EENG461, LAB3 report

Mark Clark and Jeremy Munson

Objective This lab has 3 tasks: Read values from an ADC, Dim an LED with the PWM output, and take input from a GPIO using an interrupt. The ADC reading is to be used to set the duty cycle of the PWM. The GPIO switch input is to be used to print information about the current ADC value read.

Introduction The overall system is simple, and we were able to re-use our code for button debouncing from the previous lab. As always, we are doing this lab using GCC and NOT code-composer-studio. This presented an additional task: Using uart to print the information instead of CCS provided printf. Each task is a matter of configuring the peripherals, then using the main loop to coordinate the tasks and perform calculations.

Narrative

ADC For the ADC, we configured the ADC peripheral and pin in the aptly named function ADCPinConfigure(void). This required using the alternate function selection of the pin.

```
void ADCPinConfigure(void) {
    SYSCTL_RCGCADC_R |= SYSCTL_RCGCADC_R0;                          //Enable ADC Clock
    while(!(SYSCTL_PRADC_R & SYSCTL_PRADC_R0)) {};                  //Wait for
     ↪  peripheral to be ready

    SYSCTL_RCGCGPIO_R |= SYSCTL_RCGCGPIO_R4;                        //Enable GPIO Pin
 ↪  for ADC (PE5)
    while(!(SYSCTL_RCGCGPIO_R & SYSCTL_RCGCGPIO_R4)) {};            //Wait fo peripheral
     ↪  to be ready

    GPIO_PORTE_AFSEL_R |= GPIO_PIN5;                                //Set Alternate
 ↪  Function Select
    GPIO_PORTE_DEN_R &= ~GPIO_PIN5;                                 //Clear Digital
 ↪  Enable for Pin 5
    GPIO_PORTE_AMSEL_R |= GPIO_PIN5;                                //Set Alternate Mode
 ↪  Select
}
```

Next, we have to configure the sample sequencer. We use a timer to trigger the ADC samples. We also unmask the interrupt in the NVIC here.

```
void ADCSampleSequencerConfigure(void) {

    ADC0_ACTSS_R &= ~ADC_ACTSS_ASEN3;                               //Disable Sequencer
 ↪  3
    ADC0_EMUX_R |= ADC_EMUX_EM3_TIMER;                              //Set ADC as Timer
 ↪  Triggered
```

```c
    ADC0_SSMUX3_R |= 0x8;                                       //Enable AIN8
    ADC0_SSCTL3_R |= ADC_SSCTL3_IE0 | ADC_SSCTL3_END0;          //Sequencer control
    ADC0_SAC_R = 0x6;                                           //Enables x64
 ↪  Oversampling
    ADC0_ISC_R |= ADC_ISC_IN3;                                  //Clear Interrupt
    ADC0_IM_R |= ADC_IM_MASK3;                                  //Enable Interrupt
    NVIC_EN0_R |= 1 << (INT_ADC0SS3 - 16);                      //Enable NVIC for
 ↪  ADC0 Sequencer 3

    configureAdcTimer();

    ADC0_ACTSS_R |= ADC_ACTSS_ASEN3;                            //Enable Sequencer
}
```

The interrupt itself is very short. This function is added to the NVIC table in the appropriate place. We store the ADC reading in a global `potReading`. We also mask off the ADC FIFO register so that we only read the values we intended to.

```c
volatile uint16_t potReading;

void saveADCSample(void){

    ADC0_IM_R &= ~ADC_IM_MASK3;                                 //Disable Interrupt
    ADC0_ISC_R |= ADC_ISC_IN3;                                  //Clear Interrupt

    potReading = (ADC0_SSFIFO3_R & ADC_SSFIFO3_DATA_M);         //Read Potentiometer
 ↪  Value

    ADC0_IM_R |= ADC_IM_MASK3;                                  //Enable Interrupt
}
```

PWM The PWM is configured in a function PWMConfigure(). We enable the peripheral and the pin, then configure the pin to the alternate function for PWM. We configure the PWM peripheral and start it.

```c
void PWMConfigure(void) {
    SYSCTL_RCGCPWM_R |= SYSCTL_RCGCPWM_R1;                                  //Enable
 ↪  PWM Module 1
    while(!(SYSCTL_PRPWM_R & SYSCTL_PRPWM_R1)) {}                           //Wait
     ↪  for peripheral to be ready

    SYSCTL_RCGCGPIO_R |= SYSCTL_RCGCGPIO_R5;                                //Enable
 ↪  GPIO Port F
    while(!(SYSCTL_PRGPIO_R & SYSCTL_PRGPIO_R5)) {}                         //Wait
     ↪  for peripheral to be ready

    GPIO_PORTF_AFSEL_R |= (1 << 1);                                         //Set
 ↪  Alternate Function for PF1
    GPIO_PORTF_PCTL_R |= GPIO_PCTL_PF1_M1PWM5;                              //Set
 ↪  Port Control to PF1 PWM value
    GPIO_PORTF_DEN_R |= (1 << 1);                                           //Set
 ↪  Digital Enable for PF1
```

```
    PWM1_2_CTL_R = 0x0;
 ↪  //Disable PWM1 Gen2
    PWM1_2_GENB_R = PWM_2_GENB_ACTCMPBD_ONE | PWM_2_GENB_ACTLOAD_ZERO |
 ↪  PWM_2_GENB_ACTZERO_ONE;
                                                                      //Set
                                                                       ↪  PWM1
                                                                       ↪  Gen2-B
                                                                       ↪  comparator,
                                                                       ↪  load,
                                                                       ↪  and
                                                                       ↪  zero

    PWM1_2_LOAD_R = CYCLES_PER_MS;                                    //Set
 ↪  PWM Gen2 Period to 1ms
    PWM1_2_CMPB_R = 0x0;                                              //Set
 ↪  Comparator value
    PWMEnable();                                                      //Enable
 ↪  PWM
}
```

We then created a series of utility functions to handle the PWM. These are used in the main loop as needed. These configure the period, duty cycle, and turn it on/off.

```
void PWMSetPeriod(uint16_t cycles_per_period) {
    PWMDisable();
 ↪  //Disable PWM
    PWM1_2_LOAD_R = cycles_per_period;                                //Set
 ↪  new period
    PWMEnable();                                                      //Enable
 ↪  PWM
}

void PWMSetDutyCycle(uint8_t duty_cycle) {
    PWMDisable();
 ↪  //Disable PWM
    PWM1_2_CMPB_R = CYCLES_PER_MS_DIV_100 * (duty_cycle);             //Set
 ↪  new duty cycle
    PWMEnable();                                                      //Enable
 ↪  PWM
}

void PWMEnable(void) {
    PWM1_2_CTL_R |= PWM_2_CTL_ENABLE;                                 //Enable
 ↪  PWM1 Gen 2
    PWM1_ENABLE_R |= PWM_ENABLE_PWM5EN;                               //Enable
 ↪  PWM1 Output 5
}

void PWMDisable(void) {
    PWM1_2_CTL_R &= ~PWM_2_CTL_ENABLE;
 ↪  //Disable PWM1 Gen 2
```

```
    PWM1_ENABLE_R &= ~PWM_ENABLE_PWM5EN;
 ↪  //Disable PWM1 Output 5
}
```

Timers The timer code was largely just tweaked from the pervious lab. We set it as the ADC trigger, and removed the timer ISR.

```
void configureAdcTimer (void) {

    SYSCTL_RCGCTIMER_R |= SYSCTL_RCGCTIMER_R0; //Enable Run Mode Clock Gating
 ↪  Control for Timer 0

    while (!(SYSCTL_PRTIMER_R & SYSCTL_RCGCTIMER_R0)) {}

    TIMER0_CTL_R &= ~TIMER_CTL_TAEN; //Disable Timer
    TIMER0_CTL_R |= TIMER_CTL_TASTALL; //Stall for debug
    TIMER0_CFG_R = TIMER_CFG_32_BIT_TIMER;
    TIMER0_TAMR_R |= TIMER_TAMR_TAMR_PERIOD; //Set Timer to count down periodically
    TIMER0_TAILR_R = 16000 - 1;
    TIMER0_CTL_R |= TIMER_CTL_TAOTE; //Set as an ADC Trigger
    TIMER0_CTL_R |= TIMER_CTL_TAEN; //Enable Timer
}
```

Printing Since this lab didn't really intend to have UART and implementing printf as one of the challenges, We're not going to report on it in detail. The code is available in uart_print.c and it's header.

There were some glitches encountered with the ICDI virtual com port provided on the board - random bytes would be lost. This was "solved" by inserting a ~1-2 cycle dead time between data frames (Effectively ensuring 2 stop bits or more) and by limiting transmission to 8 bytes at a time. After bytes, the transmitter waits for ~4 data frame periods. This seems to totally prevent any lost data.

We do the printing during the main loop - the debounced switch ISR sets a global flag so that main can take care of it later.

```
static void sw1_action(void) {
    NEED_PRINT = true;
}
```

Integration In main we call of the configuration functions from above. We then start a loop to perform the ADC to PWM duty cycle calculations and print if necessary.

```
    while (1) {

        //Calculate a corresponding duty cycle percentage
        uint16_t temp_duty_cycle = (uint16_t)((potReading*100)/4095);

        /*
         * Only set a new duty cycle if the potentiometer value changed
         */
        if (temp_duty_cycle != duty_cycle_last) {
            PWMSetDutyCycle(temp_duty_cycle);
            duty_cycle_last = temp_duty_cycle;
        }

        if(NEED_PRINT) {
            float voltage = ((float)potReading*3.3f)/4095.0f;
            printlf("The current ADC value is %d and the DC is %f \n\r", potReading,
↪   &voltage);
            NEED_PRINT = false;
        }

    }
```

It was observed that even when connected directly to ground, the ADC would not read '0'. Instead we read ~8-20. This has the effect of preventing the PWM duty cycle reaching 0. Similarly, the ADC was reluctant to reach it's maximum value. Use of a debugger indicates that the raw ADC reading is indeed >0, so this is out of our control.

The program behaved as expected and the LED dimming and ADC value printing worked perfectly.

Concluding Remarks All of the core parts of the lab were completed quickly and easily. We spent more time chasing down the strange UART issue than anything else. There was a slightly strange behaviour from the ADC, not reaching 0, but that does not appear to be something within our control.