ECE 3849 B2022

Real-Time Embedded Systems

Lab 2: RTOS, Spectrum Analyzer Report

Professor Stander

William Tyrrell

12/2/22

Introduction:

In this lab, we will be porting Lab 1, a 1 Msps digital oscilloscope, to run using a Real-Time Operating System (RTOS). Much of the code remains the same but needs to change to fit into the RTOS environment. The same 20kHz PWN square wave from lab 1 remains, including the adjustable voltage scale and rising and falling triggers. We will also be adding a spectrum mode that can be toggled using one of the on-board buttons. I was able to accomplish all these tasks but did not complete the extra-credit assignments to make Real-Time measurements to test if the system meets its deadlines.

Discussion and Results:

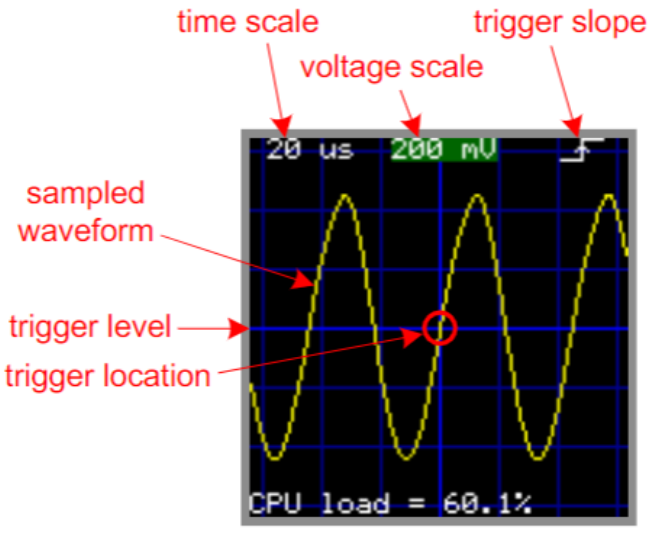

Figure 1: Sample Screen for Oscilloscope

.

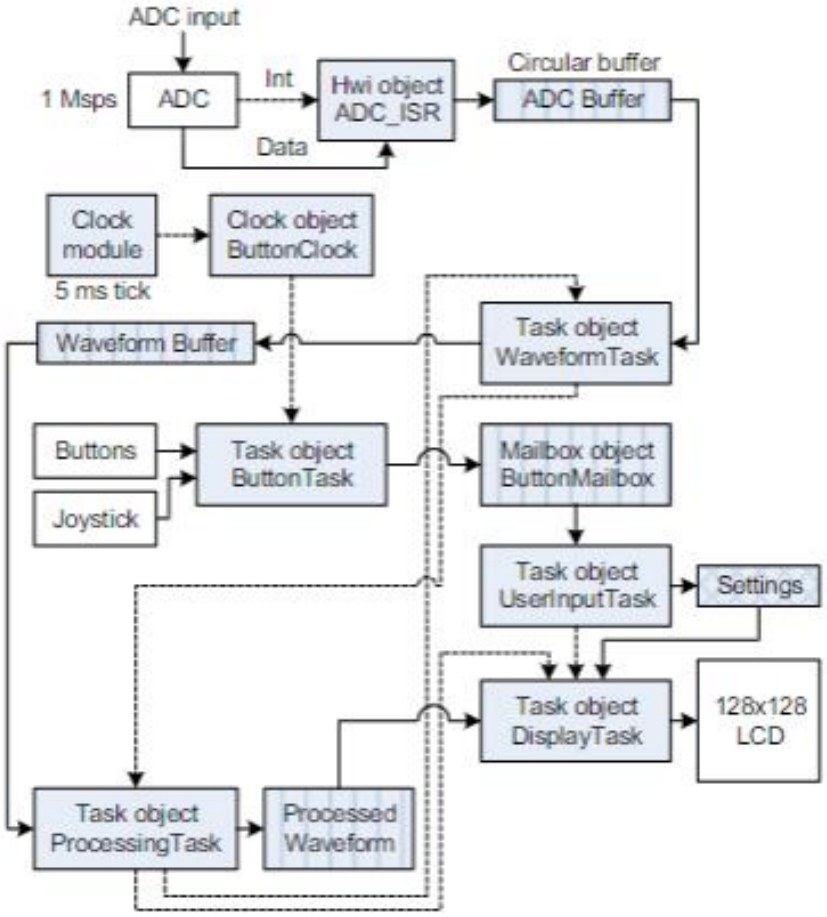

Figure 2: Software Block Diagram

Step 1: TI-RTOS Project:

The first step in this lab is to import the lab2_starter file which gives a skeleton to build the project from. We then have to copy over the .c and .h files as well as the code from main() in lab 1. We need to not copy over tm4c1294ncpdt_ccs.c as these setup steps will be taken care in the

RTOS setup. With the main() code from lab 1 we comment it out and will use pieces from it later to implement the functionality using tasks. The same jumper wire and pins from lab 1 are used pictured in figure 3 as well as the same code that creates the PWN input signal pictured in figure 4.

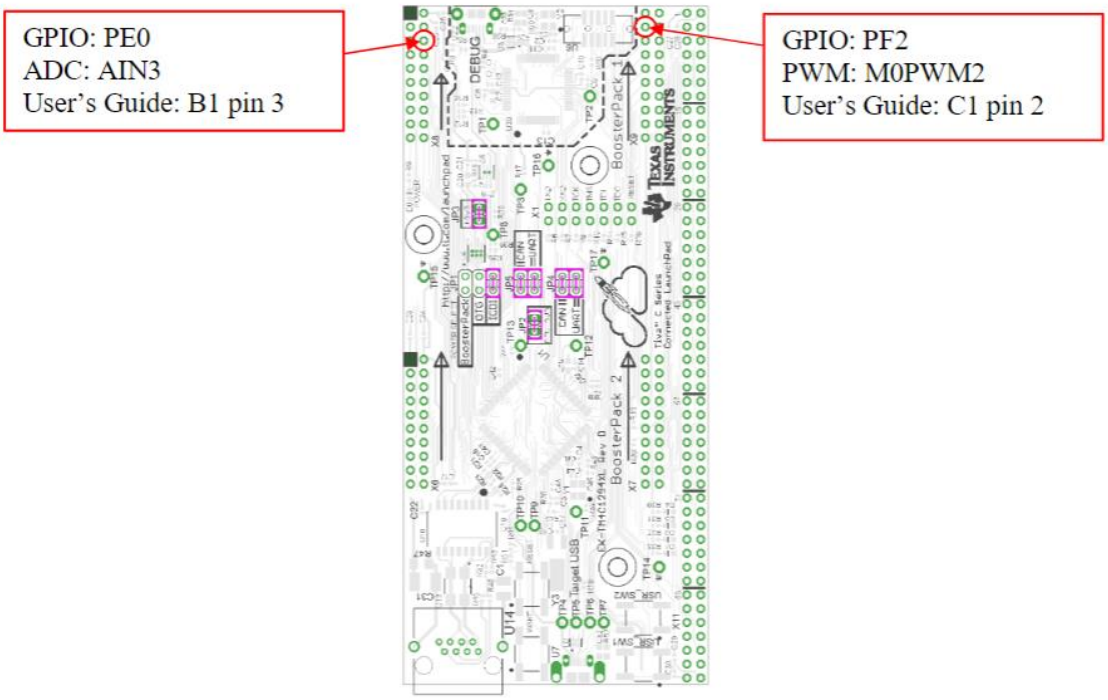

Figure 3: Source Voltage pins

```
// configure M0PWM2, at GPIO PF2, BoosterPack 1 header C1 pin 2
        // configure M0PWM3, at GPIO PF3, BoosterPack 1 header C1 pin 3
        SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
        GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_2 | GPIO_PIN_3);
        GPIOPinConfigure(GPIO_PF2_M0PWM2);
        GPIOPinConfigure(GPIO_PF3_M0PWM3);
        GPIOPadConfigSet(GPIO_PORTF_BASE, GPIO_PIN_2 | GPIO_PIN_3,
                         GPIO_STRENGTH_2MA, GPIO_PIN_TYPE_STD);

        // configure the PWM0 peripheral, gen 1, outputs 2 and 3
        SysCtlPeripheralEnable(SYSCTL_PERIPH_PWM0);
        PWMClockSet(PWM0_BASE, PWM_SYSCLK_DIV_1); // use system clock without division
        PWMGenConfigure(PWM0_BASE, PWM_GEN_1, PWM_GEN_MODE_DOWN | PWM_GEN_MODE_NO_SYNC);
        PWMGenPeriodSet(PWM0_BASE, PWM_GEN_1, roundf((float)gSystemClock/PWM_FREQUENCY));
        PWMPulseWidthSet(PWM0_BASE, PWM_OUT_2, roundf((float)gSystemClock/PWM_FREQUENCY*0.4f));
        PWMPulseWidthSet(PWM0_BASE, PWM_OUT_3, roundf((float)gSystemClock/PWM_FREQUENCY*0.4f));
        PWMOutputState(PWM0_BASE, PWM_OUT_2_BIT | PWM_OUT_3_BIT, true);
        PWMGenEnable(PWM0_BASE, PWM_GEN_1);
```

Figure 4: Source setup

Step 2: ADC Hwi:

In this step we need to configure the ADC ISR in the TI-TROS configuration. I created a M3 specific Hwi module in the GUI. This sets the interrupt vector table and priorities on its own and does not have to be done manually. The priority threshold for Hwi-disable() is set to 32 which leaves all higher priority ISR's enabled while the TI-RTOS runs the scheduler. The ADC hwi will be priority 0 making it a zero-latency interrupt thus being the highest priority and will execute every 1us without interference. To use the correct ADC, we need to set the interrupt number for ADC 1 from the TM4C1294NCPDT datasheet.

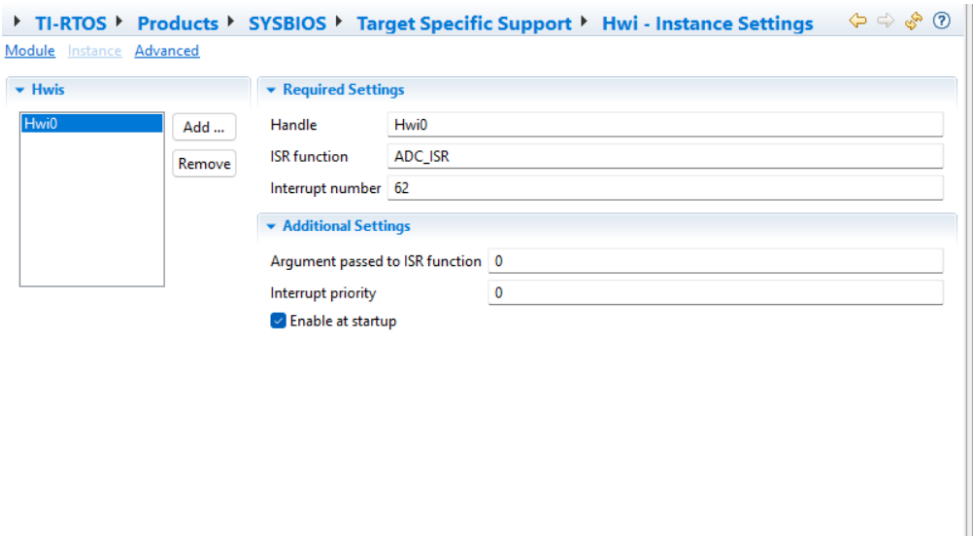

Figure 5: ADC Hwi setup

58	42	0x0000.00E8	USB MAC
59	43	0x0000.00EC	PWM Generator 3
60	44	0x0000.00F0	uDMA 0 Software
61	45	0x0000.00F4	uDMA 0 Error
62	46	0x0000.00F8	ADC1 Sequence 0
63	47	0x0000.00FC	ADC1 Sequence 1
64	48	0x0000.0100	ADC1 Sequence 2

Figure 6: ADC 1 vector number = 62

Step 3: Waveform Processing:

Waveform Task:

In this lab we must break the functionality from lab 1 into five tasks so that they can executed using the RTOS. The first of these tasks is the waveform task which is our highest priority since we need the trigger search to complete before the ADC rewrites over the buffer with new samples. This also protects shared access to the waveform buffer. This task blocks on a

semaphore that is configured in the RTOS. It searches for a trigger and then copies the triggered waveform into a buffer and then signals the processing task and blocks again. Different from lab 1 we now have a different spectrum mode so the code from lab 1 must be edited to implement this functionality. It works very similarly to the regular square wave, except it uses a different local buffer to copy the samples except the data will be slightly different and this step does not involve a trigger; it simply copies the samples that it gets from the given wave processing setup code. More will be explained in step 5.

```
void WaveformTask(UArg arg1, UArg arg2) //highest priority
{
    IntMasterEnable();
    int i, trig;
    while (true) {
        Semaphore_pend(semaphoreWaveform, BIOS_WAIT_FOREVER);
        if(mode){
            trig = triggerSlope ? RisingTrigger(): FallingTrigger();//either rise or fall
            for (i = 0; i < LCD_HORIZONTAL_MAX - 1; i++){

                // Copies waveform into the local buffer
                sample[i] = gADCBuffer[ADC_BUFFER_WRAP(trig - LCD_HORIZONTAL_MAX / 2 + i)];
            }
            Semaphore_post(semaphoreProcessing);
        }
        else{
            for (i = 0; i < NFFT; i++) { // generate an input waveform
                samplefft[i] = gADCBuffer[ADC_BUFFER_WRAP(i)];

            }//end for
            Semaphore_post(semaphoreProcessing);
        }//end if
    }//end while 1
}//end func
```

Figure 7: Waveform Task

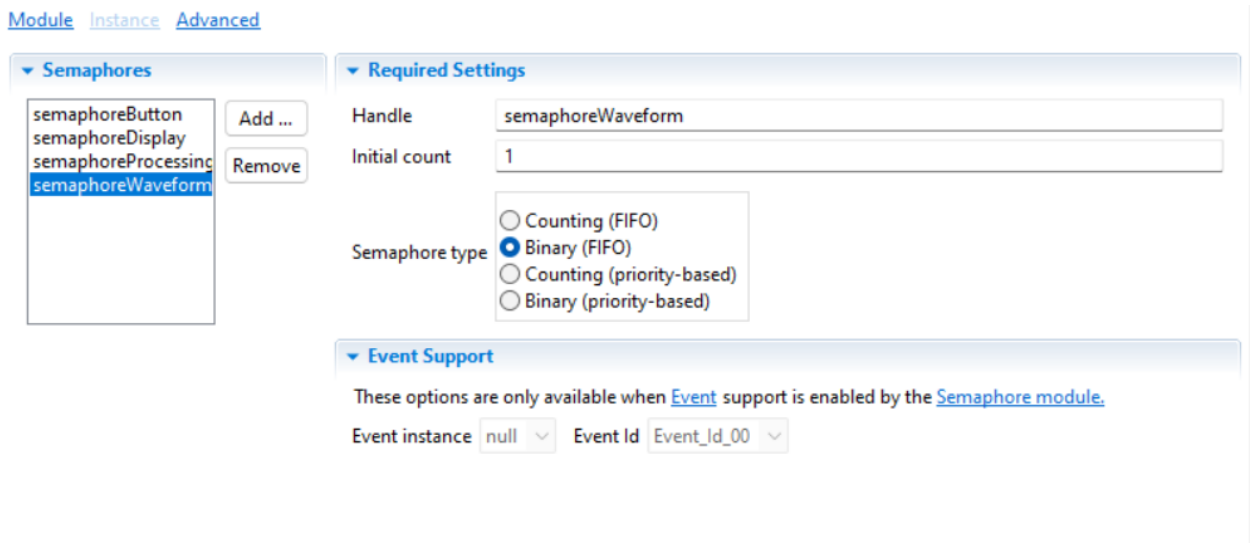

Figure 8: Waveform Semaphore

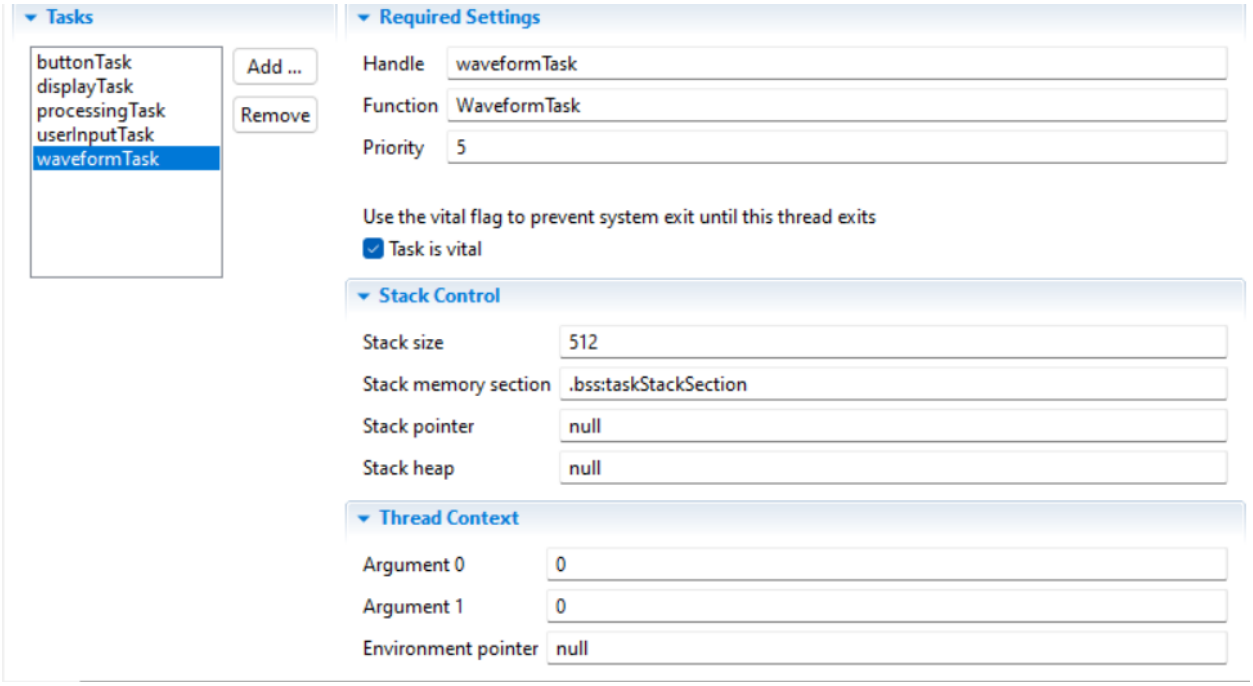

Figure 9: Waveform Task RTOS setup

Processing Task:

The next task I created was the processing task which is the lowest priority. This task scales the buffer to the screen via a global buffer. This task does not draw the waveform it simply formats it and signals to the waveform task to get another waveform. It also signals the Display task to draw the processed data. Much of the code is taken from lab 1 and for the spectrum mode the bounds of the loop and the calculation are slightly different. More will be explained in step 5. The semaphore is the same as the waveform task and the only parts that change in the RTOS setup are the priority number and stack size as I noticed it needed to be increased to 1024 as the preset 512 was not enough.

```
void ProcessingTask(UArg arg1, UArg arg2)
{
    int i;
    cfg = kiss_fft_alloc(NFFT, 0, kiss_fft_cfg_buffer, &buffer_size); // init Kiss FFT
    while (true){
        Semaphore_pend(semaphoreProcessing, BIOS_WAIT_FOREVER);
        if(mode){
            scale = (VIN_RANGE * PIXELS_PER_DIV) / ((1 << ADC_BITS) * fVoltsPerDiv[voltsperDiv]);
        for (i = 0; i < LCD_HORIZONTAL_MAX - 1; i++){
            sampledisp[i] = LCD_VERTICAL_MAX / 2 - (int)roundf(scale * ((int)sample[i] - ADC_OFFSET));
          }//end for

        Semaphore_post(semaphoreDisplay);
        Semaphore_post(semaphoreWaveform);

        }
        else{
         for (i = 0; i < NFFT; i++) {    // generate an input waveform
         in[i].r = samplefft[i]; // real part of waveform
         in[i].i = 0;                 // imaginary part of waveform
         }
         kiss_fft(cfg, in, out); //compute FFT
         // convert first 128 bins of out[] to dB for display
         for (i = 0; i < LCD_HORIZONTAL_MAX - 1; i++){
          out_db[i] = 160 - ((10.0f) * log10f((out[i].r * out[i].r) + (out[i].i * out[i].i)));
         }//end for
         Semaphore_post(semaphoreDisplay);
         Semaphore_post(semaphoreWaveform);
    }
  }//end while 1
```

Figure 10: Processing Task

Display Task:

The display task will be the second lowest priority task. This task will wait for other tasks to signal it to refresh the screen. When signaled it uses the same code from lab 1 to draw the square wave and parts of this code are used to draw the spectrum. More explanation on the spectrum drawing in step 5. Need to be careful with shared data as this accesses the processing task buffer. Setup and semaphore are like the other tasks except the priority number and the stack size also needed to be increased here from 512 to 1024.

```
void DisplayTask(UArg arg1, UArg arg2)
{
        tContext sContext;
        GrContextInit(&sContext, &g_sCrystalfontz128x128); // Initialize the grlib graphics context
        GrContextFontSet(&sContext, &g_sFontFixed6x8);     // Select font
        int i, y, yP;
        char str1[50];
        tRectangle rectFullScreen = {0, 0, GrContextDpyWidthGet(&sContext)-1, GrContextDpyHeightGet(&sContext)-1};
        while(true) {
            Semaphore_pend(semaphoreDisplay, BIOS_WAIT_FOREVER);
            if(mode){
                                GrContextForegroundSet(&sContext, ClrBlack);
                                GrRectFill(&sContext, &rectFullScreen); // fill screen with black

                                //blue grid
                                GrContextForegroundSet(&sContext, ClrBlue);
                                for(i = -3; i < 4; i++) {
                                    GrLineDrawH(&sContext, 0, LCD_HORIZONTAL_MAX - 1, LCD_VERTICAL_MAX/2 + i * PIXELS_PER_DIV);
                                    GrLineDrawV(&sContext, LCD_VERTICAL_MAX/2 + i * PIXELS_PER_DIV, 0, LCD_HORIZONTAL_MAX - 1);
                                }

                                //waveform
                                GrContextForegroundSet(&sContext, ClrYellow);
                                 for(i = 0; i < LCD_HORIZONTAL_MAX - 1; i++) {
                                    y = sampledisp[i];
                                    GrLineDraw(&sContext, i, yP, i + 1, y);
                                    yP = y;
                                }
                                //trigger direction, volts per div, cpu load
                                GrContextForegroundSet(&sContext, ClrWhite); //white text
                                if(triggerSlope){
                                    GrLineDraw(&sContext, 105, 10, 115, 10);
                                    GrLineDraw(&sContext, 115, 10, 115, 0);
                                    GrLineDraw(&sContext, 115, 0, 125, 0);
                                    GrLineDraw(&sContext, 112, 6, 115, 2);
                                    GrLineDraw(&sContext, 115, 2, 118, 6);
                                }else{
                                    GrLineDraw(&sContext, 105, 10, 115, 10);
                                    GrLineDraw(&sContext, 115, 10, 115, 0);
                                    GrLineDraw(&sContext, 115, 0, 125, 0);
                                    GrLineDraw(&sContext, 112, 3, 115, 7);
                                    GrLineDraw(&sContext, 115, 7, 118, 3);
                                }

                                GrContextForegroundSet(&sContext, ClrWhite); //white text
                                GrStringDraw(&sContext, "20 us", -1, 4, 0, false);
                                GrStringDraw(&sContext, gVoltageScaleStr[voltsperDiv], -1, 50, 0, false);

                                //Draw Missing Trigger indicator
                                GrContextForegroundSet(&sContext, ClrRed); //Red text
                                            while (checkTrigger == true){
                                                snprintf(str1, sizeof(str1), "Missing Trigger");
                                                GrStringDraw(&sContext, str1, -1, 20, 64, false);
                                                break;
                                            }

                                GrFlush(&sContext); // flush the frame buffer to the LCD
        }
```

```
            }
            else{
                GrContextForegroundSet(&sContext, ClrBlack);
                GrRectFill(&sContext, &rectFullScreen); // fill screen with black

                //blue grid
                GrContextForegroundSet(&sContext, ClrBlue);
                for(i = -3; i < 4; i++) {
                GrLineDrawH(&sContext, 0, LCD_HORIZONTAL_MAX - 1, LCD_VERTICAL_MAX/2 + i * PIXELS_PER_DIV);
                }
                for (i = 0; i < 7; i++) {
                GrLineDrawV(&sContext, i * PIXELS_PER_DIV, 0, LCD_HORIZONTAL_MAX - 1);
                }

                //waveform
                GrContextForegroundSet(&sContext, ClrYellow);
                for(i = 0; i < LCD_HORIZONTAL_MAX - 1; i++) {
                y = out_db[i];
                GrLineDraw(&sContext, i, yP, i + 1, y);
                yP = y;
                }
                //frequency and decibel
                GrContextForegroundSet(&sContext, ClrWhite); //white text
                GrStringDraw(&sContext, "20 kHz", -1, 4, 0, false);
                GrStringDraw(&sContext, "10 dBV", -1, 50, 0, false);

                GrFlush(&sContext); // flush the frame buffer to the LCD

            }//end else
        }//end while 2
}//end function
```

Figure 11: Display Task

Step 4: Button Scanning

Clock0 Task:

A clock object needs to be configured to post the Button Semaphore. It uses Timer0 to do this. It is configured with a tick period of 5ms to check the buttons at this interval. A corresponding clock task is created as well as the RTOS configuration.

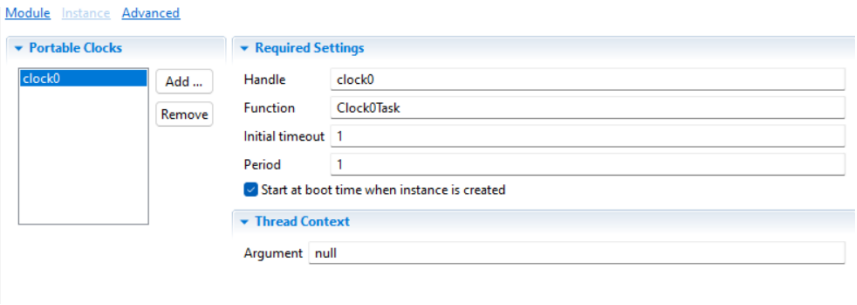

Figure 12: RTOS clock instance

```
void Clock0Task(UArg arg1, UArg arg2)
{
    Semaphore_post(semaphoreButton);
}
```

Figure 13: Clock0 Task

User Input Task:

Most of this code was taken from buttons.c in lab 1. The RTOS now handles the button handling for their specific functions. This task is middle priority. It pends a button mailbox which is configured in the RTOS. When receiving the char it completes the desired command based on the case. When it receives the char of the desired button press it signals the change to the oscilloscope and pends the display task via a semaphore.

```
void UserInputTask(UArg arg1, UArg arg2)
{
    char button;
    while(true){
        if(Mailbox_pend(mailbox0, &button, BIOS_WAIT_FOREVER)){
        switch(button){
         case 'a':
             voltsperDiv = voltsperDiv >= 4 ? 4 : voltsperDiv++; //Extra credit lab 1
             break;
         case 'b':
             voltsperDiv = voltsperDiv <= 0 ? 0 : voltsperDiv--; //Extra credit lab 1
             break;
         case 'c':
             mode = !mode; //spectrum mode
             break;
         case'd':
             triggerSlope = !triggerSlope; //Trigger slope change
             break;
      }
         Semaphore_post(semaphoreDisplay);
    }
  }//end while
}//end function
```

Figure 14: User Input Task

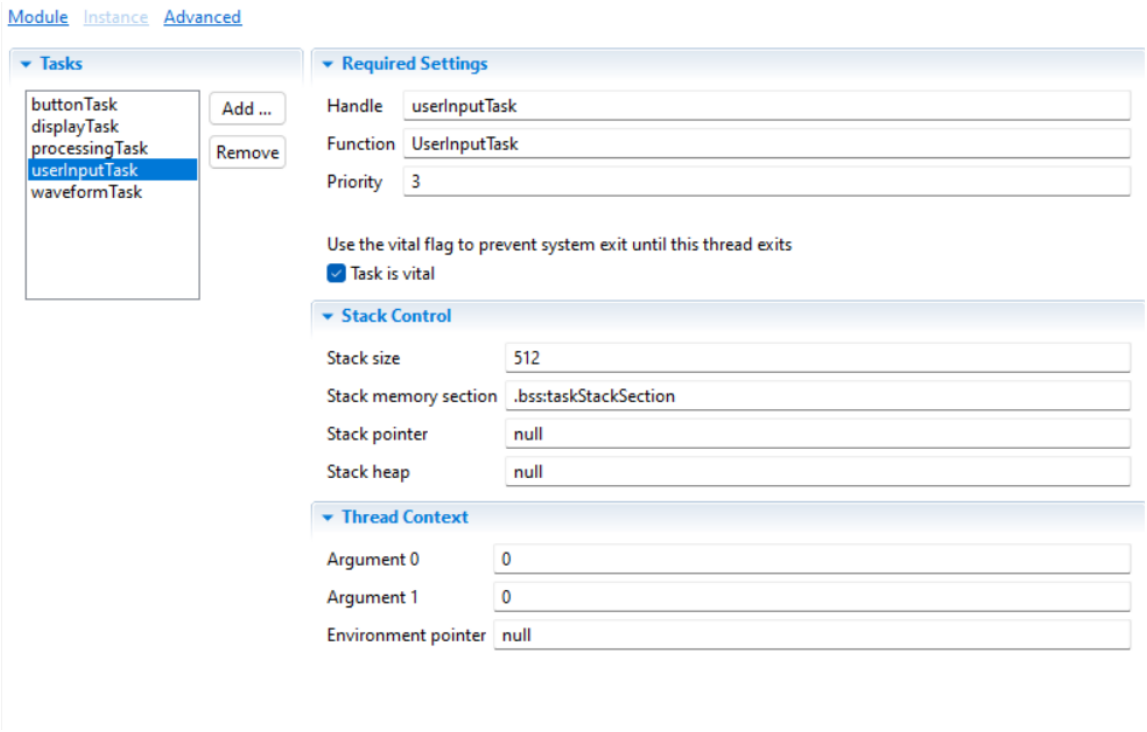

Figure 15: RTOS setup of User Input Task

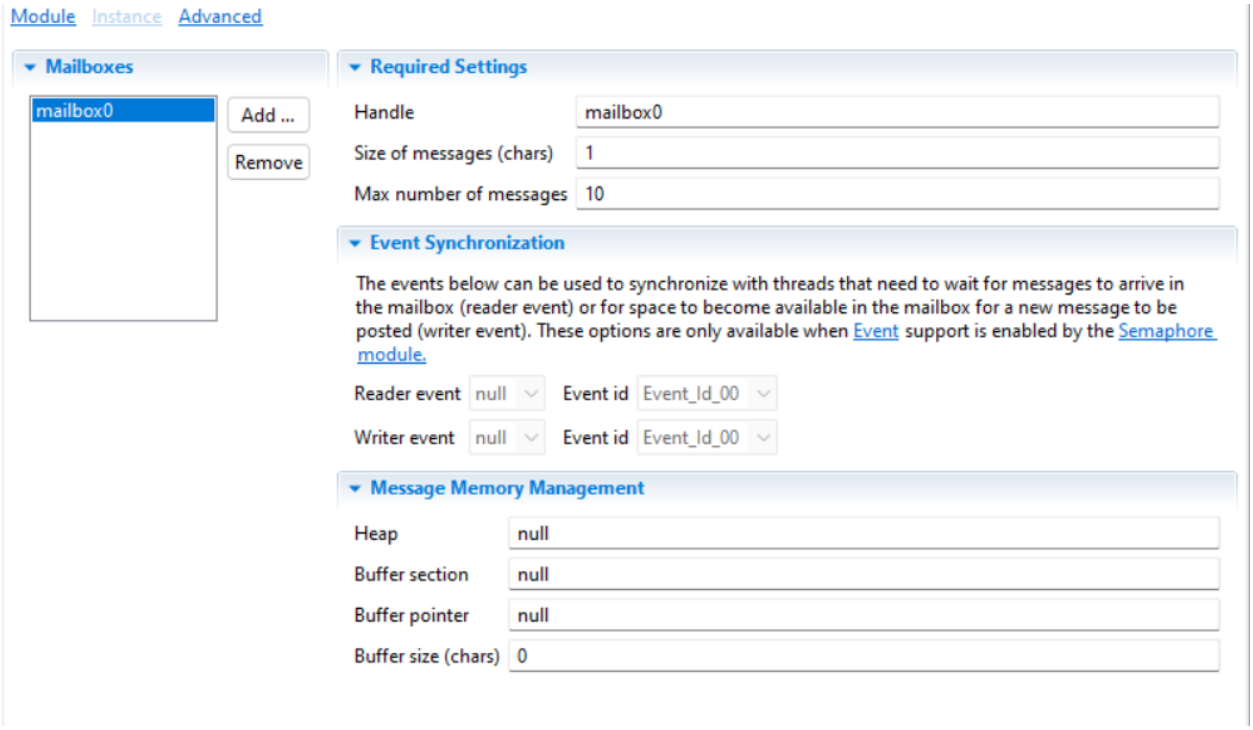

Figure 16: Mailbox Configuration

Button Task:

Much of the code in this task was also taken from buttons.c in lab 1. It configures the buttons and handles the button triggering. It is the second highest priority task to ensure that button presses are not missed. The clock instance signals the Button task via a semaphore. It posts to the mailbox on which the button is pushed. The two onboard buttons have their same function from lab 1: they control the voltage scale for the square wave. Button S2 on the booster pack switches the rising and falling trigger for the square wave and button S1 toggles between spectrum and the normal oscilloscope mode.

```
void ButtonTask(UArg arg1, UArg arg2)
{
    char button;
    while(1){
        Semaphore_pend(semaphoreButton, BIOS_WAIT_FOREVER);
        uint32_t gpio_buttons =
                    (~GPIOPinRead(GPIO_PORTJ_BASE, 0xff) & (GPIO_PIN_1 | GPIO_PIN_0)) | //EK-TM4C1294XL
                    (~GPIOPinRead(GPIO_PORTH_BASE, 0xff) & (GPIO_PIN_1)) << 1 |         //BoosterPack bu
                    (~GPIOPinRead(GPIO_PORTK_BASE, 0xff) & (GPIO_PIN_6)) >> 3 |         //BoosterPack bu
                    ~GPIOPinRead(GPIO_PORTD_BASE, 0xff) & (GPIO_PIN_4);                 //BoosterPack bu


            uint32_t old_buttons = gButtons;    // save previous button state
            ButtonDebounce(gpio_buttons);       // Run the button debouncer. The result is in gButtons.
            ButtonReadJoystick();               // Convert joystick state to button presses. The result
            uint32_t presses = ~old_buttons & gButtons;   // detect button presses (transitions from not
            presses |= ButtonAutoRepeat();      // autorepeat presses if a button is held long enough

            if (presses & 1) { // EK-TM4C1294XL button 1 pressed
                button = 'a';
                Mailbox_post(mailbox0, &button, BIOS_NO_WAIT);
            }
            if (presses & 2) { // EK-TM4C1294XL button 2 pressed
                button = 'b';
                Mailbox_post(mailbox0, &button, BIOS_NO_WAIT);
                }
            if (presses & 4) { //S1
                button = 'c';
                Mailbox_post(mailbox0, &button, BIOS_NO_WAIT);
            }
            if (presses & 8) { //S2
                button = 'd';
                Mailbox_post(mailbox0, &button, BIOS_NO_WAIT);
            }
        }//end while 1
    }//end function
```

Figure 16: Button task

Step 5: Spectrum (FFT) Mode

This step explains how the spectrum mode is implemented into the code and how each task contributes to it. There were many given pieces to this code pictured in figures 17-19.

Waveform Task:

The waveform task captures the newest 1024 samples with no trigger search. The newest samples are the 1024 samples taken prior to the current gADCBufferIndex. Using these will avoid shared data issues.

Processing Task:

The processing task uses 1024-point FFT of the captured waveform in a single precision float number. It then converts the lowest 128 frequency bins of the complex spectrum to magnitude in dB and saves them into the processed waveform buffer scaled at 1 dB/pixel for display.

Display Task:

The display task prints the frequency and dB scales. The grid is also shifted slightly from the given code to better fit in the screen.

```
#include <math.h>
#include "kiss_fft.h"
#include "_kiss_fft_guts.h"

#define PI 3.14159265358979f
#define NFFT 1024         // FFT length
#define KISS_FFT_CFG_SIZE (sizeof(struct kiss_fft_state)+sizeof(kiss_fft_cpx)*(NFFT-1))
static char kiss_fft_cfg_buffer[KISS_FFT_CFG_SIZE]; // Kiss FFT config memory
size_t buffer_size = KISS_FFT_CFG_SIZE;
kiss_fft_cfg cfg;                          // Kiss FFT config
static kiss_fft_cpx in[NFFT], out[NFFT]; // complex waveform and spectrum buffers
int i;

cfg = kiss_fft_alloc(NFFT, 0, kiss_fft_cfg_buffer, &buffer_size); // init Kiss FFT
```

Figure 17: Defines, variables and cfg function

```
for (i = 0; i < NFFT; i++) {    // generate an input waveform
  in[i].r = sinf(20*PI*i/NFFT); // real part of waveform
  in[i].i = 0;                  // imaginary part of waveform
}

kiss_fft(cfg, in, out);         // compute FFT

// convert first 128 bins of out[] to dB for display
```

Figure 18: Waveform generation

```
out_db[i] = 10 * log10f(out[i].r * out[i].r + out[i].i * out[i].i);
```

Figure 19: Given out_dB function

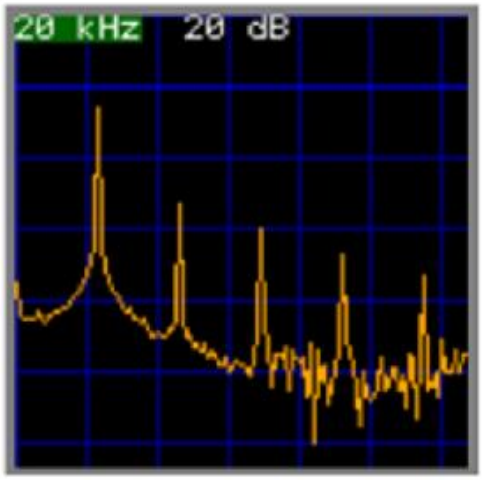

Figure 20: Sample Spectrum

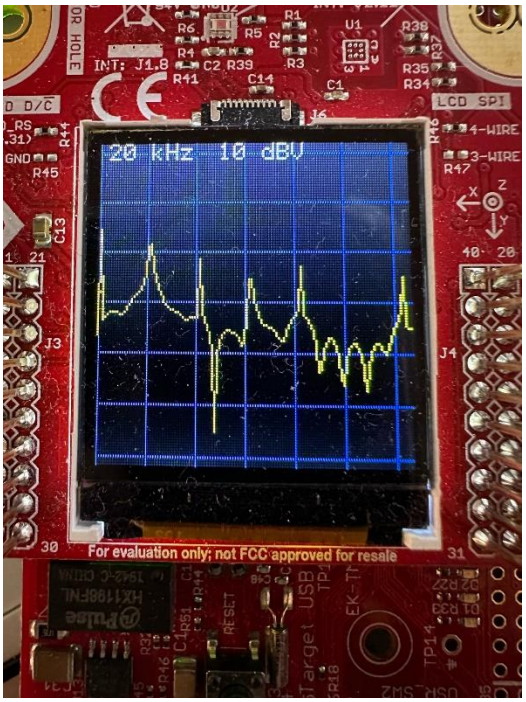

Figure 21: Spectrum Mode Implementation

Conclusion:

I found this lab to be straight forward considering that a large part of the code was either given or taken from lab 1. The one problem that I ran into was trying to properly get the spectrum to display but this problem was resolved relatively quickly by manipulating the way buffer was copying the samples. The important things that I learned from this lab were how to properly implement the RTOS, how to implement functionality into tasks, how to use semaphores within these tasks, and how to manage a system with many different levels of priority. I found it valuable that this lab showed how an RTOS is implemented, and I found that I liked coding using the RTOS more than the way in which I did in lab 1. I do not believe this lab needs any improvements as it demonstrated topics from class without being too difficult.