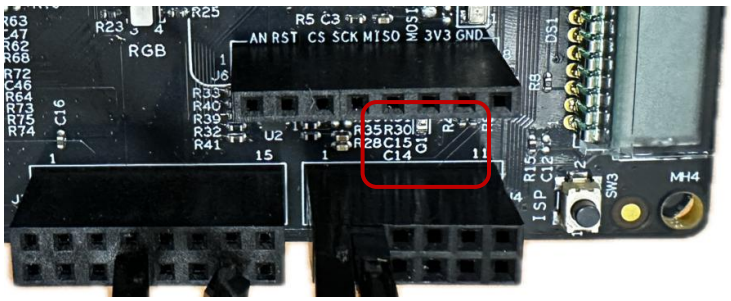
Our sample code reads the visible light sensor on the FRDM-MCXC444 board:



Examining the board schematics we can see this sensor is connected to ADC0_SE3:



You can locate the sensor labelled "Q1" near the bottom of the board (it is very tiny):



You are provided with a file called "adc.c". Create a new C/C++ project called ADCDemo, then clear the contents of ADCDemo.c, and copy everything adc.c

over. This code controls the brightness of the RGB LEDs using the readings taken from the visible light sensor.

Let's examine initADC. Most of the code has been explained in the lab lecture, so this is mostly to help you see how everything fits together:

We begin as always by disabling interrupts and clearing any existing ones, then switching on the clock gating to the ADC and PORTE, and setting PTE22 to the ADC function (ALT0):

```
// Configure interrupt
NVIC_DisableIRQ(ADC0_IRQn);
NVIC_ClearPendingIRQ(ADC0_IRQn);

// Enable clock gating to ADC0
SIM->SCGC6 |= SIM_SCGC6_ADC0_MASK;

// Enable clock gating to PTE
// This is done when we initialize the PWM
// but we want to make our function self-contained
// so we do it again
SIM->SCGC5 |= SIM_SCGC5_PORTE_MASK;

// Set PTE22 (ADC0_DP3) to ADC
PORTE->PCR[ADC_PIN] &= ~PORT_PCR_MUX_MASK;
PORTE->PCR[ADC_PIN] |= PORT_PCR_MUX(0);
```

We now turn on the ADC interrupt, select single-ended ADC, and 12 bit resolution:

```
// Configure the ADC
// Enable ADC interrupt
ADC0->SC1[0] |= ADC_SC1_AIEN_MASK;

// Select single-ended ADC
ADC0->SC1[0] &= ~ADC_SC1_DIFF_MASK;
ADC0->SC1[0] |= ADC_SC1_DIFF(0b0);

// Set 12 bit conversion
ADC0->CFG1 &= ~ADC_CFG1_MODE_MASK;
ADC0->CFG1 |= ADC_CFG1_MODE(0b01);
```

Question 1. (4 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 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

We configure the ADC to use software triggering and to use the alternative voltage reference, which is needed to drive the ADC pins on the FRDM-MCXC444 board, and turn off averaging:

```
// Use software trigger
ADC0->SC2 &= ~ADC_SC2_ADTRG_MASK;

// Use VALTH and VALTL
ADC0->SC2 &= ~ADC_SC2_REFSEL_MASK;
ADC0->SC2 |= ADC_SC2_REFSEL(0b01);

// Don't use averaging
ADC0->SC3 &= ~ADC_SC3_AVGE_MASK;
ADC0->SC3 |= ADC_SC3_AVGE(0);
```

Question 3. (4 MARKS)

Consult the Reference Manual and state how many samples the ADC can average over. (2 marks)

State ONE advantage and ONE disadvantage of doing averaging. (2 marks)

Finally we turn on continuous conversion mode, set the interrupt priority level and turn on interrupts:

```
// Use continuous conversion
ADC0->SC3 &= ~ADC_SC3_ADCO_MASK;
ADC0->SC3 |= ADC_SC3_ADCO(1);


// Lowest priority
NVIC_SetPriority(ADC0_IRQn, 192);
NVIC_EnableIRQ(ADC0_IRQn);
```

Question 4. (3 MARKS)

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

We declare our interrupt handler ADC0_IRQHandler, where we check that the COCO bit in ADC0->SC1[0] is set, then reading the result from ADC0->R[0] and using the results to light the LEDs.

```
void ADC0_IRQHandler() {
    NVIC_ClearPendingIRQ(ADC0_IRQn);

    if(ADC0->SC1[0] & ADC_SC1_COCO_MASK){
        int result = ADC0->R[0];
        PRINTF("IRQ: Value = %d\r\n", result);
        int val = (int)((4096 - result) / 4096.0 * 100);
        setPWM(BLUE, val);
        setPWM(RED, val/2);
        setPWM(GREEN, ((val-256) > 0 ? val-256 : 0));
    }
}
```

(Notice that we do $(4096 - result)$ (circled above) before scaling it to 100. This is because the light sensor produces a LOWER voltage at ADC0_SE3 with MORE light. This subtraction reverses this so that while the voltage is still lower with increasing light, the result is higher)

Finally we turn on the ADC by writing a 3 to the ADCH bits of ADC0->SC1[0], which, in single ended mode, chooses ADC0_SE3.

```
void startADC(int channel) {
    PRINTF("Started ADC on channel %d. Waiting\r\n", channel);

    ADC0->SC1[0] &= ~ADC_SC1_ADCH_MASK;
    ADC0->SC1[0] |= ADC_SC1_ADCH(channel);
}
```

Compile your program, and send it to the board using Debug. Observe how the LED changes when you cover the sensor, or when you shine a light (e.g. from your mobile phone) on it.

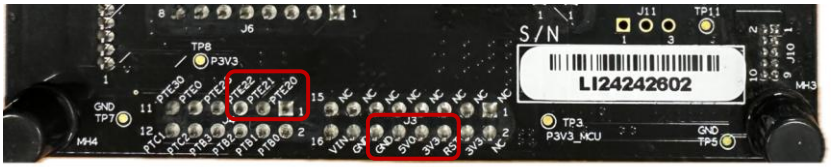
2. Connecting the Joystick

The ADC on the MCXC444 can only read from one channel at a time. We will now learn how to read from multiple ADC channels, albeit still one at a time.

i) Locate the joystick from your component kit. It will look like this:

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

ii) Connect the pin labelled 5V to the 3.3V pin on the board (DO NOT CONNECT IT TO THE 5V PIN!), and connect the VRX and VRY pins to pins PTE20 and PTE21 (VRX can go to PTE20 and VRY can go to PTE21, but it's not actually important what goes where except that you connect them to PTE20 and PTE21). Look under the board for where the PTE20 and PTE21, 3.3v (3V3)and GND pins are (they are close to the bottom of the board)



Question 5. (2 marks)

Which ADC channels correspond to pins PTE20 and PTE21? Write your answer as ADC0_SE2a, ADC1_SE5, etc.

3. Completing the Code

You are given a file called joystick.c, which contains the skeleton that is supposed to read the joystick and control the RGB LED.

- Create a new C/C++ project called CG2271Lab5.
- Remove the contents of CG2271Lab5.c, then copy over the contents of joystick.c.
- Complete the initADC and startADC functions. To be able to handle multiple channels you need to **SWITCH OFF** continuous conversion.

The idea behind converting multiple channels works like this:

- 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 want to save the results of both channels, we expand the "result" variable into a two-element array (this has been done for you).

```
int result[2];
```

To track which channel we are currently converting, we create a static variable called turn.

Question 6. (2 MARKS)

Why is turn declared as "static"? What does "static" mean?

Our ADC0_IRQHandler now looks like this:

```
void ADC0_IRQHandler(){
    static int turn=0;

    NVIC_ClearPendingIRQ(ADC0_IRQn);
    if(ADC0->SC1[0] & ADC_SC1_COCO_MASK) {
        // Read the result into result[turn]
        result[turn] = /* Statement to read result */
        turn = 1 - turn;
        if(turn == 0) {
            // Call startADC to convert PTE20
        } else {
            //Call startADC to convert PTE21
        }
    }
}
```

Complete this handler.

Demo. (3 MARKS)

The RGB LEDs will change as you manipulate the joystick. Demonstrate your code to your TA.