EE4930 Advanced Embedded Systems

Section 011, Winter 2019/20

Draven Schilling

2-24-20

Final Exam – RTOS System Phase 2

# Objectives

The objective for this lab was to expand our implementation of the MSP432 RTOS (sys/bios). We were to build an RTOS system which uses tasks, events, a hardware interrupt, and a software interrupt to make a system which uses an ADC to take readings from a potentiometer which then converts the reading to a temperature setpoint value between 50-90F and displays the setpoint to the LCD. Concurrently, a temperature sensor reads the surrounding temperature and displays the results to the LCD. If the temperature is less than 5 degrees from the setpoint a fan and heating coil are to be turned on. Once the temperature is 5 above the setpoint, then it is to turn off.

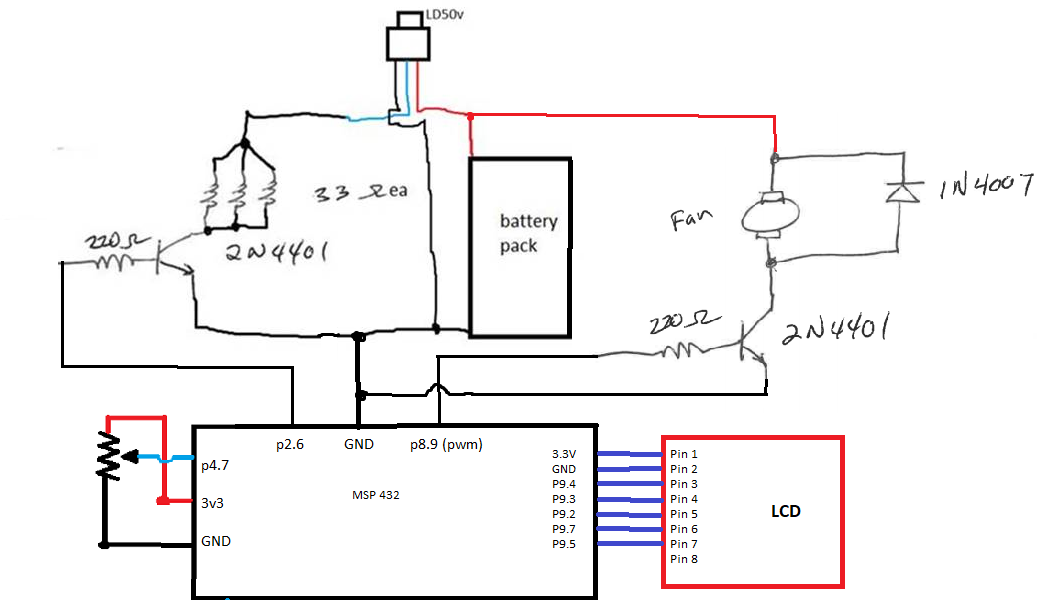
# Description

For this lab, I started by first using the code from the pervious RTOS lab as a base since it already implemented the structures for the events, tasks, and interrupts. I would also reuse the code for the setpoint potentiometer. Next, I went on to view the example CCS code for the I2C sensor and imported the functionality I would need for the tmp102 sensor. After fenagling with some of the settings and re-configuring sysconfig, I got the connection successfully established. Next, I implemented the correct conversion formula, setup a sampling period of ~0.5 sec, and printed the temperature value to the LCD. Now with the setpoint and temperature readings I was able to create a task for managing the heater/fan. I setup some simple logic of when to toggle the heater/fan and tested the output. Since the fan was too strong with direct voltage, I decided to set it up on PWM using the driver library. I again imported the driver example, changed the period, and setup sysconfig and it definitely helped out regulating the fan output and was simple to use. Lastly was the circuit setup; and truth be told, this was the hardest part for me. First I learned that I needed the external battery pack with the 5V regulator to supply the input voltage of the transistor. I also then had a problem not providing a “common” ground that connected between the battery pack and the MSP432. Once the circuit was complete, I tested and verified functionality.

# Conclusions

In conclusion, this lab was very interesting. I learned a lot about how the structures of an RTOS works. The main difficulties I encountered were solving hardware connection problems, which actually wasn’t that bad. I also had some minor problems with linker include errors which somehow was fixed by creating a new project. More than anything though, with the experience I gained from the I2C drivers I was able to apply what I learned to also find a driver solution to provide a PWM signal to the fan which was very helpful. I think this lab serves as a good application for the basic RTOS features because it combines many RTOS features with multiple tasks. Without an RTOS or driver library, I believe this lab would be MUCH more challenging to complete because there are just so many features to manage.

# Attachments



**Source Code:**

**#define** \_\_MSP432P401R\_\_

**#include** "msp.h"

/\* POSIX Header files \*/

**#include** <pthread.h>

**#include** <stdio.h>

**#include** <string.h>

**#include** <stdint.h>

**#include** <stddef.h>

**#include** <unistd.h>

**#include** "msoe\_lib\_clk.h"

**#include** "msoe\_lib\_lcd.h"

**#include** "msoe\_lib\_delay.h"

**#include** <ti/drivers/GPIO.h>

**#include** <ti/drivers/I2C.h>

**#include** <ti/drivers/PWM.h>

**#include** "ti\_drivers\_config.h"

/\* XDC module Headers \*/

**#include** <xdc/std.h>

**#include** <xdc/runtime/Error.h>

**#include** <xdc/runtime/System.h>

**#include** <xdc/runtime/Timestamp.h>

/\* BIOS module Headers \*/

**#include** <ti/sysbios/BIOS.h>

**#include** <ti/sysbios/knl/Clock.h>

**#include** <ti/sysbios/knl/Swi.h>

**#include** <ti/sysbios/hal/Hwi.h>

**#include** <ti/sysbios/knl/Task.h>

**#include** <ti/sysbios/knl/Event.h>

**#include** <ti/drivers/Board.h>

**#define** TIMEOUT 12 /\* Timeout value \*/

**#define** CLKPERIOD 500 // number of clock ticks per adc trigger

// period of ~0.5 sec

**#define** TMP102\_ADDR = 0x48

**#define** TASKSTACKSIZE 1024

**#define** ADCINTERRUPT 40 // interrupt number for the ADC

**#define** ADC\_RANGE 4096 // for 12 bit conversions

**void** **initADC**();

**void** **initGPIOOutputs**();

Void **clkFxn**(UArg arg0);

Void **inputTask**(UArg arg0, UArg arg1);

Void **lcdTask**(UArg arg0, UArg arg1);

Void **i2cTask**(UArg arg0, UArg arg1);

Void **heatFanTask**(UArg arg0, UArg arg1);

Void **swiFxn**(UArg arg0, UArg arg1);

Void **hwiFxn**();

// create four tasks

Task\_Struct inputTaskStruct, lcdTaskStruct, i2cTaskStruct, heatFanTaskStruct;

Char inputTaskStack[TASKSTACKSIZE], lcdTaskStack[TASKSTACKSIZE], i2cTaskStack[TASKSTACKSIZE], heatFanTaskStack[TASKSTACKSIZE];

// create clock

Clock\_Struct clkStruct;

Clock\_Handle clkHandle;

// create two events

Event\_Struct readingAvailableEventStruct, setpointDisplayEventStruct;

Event\_Handle readingAvailableEventHandle, setpointDisplayEventHandle;

// create SWI

Swi\_Struct swiStruct;

Swi\_Handle swiHandle;

// create HWI

Hwi\_Struct hwiStruct;

Hwi\_Handle hwiHandle;

uint8\_t temperature = 0; //global temperature variable

uint16\_t reading = 0;

**int** **main**()

{

WDT\_A->CTL = WDT\_A\_CTL\_PW | WDT\_A\_CTL\_HOLD; // stop watchdog timer

// setup lcd

LCD\_Config();

LCD\_clear();

LCD\_home();

LCD\_contrast(10);

// init functions

Hwi\_enable();

initADC();

initGPIOOutputs();

/\* Construct BIOS Objects \*/

Task\_Params taskParams;

Clock\_Params clkParams;

Swi\_Params swiParams;

Hwi\_Params hwiParams;

/\* Call driver init functions \*/

Board\_init();

/\* Construct Task threads \*/

Task\_Params\_init(&taskParams);

taskParams.stackSize = TASKSTACKSIZE;

taskParams.priority = 4; // update internal temperature task is less important than interrupt methods

taskParams.stack = &inputTaskStack;

Task\_construct(&inputTaskStruct, (Task\_FuncPtr)inputTask, &taskParams, NULL);

taskParams.priority = 5; // lcd update is not that important

taskParams.stack = &lcdTaskStack;

Task\_construct(&lcdTaskStruct, (Task\_FuncPtr)lcdTask, &taskParams, NULL);

taskParams.priority = 3; // i2c task is most important task

taskParams.stack = &i2cTaskStack;

Task\_construct(&i2cTaskStruct, (Task\_FuncPtr)i2cTask, &taskParams, NULL);

taskParams.priority = 6; // updating the output task is least important task

taskParams.stack = &heatFanTaskStack;

Task\_construct(&heatFanTaskStruct, (Task\_FuncPtr)heatFanTask, &taskParams, NULL);

/\* Obtain event handlers \*/

Event\_construct(&readingAvailableEventStruct, NULL);

readingAvailableEventHandle = Event\_handle(&readingAvailableEventStruct);

Event\_construct(&setpointDisplayEventStruct, NULL);

setpointDisplayEventHandle = Event\_handle(&setpointDisplayEventStruct);

/\* setup clock \*/

Clock\_Params\_init(&clkParams);

clkParams.startFlag = TRUE;

Clock\_construct(&clkStruct, (Clock\_FuncPtr)clkFxn,

CLKPERIOD\*2, &clkParams);

clkHandle = Clock\_handle(&clkStruct);

Clock\_setPeriod(clkHandle, CLKPERIOD);

/\* setup SWI \*/

Swi\_Params\_init(&swiParams);

swiParams.arg0 = 1;

swiParams.arg1 = 0;

swiParams.priority = 2; // priority after HWI

swiParams.trigger = 0;

Swi\_construct(&swiStruct, (Swi\_FuncPtr)swiFxn, &swiParams, NULL);

swiHandle = Swi\_handle(&swiStruct);

/\* setup HWI \*/

Hwi\_Params\_init(&hwiParams);

hwiParams.eventId = 40;

hwiParams.priority = 1; // top priority

Hwi\_construct(&hwiStruct, 40, hwiFxn, &hwiParams, NULL);

hwiHandle = Hwi\_handle(&hwiStruct);

/\* start the BIOS \*/

BIOS\_start();

**return**(0);

}

//Initialize ADC6 on pin 4.7

//Interrupt upon completed conversion

**void** **initADC**(){

//setup pins 4.6 and 4.7 in analog mode

P4->SEL0 |= 0b1<<7;

P4->SEL1 |= 0b1<<7;

// start sampling on SC bit, source a timer, use SMCLK, single-channel single conversion mode

// 96 sample/hold time, turn core on

ADC14->CTL0 &= 0x0;

ADC14->CTL0 |= (1<<26) | (1<<21) | (0b101<<12) | (0b101<<8) | (1<<4);

// 12 bit resolution and use memory location 4 to start

ADC14->CTL1 &= 0xF0000000;

ADC14->CTL1 |= (0b10<<4) | (4<<16);

ADC14->MCTL[4] |= 0x6; // input on A6

ADC14->IER0 |= (1<<4); // enable interrupt for mem[4]

ADC14->CTL0 |= 0b10; // enable

}

// Initializes Outputs for both the fan and compressor

// initializes P2.4 as GPIO output (fan)

// initializes P2.6 as GPIO output (heater)

**void** **initGPIOOutputs**(){

// P2.4

P2->SEL0 &= ~(1<<4); // use GPIO function

P2->SEL1 &= ~(1<<4);

P2->DIR |= (1<<4); // make output

P2->OUT &= ~(1<<4); // setup output low to start

// P2.6

P2->SEL0 &= ~(1<<6); // use GPIO function

P2->SEL1 &= ~(1<<6);

P2->DIR |= (1<<6); // make output

P2->OUT &= ~(1<<6); // setup output low to start

}

/\*

\* Clock function triggers an ADC conversion every ~0.5 sec

\*/

Void **clkFxn**(UArg arg0)

{

ADC14->CTL0 |= 1; // start ADC conversions

}

/\*

\* This task updates the LCD with the stored temperature value

\* whenever an event is triggered

\*/

Void **lcdTask**(UArg arg0, UArg arg1)

{

UInt posted;

**while**(1){

//wait for event

posted = Event\_pend(setpointDisplayEventHandle,

Event\_Id\_00,

Event\_Id\_NONE,

BIOS\_WAIT\_FOREVER);

**if** (posted == 0) {

System\_printf("Timeout expired for Event\_pend()\n");

**break**;

}

// update the LCD if the correct event trigger was posted.

**if** (posted & Event\_Id\_00) {

LCD\_goto\_xy(0,0);

LCD\_print\_str("Setpt:");

LCD\_print\_udec3(temperature);

LCD\_print\_str(" F");

LCD\_goto\_xy(0,1);

LCD\_print\_str("TMP:");

LCD\_print\_udec3(reading);

LCD\_print\_str(" F");

}

}

}

/\*

\* This task waits on a new temperature value to be obtained from the ADC

\* then determines if it is different than the current temperature displayed

\* and if it is different, posts and event to signal the LCD should update.

\*/

Void **inputTask**(UArg arg0, UArg arg1)

{

UInt posted;

uint8\_t old = 0;

uint8\_t old\_reading = 0;

**while**(1) {

// waits for the correct event to signal a new conversion is completed

posted = Event\_pend(readingAvailableEventHandle,

Event\_Id\_00,

Event\_Id\_NONE,

BIOS\_WAIT\_FOREVER);

**if** (posted == 0) {

System\_printf("Timeout expired for Event\_pend()\n");

**break**;

}

// signal to LCD task if the new temperature value is different than whats currently displayed

**if** (posted & Event\_Id\_00) {

**if**(temperature != old || (reading != old\_reading)){

old = temperature;

old\_reading = reading;

Event\_post(setpointDisplayEventHandle, Event\_Id\_00);

}

}

}

BIOS\_exit(0);

}

/\*

\* Hardware Interrupt function

\* called when a new ADC value is ready.

\* signals to a SWI when a conversion is ready

\*/

Void **hwiFxn**(){

uint32\_t adc\_val = ADC14->MEM[4]; //clear the interrupt by reading the value

Swi\_post(swiHandle); // signal to the SWI that a new conversion is ready

}

/\*

\* Software interrupt function

\* Reads the ADC value, converts it to the temperature range 50-90F

\* uses an event to signal to the inputTask that a new conversion has been read

\*/

Void **swiFxn**(UArg arg0, UArg arg1)

{

temperature = (40-(uint8\_t)((((**float**)ADC14->MEM[4])/ADC\_RANGE)\*40))+50;

Event\_post(readingAvailableEventHandle, Event\_Id\_00);

}

Void **i2cTask**(UArg arg0, UArg arg1){

UInt posted;

uint16\_t tmp\_data;

uint8\_t rxBuffer[2];

uint8\_t neg = 0;

**I2C\_init**();

I2C\_Params params;

**I2C\_Params\_init**(&params);

params.bitRate = *I2C\_400kHz*;

I2C\_Handle i2cHandle = **I2C\_open**(0, &params);

I2C\_Transaction transaction = {0};

transaction.slaveAddress = 0x48;

transaction.readBuf = rxBuffer;

transaction.readCount = **sizeof**(rxBuffer);

transaction.writeCount = 0;

/\* Take samples forever \*/

**while**(1){

//get a new sample the same time the adc conversions finnish. ~0.5 sec

posted = Event\_pend(readingAvailableEventHandle,

Event\_Id\_00,

Event\_Id\_NONE,

BIOS\_WAIT\_FOREVER);

**if** (posted == 0) {

System\_printf("Timeout expired for Event\_pend()\n");

**break**;

}

// signal to LCD task if the new temperature value is different than whats currently displayed

**if** (posted & Event\_Id\_00) {

**if** (**I2C\_transfer**(i2cHandle, &transaction)) {

tmp\_data = (rxBuffer[0] << 4) | (rxBuffer[1] >> 4); // combine data result before multiplication

neg = (rxBuffer[0]>>7); // detect if negative

tmp\_data = (neg ? ~(tmp\_data-1) : tmp\_data); // convert 2's compliment to absolute value if negative

reading = (tmp\_data\*0.0625);//\*(9/5))+32; // convert to celcious

reading = ((((**float**)reading)\*9)/5)+32; // convert celcious to farenheight

reading = neg ? -reading : reading; // apply negative sign

System\_printf("reading: %d\n", tmp\_data);

} **else** {

System\_printf("I2C bus fault.\n");

}

}

}

**I2C\_close**(i2cHandle);

System\_printf("I2C closed\n");

BIOS\_exit(0);

}

/\*

\* Task for managing the heater / fan outputs

\*/

Void **heatFanTask**(UArg arg0, UArg arg1){

UInt posted;

//setup pwm

PWM\_Handle pwm;

PWM\_Params pwmParams;

uint32\_t dutyValue;

// Initialize the PWM driver.

**PWM\_init**();

// Initialize the PWM parameters

**PWM\_Params\_init**(&pwmParams);

pwmParams.idleLevel = *PWM\_IDLE\_LOW*; // Output low when PWM is not running

pwmParams.periodUnits = *PWM\_PERIOD\_HZ*; // Period is in Hz

pwmParams.periodValue = 13; // 1MHz

pwmParams.dutyUnits = *PWM\_DUTY\_FRACTION*; // Duty is in fractional percentage

pwmParams.dutyValue = 0; // 0% initial duty cycle

// Open the PWM instance

pwm = **PWM\_open**(0, &pwmParams);

**if** (pwm == NULL) {

// PWM\_open() failed

**while** (1);

}

dutyValue = (uint32\_t) (((uint64\_t) PWM\_DUTY\_FRACTION\_MAX \* 40) / 100);

**PWM\_setDuty**(pwm, dutyValue); // set duty cycle to 40%

**while**(1){

//P2->OUT |= ((1<<4)|(1<<6));

//update output the same time the adc conversions finish. ~0.5 sec

posted = Event\_pend(readingAvailableEventHandle,

Event\_Id\_00,

Event\_Id\_NONE,

BIOS\_WAIT\_FOREVER);

**if** (posted == 0) {

System\_printf("Timeout expired for Event\_pend()\n");

**break**;

}

**if** (posted & Event\_Id\_00) {

**if**((reading - 1) >= temperature){

//turn off

P2->OUT &= ~((1<<4)|(1<<6));

**PWM\_stop**(pwm);

} **else** **if**((reading + 1) <= temperature){

//turn on

P2->OUT |= (1<<6);

**PWM\_start**(pwm);

}

}

}

BIOS\_exit(0); //should never get here

}