ECE 3849 D2018
Real-Time Embedded Systems
Lab 2: RTOS & Spectrum Analyzer

by

Chaiwat Ekkaewnumchai

Tianyi Xu

Box#: 366

Date:

17 April 2018

Table of Contents

Introduction

The Lab 2 has two main parts. The first part is to port Lab 1 application, a 1 Msps digital oscilloscope,to an RTOS (TI-RTOS) using Task, Hwi, Clock, Semaphore, Mailbox, etc. We divided the main() code from Lab1 into lots of different Tasks, Hwi and Clock threads to complete this part. The second part is to add an FFT mode that can turn the system into a simple spectrum analyzer, so that we can display the signal in the frequency domain. Additionally, we can switch between the FFT and oscilloscope modes using the specific button.

Discussion and Results

- System requirements and real-time goals. Assign thread types and priorities based on the top-level design.

The system requirements consist of the correctness of measurement and ADC, the correctness of offset when interpreting raw ADC values, the continuity of the graph from multiple ADC values (e.g. the given generated example), the correctness of multiple user input to change volt scale and trigger, and the smooth of real-time graph change. The real-time goals are to be able to do five different tasks without error and share data bug. Without extra credit part, we used one clock, one Hwi, five tasks, six semaphores, and one mailbox. Clock object is used to schedule periodic button scanning, and the Clock function can signal the Button Task using a Semaphore. TI-RTOS Hwi object is ADC_ISR. The five tasks from highest priority to lowest priority are Button Task, Waveform Task, User Input Task, Display Task and Processing Task. Four of six semaphores are used to singal and communicate between tasks, and rest two semaphores are used to protect the shared data. Mailbox is used to post and pend button ID between Button Task and User Input Task.

- Implementation of each major lab component.

The first step we did was creating Hwi object inside the rtos GUI, and this Hwi object should call ADC_ISR function. Then we also created a clock object to schedule periodic button scanning. We initialized the Clock Module to a 5 ms clock tick period and used semaphore to signal Button Task inside the clock_tick_func function. We can use semaphore_post to signal the next task, and write semaphore_pend in the beginning of the task being signaled with the same semaphore handle. The code for this part is shown in Figure 1.

```
void Clock_tick_func(){
    Semaphore_post(button_sem);
}
```

Figure1. Clock Function Code

Then, we have to break main() code in lab1 to five tasks. We also gave these five tasks different priority, and used semaphore to connect and signal them. We break them into many tasks instead of just two, three tasks because some codes were very heavy, breaking them into small pieces could avoid missing deadlines. The five tasks from highest priority to lowest priority are Button Task, Waveform Task, User Input Task, Display Task and Processing Task. We set their priority inside the rtos GUI.

The Button Task was signaled by Clock function first, so we wrote semaphore_pend() in the beginning of this task. In the end of this function, we used Mailbox_post to post button IDs and signal User Input Task.

```
// highest priority -> 5
// may need to be fixed for priority inversion
void ButtonTask_func(UArg arg1, UArg arg2)
{
    IntMasterEnable();


    while (true) {
        // wait for the interrupt to occur
        Semaphore_pend(button_time_sem, BIOS_WAIT_FOREVER);

        // wait for the signal
        Semaphore_pend(button_sem, BIOS_WAIT_FOREVER);

        // measuring latency
        t = TIMER0_PERIOD - TIMER0_TAR_R;
        if (t > button_task_latency) button_task_latency = t; // measure latency

        // after getting signaled
        uint32_t presses = button_func();
        char msg;

        if(presses & VOLT_DEC) { // left side
            msg = '6';
            Mailbox_post(button_mailbox, &msg, BIOS_WAIT_FOREVER);
        }
        if(presses & VOLT_INC) {  // right side
            msg = '5';
            Mailbox_post(button_mailbox, &msg, BIOS_WAIT_FOREVER);
        }
        if(presses & TRIG_CHANGE) { // user switch 1
            msg = '2';
            Mailbox_post(button_mailbox, &msg, BIOS_WAIT_FOREVER);
        }
        if(presses & MODE_CHANGE) { // user switch 2
            msg = '3';
            Mailbox_post(button_mailbox, &msg, BIOS_WAIT_FOREVER);
        }

        // measuring response time and missing deadline
        if (Semaphore_getCount(button_time_sem)) { // next event occurred
            button_task_missed++;
            t = 2 * TIMER0_PERIOD; // timer overflowed since last event
        }
        else t = TIMER0_PERIOD;
        t -= TIMER0_TAR_R;
        if (t > button_task_response) button_task_response = t; // measure response time

    }
}
```

Figure 2. Button Task Code

We wrote Mailbox_pend() in the beginning of this task in order to response to the last task, Button Task. Mailbox_pend() could get a button ID from mailbox, then this task could modify the oscilloscope settings and signal the Display Task using a Semaphore in the end.

```
// medium priority -> 3
void UserInputTask_func(UArg arg1, UArg arg2)
{

    while (true) {
        // read the value from mailbox
        char msg;
        Mailbox_pend(button_mailbox, &msg, BIOS_WAIT_FOREVER);

        // modified shared data for display
        switch(msg){
           case '6': // decrease value of voltage
               currentVoltSel = currentVoltSel > 0? currentVoltSel - 1 : 0;
               break;
           case '5': // increase value of voltage
               currentVoltSel = currentVoltSel < 3? currentVoltSel + 1 : 3;
               break;
           case '2':
               risingTrig = !risingTrig;
               break;
           case '3':
               FFTMode = !FFTMode;
               break;
       }

       // signal display function
       Semaphore_post(display_sem);


    }
}
```

Figure 3. User Input Task Code

The Display Task was used to draw one frame including the settings and the waveform to the LCD. It was signaled from User Input Task, and it didn't signal any other tasks. In addition, because waveform buffer was shared data, we used another semaphore to protect it. In order to do it, semaphore_pend was put before shared data, and semaphore post was put after shared data. The code for this task is shown in figure below.

```c
// low priority -> 2
void DisplayTask_func(UArg arg1, UArg arg2){
    int buffer[LCD_HORIZONTAL_MAX], i;
    bool curr_trig;
    int curr_volt_stage;
    tRectangle rectFullScreen = {0, 0, GrContextDpyWidthGet(&sContext)-1, GrContextDpyHeightGet(&sContext)-1};
    bool fft;
    int x;

    while (true) {
        // wait for signal
        Semaphore_pend(display_sem, BIOS_WAIT_FOREVER);

        fft = FFTMode;

        // read the data; if the semaphore is held -> ignore and use previous data
        if(Semaphore_pend(display_data_sem, BIOS_NO_WAIT)){
            curr_trig = risingTrig;
            curr_volt_stage = currentVoltSel;
            for(i = 0; i < LCD_HORIZONTAL_MAX; i++)
                buffer[i] = displayBuffer[i];
            Semaphore_post(display_data_sem);
        }
        // start drawing
        GrContextForegroundSet(&sContext, ClrBlack);
        GrRectFill(&sContext, &rectFullScreen); // fill screen with black

        GrContextForegroundSet(&sContext, ClrBlue); // blue
        if(fft){
            // draw vertical line
            for (x = 0; x <= 124; x = x + 20)
                GrLineDraw(&sContext, x, 0, x, LCD_VERTICAL_MAX);
            for (x = 0; x <= 124; x = x + 20)
                GrLineDraw(&sContext, 0, x, LCD_HORIZONTAL_MAX, x);
        } else {
            // draw vertical line
            for (x = 4; x <= 124; x = x + 20)
                GrLineDraw(&sContext, x, 0, x, LCD_VERTICAL_MAX);
            GrLineDraw(&sContext, 63, 0, 63, LCD_VERTICAL_MAX);
            for (x = 4; x <= 124; x = x + 20)
                GrLineDraw(&sContext, 0, x, LCD_HORIZONTAL_MAX, x);
            GrLineDraw(&sContext, 0, 63, LCD_HORIZONTAL_MAX, 63);
        }

        // draw waveform
        GrContextForegroundSet(&sContext, ClrYellow); // yellow
        // change buffer to position on LCD
        for(i = 1; i < LCD_HORIZONTAL_MAX; i++)
            GrLineDraw(&sContext, i - 1, buffer[i - 1], i, buffer[i]);

        GrContextForegroundSet(&sContext, ClrWhite); // white

        // draw string
        if(fft){
            GrStringDraw(&sContext, "20 KHz", /*length*/ -1, /*x*/ 0, /*y*/ 0, /*opaque*/ false);
            GrStringDraw(&sContext, "20 dB", /*length*/ -1, /*x*/ 40, /*y*/ 0, /*opaque*/ false);

        } else {
            // printf context on screen
            GrStringDraw(&sContext, "20us", /*length*/ -1, /*x*/ 0, /*y*/ 0, /*opaque*/ false);
            GrStringDraw(&sContext, gVoltageScaleStr[curr_volt_stage], /*length*/ -1, /*x*/ 40, /*y*/ 0, /*opaque*/ false);
            if(curr_trig){
                GrLineDraw(&sContext, 85, 7, 90, 7);
                GrLineDraw(&sContext, 90, 2, 95, 2);
                GrLineDraw(&sContext, 90, 2, 90, 7);
            }
            else{
                GrLineDraw(&sContext, 90, 7, 95, 7);
                GrLineDraw(&sContext, 85, 2, 90, 2);
                GrLineDraw(&sContext, 90, 2, 90, 7);
            }
        }

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

Figure 4. Display Task Code

The Waveform Task was used to search for trigger in the ADC buffer and copy the triggered waveform into the waveform buffer. It should be signaled by Processing Task, and signaled the Processing Task after it's done. We also used semaphore_pend and post to protect shared data in this task.

```c
// high priority -> 4
void WaveformTask_func(UArg arg1, UArg arg2)
{

    bool curr_trig;
    int32_t i, search_count;
    int32_t trig_idx;   // trigger index
    volatile int32_t curr_idx;   // current index of ADC when loaded

    while (true) {
        // wait for the signal
        Semaphore_pend(trigger_sem, BIOS_WAIT_FOREVER);

        start = SysTickValueGet(); // read SysTick timer value

        if(FFTMode) { // ====== FFT MODE ====== //
            // copy new data to FFTBuffer
            Semaphore_pend(fft_data_sem, BIOS_WAIT_FOREVER);
            for(search_count = 0, i = gADCBufferIndex; search_count < NFFT; search_count++, i = ADC_BUFFER_WRAP(i - 1))
                FFTBuffer[search_count] = gADCBuffer[i];
            Semaphore_post(fft_data_sem);

        } else { // ====== NORMAL MODE ====== //
            // Create buffer to create oscilloscope
            curr_idx = gADCBufferIndex;
            trig_idx = ADC_BUFFER_WRAP(curr_idx - HALF_SCREEN);
            curr_trig = risingTrig;
            search_count = 0;
            if(gADCBuffer[trig_idx] != ADC_OFFSET){
                if(curr_trig)
                    while(!((gADCBuffer[ADC_BUFFER_WRAP(trig_idx + 1)] >= ADC_OFFSET) && (gADCBuffer[trig_idx] < ADC_OFFSET)) && (++search_count < SEARCH_REGION))
                        trig_idx = ADC_BUFFER_WRAP(trig_idx - 1);
                else
                    while(!((gADCBuffer[ADC_BUFFER_WRAP(trig_idx + 1)] <= ADC_OFFSET) && (gADCBuffer[trig_idx] > ADC_OFFSET)) && (++search_count < SEARCH_REGION))
                        trig_idx = ADC_BUFFER_WRAP(trig_idx - 1);

                if(search_count >= SEARCH_REGION)
                    trig_idx = ADC_BUFFER_WRAP(curr_idx - HALF_SCREEN);
            }

            // copy the the buffer to global variable
            Semaphore_pend(waveform_data_sem, BIOS_WAIT_FOREVER);
            for(i = 0; i < HALF_SCREEN * 2; i++)
                waveformBuffer[i] = gADCBuffer[ADC_BUFFER_WRAP(trig_idx + i - HALF_SCREEN + 1)];
            Semaphore_post(waveform_data_sem);

        }

        finish = SysTickValueGet(); // read SysTick timer value
        duration = (start - finish) & 0xffffff;
        if(max_duration < duration) max_duration = duration;

        // signal process to process data
        Semaphore_post(process_sem);
    }
}
```

Figure 5. Waveform Task Code

The Processing Task was used to scale the captured waveform for display, and then store the processed waveform in another global buffer. In addition, it was used to calculate and display FFT mode in the next part. It should be signaled from Waveform Task, and Waveform Task after this task finished. When updating data into a global buffer, we also used semaphore to protect the shared data.

```
// lowest priority -> 1
void ProcessingTask_func(UArg arg1, UArg arg2)
{
    TimerEnable(TIMER0_BASE, TIMER_A);
    int curr_volt_stage;
    float fScale;
    int i, temp;
    // FFT
    static char kiss_fft_cfg_buffer[KISS_FFT_CFG_SIZE];
    size_t buffer_size = KISS_FFT_CFG_SIZE;
    kiss_fft_cfg cfg = kiss_fft_alloc(NFFT, 0, kiss_fft_cfg_buffer, &buffer_size);
    static kiss_fft_cpx in[NFFT], out[NFFT];
    // buffer
    uint16_t inputBuffer[LCD_HORIZONTAL_MAX];
    int outputBuffer[LCD_HORIZONTAL_MAX], minOutput;
    while (true) {
        // wait for signal
        Semaphore_pend(process_sem, BIOS_WAIT_FOREVER);

        // read shared data
        curr_volt_stage = currentVoltSel;

        fScale = (VIN_RANGE * PIXELS_PER_DIV) / ((1 << ADC_BITS) * fVoltsPerDiv[curr_volt_stage]);

        if(FFTMode){ // ====== FFT Mode ====== //
            // copy the waveform data to local variable if available
            if(Semaphore_pend(fft_data_sem, BIOS_NO_WAIT)){
                for(i = 0; i < NFFT; i++){
                    in[i].r = (float)(FFTBuffer[i] - ADC_OFFSET) * VIN_RANGE / (1 << ADC_BITS);
                    //in[i].r = sinf(20 * PI * i / NFFT);
                    in[i].i = 0;
                }
                Semaphore_post(fft_data_sem);
            }
            // compute FFT
            kiss_fft(cfg, in, out);

            // use only 128 lowest bin
            minOutput = 10000;
            for(i = 0; i < LCD_HORIZONTAL_MAX; i++){
                // 20 * log10(abs) = 20 * 1/2 * log10(abs^2)
                outputBuffer[i] = (int)(10.0f * log10f(out[i].r * out[i].r + out[i].i * out[i].i));
                minOutput = minOutput <= outputBuffer[i] ? minOutput : outputBuffer[i];
            }
            for(i = 0; i < LCD_HORIZONTAL_MAX; i++)
                outputBuffer[i] = /*LCD_VERTICAL_MAX*/124 - (outputBuffer[i] - minOutput);

        } else { // ====== Normal Mode ====== //
            // copy the waveform data to local variable if available
            if(Semaphore_pend(waveform_data_sem, BIOS_NO_WAIT)){
                for(i = 0; i < LCD_HORIZONTAL_MAX; i++)
                    inputBuffer[i] = waveformBuffer[i];
                Semaphore_post(waveform_data_sem);
            }
            // update data to local buffer
            for(i = 0; i < LCD_HORIZONTAL_MAX; i++){
                temp = (int)roundf(fScale * ((int)inputBuffer[i] - ADC_OFFSET));
                if(temp > LCD_VERTICAL_MAX / 2)
                    outputBuffer[i] = -1;
                else if(temp < - LCD_VERTICAL_MAX / 2)
                    outputBuffer[i] = LCD_VERTICAL_MAX;
                else
                    outputBuffer[i] = LCD_VERTICAL_MAX / 2 - temp;
            }
        }
        // update data to global buffer
        Semaphore_pend(display_data_sem, BIOS_WAIT_FOREVER);
        for(i = 0; i < LCD_HORIZONTAL_MAX; i++)
            displayBuffer[i] = outputBuffer[i];
        Semaphore_post(display_data_sem);

        // signal Display and then Waveform
        Semaphore_post(display_sem);
        Semaphore_post(trigger_sem);
```

Figure 6. Processing Task Code

The second part of this lab2 is implementing FFT mode. We first downloaded a Kiss_FFT package online, so that we could just call kiss_fft() function to convert the buffer in the time domain into a buffer in the frequency domain. The only two tasks that

we need modify are Processing Task and Display Task. In the Display Task, we just added a if statement to display different text and buffer based on different mode. In the Processing Task, we also added a if statement to switch between FFT mode and normal mode. ADC_ISR stored all the updated data into FFTBuffer. Then we need convert that buffer to a voltage, which is between 0 to 3.3V. The results were saved into in[] real buffer, in[].r. In addition, we also set in buffer imaginal part to 0. After that, we called function kiss_fft() and passed in buffer into that function, and get a out buffer, which contained frequency domain datas. Next, we converted the output results to dB using formula output = 20log(out.r^2+out.i^2)^0.5. The buffer outputBuffer saved the data in dB. Finally, we found the smallest data from outputBuffer, and used that smallest data as an offset to properly display other datas on LCD screen. The code for this part is shown below.

```
if(FFTMode){ // ====== FFT Mode ====== //
    // copy the waveform data to local variable if available
    if(Semaphore_pend(fft_data_sem, BIOS_NO_WAIT)){
        for(i = 0; i < NFFT; i++){
            in[i].r = (float)(FFTBuffer[i] - ADC_OFFSET) * VIN_RANGE / (1 << ADC_BITS);
            //in[i].r = sinf(20 * PI * i / NFFT);
            in[i].i = 0;
        }
        Semaphore_post(fft_data_sem);
    }
    // compute FFT
    kiss_fft(cfg, in, out);

    // use only 128 lowest bin
    minOutput = 10000;
    for(i = 0; i < LCD_HORIZONTAL_MAX; i++){
        // 20 * log10(abs) = 20 * 1/2 * log10(abs^2)
        outputBuffer[i] = (int)(10.0f * log10f(out[i].r * out[i].r + out[i].i * out[i].i));
        minOutput = minOutput <= outputBuffer[i] ? minOutput : outputBuffer[i];
    }
    for(i = 0; i < LCD_HORIZONTAL_MAX; i++)
        outputBuffer[i] = /*LCD_VERTICAL_MAX*/124 - (outputBuffer[i] - minOutput);
```

Figure 7. FFT Mode Code

For extra credit Button Task part, we used timer count to measure latency and response time. In order to do this, we first created another Hwi object, ISR_event0, then we read the timer count to determine the time since the last interrupt, and recorded the maximum number, which is the maximum latency. We determine if a deadline was missed by checking the count of the Semaphore used to signal the Button Task.

For trigger search, we added another module TimeStamp into rtos GUI first. Then we used code t = TIMER0_PERIOD - TIMER0_TAR_R to measure the latency, and the code below to measure the response time.

```
// measuring response time and missing deadline
if (Semaphore_getCount(button_time_sem)) { // next event occurred
    button_task_missed++;
    t = 2 * TIMER0_PERIOD; // timer overflowed since last event
}
else t = TIMER0_PERIOD;
t -= TIMER0_TAR_R;
if (t > button_task_response) button_task_response = t; // measure response time
```

Figure 8. Extra Credit Waveform Task Response Time Code

The results for extra credit part are shown below.

Task	Latency	Response Time	Relative Deadline
Button Task	39us when pressing buttons 19us without pressing buttons	91us when pressing buttons 39us without pressing buttons	5ms
Trigger search and waveform copy in Waveform Task	N/A	1.12ms	1ms

Figure 9. Extra Credit Measurement Results

- Picture of lab results.

The picture below shows the lab results. The breadboard circuit on the left can generate a sinewave. The LCD screen on the right is in FFT mode now, which displays the sinewave signal from breadboard in the frequency domain.

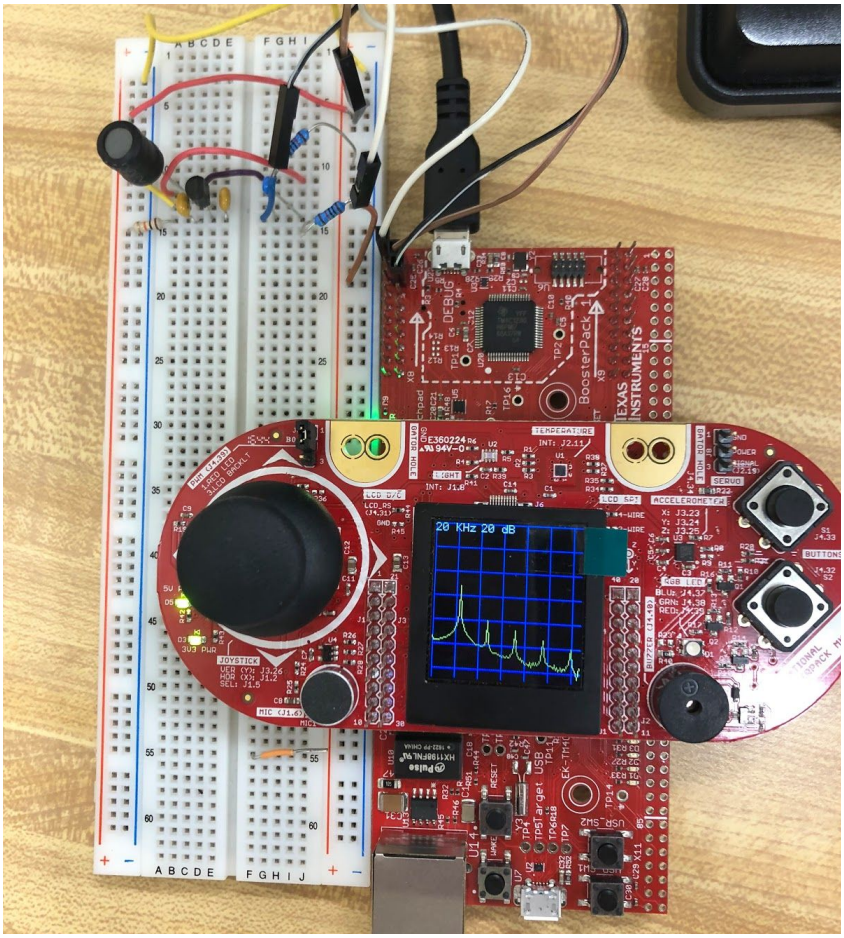

Figure 7. Picture of Lab Result

- Serious difficulties we encountered.

The first difficulty we encountered was that we didn't get correct out buffer from kiss_fft() function in the FFT mode part. Some datas in out buffer were really big. Then, we noticed that we couldn't pass global buffer FFTBuffer directly into function as in, instead, we need convert ADC datas to a voltage between 0 to 3.3V. Therefore, we added a line of code to scale it.

Another difficulty was in extra credit part, we didn't know how to get relative deadline at first. We thought we had to measure it at first, then we noticed that we could calculate this value. For example, button task runs every 5ms, so relative deadline of Button Task is just 5ms. Moreover, there are 1024 samples in trigger search part, since each data needs 1us, the relative deadline is the period of getting 1024 samples, which is 1ms.

Conclusions

In summary, our team successfully developed a 1 Msps digital oscilloscope using RTOS (TI-RTOS). We learned how to convert ISR to Hwi and how to break main() code to multiple tasks using rtos GUI. We also learned how to use Task, Hwi, Clock, Semaphore and Mailbox to communicate between tasks. Additionally, we learned how to use semaphore to protect the critical section and shared data in the task. Finally, in the extra credit part, we improved debug skills. We learned how to measure the real-time latency, response time and the relative deadline for each of these tasks.