ECE 3849 D2022
Real-Time Embedded Systems
Lab 2: RTOS, Spectrum Analyzer

Adam Grabowski, Michael Rideout
April 14, 2022

Introduction

In this lab, we ported the Msps digital oscilloscope from Lab 1 to TI-RTOS. We also implemented an FFT mode that turns the system into a spectrum analyzer. First, we configured the ADC ISR as a TI-RTOS Hwi object using the M3 specific Hwi module. Then, we implemented the waveform task which copies the triggered waveform into the waveform buffer and signals the processing task. The processing task which scales the captured waveform for display and stores the processed waveform in a global buffer. Next, we implemented the display task which draws one frame including the settings and the waveform to the LCD. We also implemented button scanning which processes user commands from the button mailbox and controls the oscilloscope settings. Lastly, using the Kiss FFT package to implement a spectrum analyzer which measures the magnitude of the input signal versus its frequency.

Discussion and Results

This section describes our implementation of each major lab component in the signoff. The following figure outlines the recommended structure of the digital oscilloscope from Lab 1 ported to TI-RTOS.

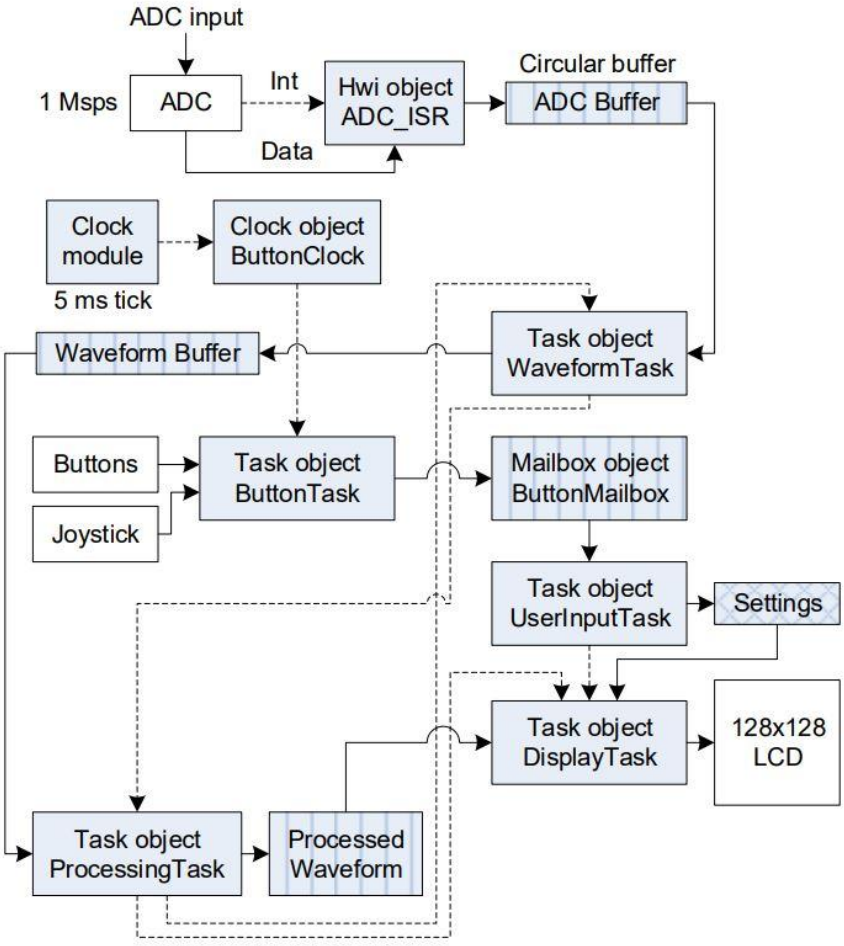

Figure 1: Recommended Implementation

As shown in the figure above, there are five different functions for each task: waveform, button, user input, display, and processing. In order to port the Lab 1 implementation to TI-RTOS, the ADC ISR was converted to a Hwi object while the processes listed earlier were made into tasks. All semaphores utilized in this lab to protect critical sections or signal tasks were configured as binary semaphores.

ADC ISR to Hwi Object

Before the ADC ISR was configured as a Hwi object in TI-RTOS, it ran periodically every 1 µs using a timer. By creating a Hwi object attached to the ADC ISR function, it causes the ISR to execute every 1 µs without interference from critical sections in the operating system. Using the TM4C1294NCPDT datasheet, we found that the vector number of ADC1 has a value of 62 which was used as the interrupt number setting of the Hwi object.

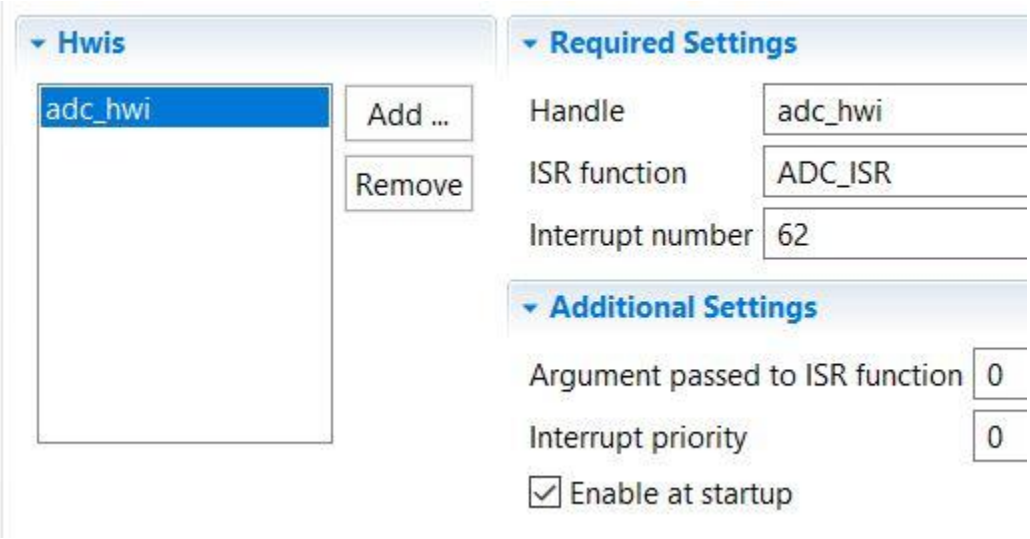

Figure 2: ADC ISR Hwi Instance

Waveform, Processing, and Display Tasks

To create our tasks, we used the rtos.cfq manager and created tasks in the instances window. For each task we were able to control the priority value, and what function the body of the task would be written in.

Waveform Task

The waveform task is the highest priority because it has to run as often as the ADC ISR function. In order to create the waveform task, we implemented a function in sampling.c, called waveformTask_func, and used rtos.cfg to link the instance to this function. The body of the function used a semaphore to wait on other tasks, and another semaphore to protect critical sections. The functionality of this code is to run a trigger search if the board is in standard mode, and to take 1024 samples from ADC for the FFT otherwise. The trigger search function searches for the trigger in the ADC buffer and copies the triggered waveform into a waveform buffer. As shown in the image below, the sem_cs semaphore is used to protect the critical section and the processing task is unblocked.

```
// TI-RTOS waveform task function
void waveformTask_func(UArg arg1, UArg arg2)
{
    IntMasterEnable(); // enable interrupts

    while(true){
        Semaphore_pend(semWaveform, BIOS_WAIT_FOREVER); // from processing

        trigger_value = zeroCrossPoint(); // Dynamically finds the ADC_OFFSET
        if (spectrumMode){
            int i;
            int buffer_ind = gADCBufferIndex;

            Semaphore_pend(sem_cs, BIOS_WAIT_FOREVER); // protect critical section
            for (i = 0; i < NFFT; i++){
                fft_samples[i] = gADCBuffer[ADC_BUFFER_WRAP(buffer_ind - NFFT + i)];
            }
            Semaphore_post(sem_cs);

        } else {
            Semaphore_pend(sem_cs, BIOS_WAIT_FOREVER); // protect critical section

            triggerSearch(); // searches for trigger

            Semaphore_post(sem_cs);
        }

        Semaphore_post(semProcessing); // to processing
    }
}
```

Figure 3: Waveform Task Function Implementation

Processing Task

Processing occurs after the waveform is retrieved from the ADC buffer and is the lowest priority task. The processing task scales the captured waveform for display and stores the processed waveform in a global buffer called processedWaveform. We followed the same methods using rtos.cfg to implement the processing task. The function we linked it to is processingTask_func and the priority was set to 1. The purpose of the processing task's code was to receive the input wave that is created by the signalInit function, and process it to be displayed by the display task. The saving of the data was done by creating an array and filling it with the received values. If the board was not in waveform mode the values were just saved. If it is, we take the FFT of the values to create a new set of data and convert it to decibels to be displayed properly.

```
// TI-RTOS processing task function
void processingTask_func(UArg arg1, UArg arg2)
{
    IntMasterEnable(); // enable interrupts

    static char kiss_fft_cfg_buffer[KISS_FFT_CFG_SIZE];                // KISS FFT config memory
    size_t buffer_size = KISS_FFT_CFG_SIZE;                            // KISS FFT buffer size
    kiss_fft_cfg cfg;                                                  // KISS FFT config
    static kiss_fft_cpx in[NFFT], out[NFFT];                           // complex waveform and spectrum buffers
    cfg = kiss_fft_alloc(NFFT, 0, kiss_fft_cfg_buffer, &buffer_size);  // init Kiss FFT
    int i;

    static float w[NFFT]; // window function
    for (i = 0; i < NFFT; i++) {
        // blackman window
        w[i] = (0.42f - 0.5f * cosf(2*PI*i/(NFFT-1)) + 0.08f * cosf(4*PI*i/(NFFT-1)));
    }

    while(true){
        Semaphore_pend(semProcessing, BIOS_WAIT_FOREVER); // from waveform

        if (spectrumMode){
            Semaphore_pend(sem_cs, BIOS_WAIT_FOREVER); // protect critical section

            for (i = 0; i < NFFT; i++) { // generate an input waveform
                in[i].r = ((float)fft_samples[i] - trigger_value) * w[i];   // real part of waveform
                in[i].i = 0;                                                // imaginary part of waveform
            }

            Semaphore_post(sem_cs);

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

            // convert first 128 bins of out[] to dB for display
            Semaphore_pend(sem_cs, BIOS_WAIT_FOREVER); // protect critical section

            for (i = 0; i < ADC_TRIGGER_SIZE - 1; i++) {
                processedWaveform[i] = (int)roundf(128 - 10*log10f(out[i].r*out[i].r +out[i].i*out[i].i));
            }

            Semaphore_post(sem_cs);

    } else {
        // determines fScale
        fScale = (VIN_RANGE/(1 << ADC_BITS))*(PIXELS_PER_DIV/fVoltsPerDiv[stateVperDiv]);
        int i;

        Semaphore_pend(sem_cs, BIOS_WAIT_FOREVER); // protect critical section

        for (i = 0; i < ADC_TRIGGER_SIZE - 1; i++) {
            processedWaveform[i] = ((int)(ADC_TRIGGER_SIZE/2) - (int)roundf(fScale*(int)(trigger_samples[i] - trigger_value)));
        }

        Semaphore_post(sem_cs);
    }

    Semaphore_post(semWaveform);    // to waveform
    Semaphore_post(semDisplay);     // to display
}
```

Figure 4: Processing Task Function Implementation

Display Task

Display task was created the same way in rtos.cfg as the first two tasks, with a priority of 3 and a function name of displayTask_func. The purpose of the display task is to take the processed data from the waveform and processing tasks, and display it. We were able to use much of the display code from lab 1 to accomplish this.

```
// TI-RTOS display task function
void displayTask_func(UArg arg1, UArg arg2)
{
    IntMasterEnable(); // enable interrupts

    char tscale_str[50];   // time string buffer for time scale
    char vscale_str[50];   // time string buffer for voltage scale
    char tslope_str[50];   // time string buffer for trigger edge

    while(true){
        Semaphore_pend(semDisplay, BIOS_WAIT_FOREVER);  // from user input
        Semaphore_pend(sem_cs, BIOS_WAIT_FOREVER);      // protect critical section

        countLoaded = cpuLoadCount();
        cpuLoad = 1.0f - (float)countLoaded/countUnloaded; // compute CPU load

        Semaphore_post(sem_cs); // protect critical section

        // full-screen rectangle
        tRectangle rectFullScreen = {0, 0, GrContextDpyWidthGet(&sContext)-1, GrContextDpyHeightGet(&sContext)-1};

        GrContextForegroundSet(&sContext, ClrBlack);
        GrRectFill(&sContext, &rectFullScreen);         // fill screen with black
        GrContextForegroundSet(&sContext, ClrWhite);    // yellow text

        // draw grid in blue
        GrContextForegroundSet(&sContext, ClrBlue); // yellow text
        int i;
        for (i = 1; i < 128; i+=21){
            GrLineDraw(&sContext, i, 0, i, 128);
            GrLineDraw(&sContext, 0, i, 128, i);
        }

        // draw center grid lines in dark blue
        GrContextForegroundSet(&sContext, ClrDarkBlue); // blue
        if (spectrumMode){
            GrLineDraw(&sContext, 0, 22, 128, 22);
        } else {
            GrLineDraw(&sContext, 64, 0, 64, 128);
            GrLineDraw(&sContext, 0, 64, 128, 64);
        }

    // draw waveform
    GrContextForegroundSet(&sContext, ClrYellow); // yellow text
    int x;
    int y_old;
    Semaphore_pend(sem_cs, BIOS_WAIT_FOREVER); // protect critical section
    for (x = 0; x < LCD_HORIZONTAL_MAX - 1; x++) {
        if (x!=0)
            GrLineDraw(&sContext, x-1, y_old, x, processedWaveform[x]);
        y_old = processedWaveform[x];
    }

    Semaphore_post(sem_cs);

    // time scale, voltage scale, trigger slope and CPU load
    GrContextForegroundSet(&sContext, ClrWhite); // yellow text
    if (spectrumMode){
        snprintf(tscale_str, sizeof(tscale_str), "20kHz");  // convert time scale to string
        snprintf(vscale_str, sizeof(vscale_str), "20dB");   // convert vscale to string
    } else {
        snprintf(tscale_str, sizeof(tscale_str), "20us");                           // convert time scale to string
        snprintf(vscale_str, sizeof(vscale_str), gVoltageScaleStr[stateVperDiv]);   // convert vscale to string
        snprintf(tslope_str, sizeof(tslope_str), gTriggerSlopeStr[risingSlope]);    // convert slope to string

        GrStringDraw(&sContext, tslope_str, /*length*/ -1, /*x*/ LCD_HORIZONTAL_MAX/2 + 20, /*y*/ 5, /*opaque*/ false);
    }

    GrStringDraw(&sContext, tscale_str, /*length*/ -1, /*x*/ 7, /*y*/ 5, /*opaque*/ false);
    GrStringDraw(&sContext, vscale_str, /*length*/ -1, /*x*/ LCD_HORIZONTAL_MAX/2 - 20, /*y*/ 5, /*opaque*/ false);

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

    Semaphore_pend(semDisplay, BIOS_WAIT_FOREVER); // block again
}
```

Figure 5: Display Task Function Implementation

User Commands

The following sections describe the necessary TI-RTOS objects and functions required to implement user commands. Much like the waveform, processing, and display tasks from before, TI-RTOS instances were created for the user input and button tasks.

Clock Object

A clock object instance was added in order to schedule periodic button scanning and was set to a 5 ms click tick period. Using the semButtons semaphore, the clock function unblocks the button task.

```
// signal button task periodically using semaphore
void clock_func(UArg arg1){
    Semaphore_post(semButtons); // to buttons
}
```

Figure 6: Clock Function Implementation

User Input Task

The user input task function checks whether there are any presses in the mailbox by pending it. If any button presses exist in the queue, the associated action is taken. The user can switch the board between modes which was done using button input. The 3 tasks listed above contain conditions for whether spectrumMode is 1 or 0, and this value is what the button controls. So, by pressing the button, the display task switches to spectrum mode, the processing task processes an FFT in decibels, and the waveform task adjusts to compensate for an FFT instead of just running a trigger search. Also, the critical sections in this task function are protected by the sem_cs semaphore.

```
// TI-RTOS user input task function
void userInputTask_func(UArg arg1, UArg arg2)
{
    IntMasterEnable(); // enable interrupts

    char bpresses[10]; // holds fifo button presses

    while(true){
        // read bpresses and change stats
        if (Mailbox_pend(mailbox0, &bpresses, TIMEOUT)) {
            int i;

            Semaphore_pend(sem_cs, BIOS_WAIT_FOREVER); // protect critical section

            for (i = 0; i < 10; i++){
                if (bpresses[i]==('u') && gButtons == 4) { // increment state
                    stateVperDiv = (++stateVperDiv) % 5;
                } else if (bpresses[i]==('t') && gButtons == 2) { // trigger
                    risingSlope = !risingSlope;
                } else if (bpresses[i]==('s') && gButtons == 8) { // spectrum mode
                    spectrumMode = !spectrumMode;
                }
            }

            Semaphore_post(sem_cs);
        }

        Semaphore_post(semDisplay); // to display
    }
}
```

Figure 7: User Input Task Function Implementation

Button Task

The button ISR from Lab 1 was also converted into a task with medium priority. The only change that was made to this task was to implement the mailbox instead of the FIFO data structure. Now, a message is posted to the mailbox which contains a character indicating the button that was pressed.

```
// TI-RTOS button task function
void buttonTask_func(UArg arg1, UArg arg2)
{
    IntMasterEnable(); // enable interrupts

    while(true){
        Semaphore_pend(semButtons, BIOS_WAIT_FOREVER); // from clock

        // read hardware button state
        uint32_t gpio_buttons =
                ~GPIOPinRead(GPIO_PORTJ_BASE, 0xff) & (GPIO_PIN_1 | GPIO_PIN_0) |   // EK-TM4C1294XL buttons in positions 0 and 1
                (~GPIOPinRead(GPIO_PORTH_BASE, 0xff) & (GPIO_PIN_1)) << 1 |         // BoosterPack button 1
                (~GPIOPinRead(GPIO_PORTK_BASE, 0xff) & (GPIO_PIN_6)) >> 3 |         // BoosterPack button 2
                ~GPIOPinRead(GPIO_PORTD_BASE, 0xff) & (GPIO_PIN_4);                 // BoosterPack buttons joystick select button

        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 is in gButtons.
        uint32_t presses = ~old_buttons & gButtons; // detect button presses (transitions from not pressed to pressed)
        presses |= ButtonAutoRepeat();              // autorepeat presses if a button is held long enough
        char button_char;

        if (presses & 2) { // boosterpack button 1 pressed
            // trigger slope change
            button_char = 't';
            Mailbox_post(mailbox0, &button_char, TIMEOUT);
        }

        if (presses & 4) { // boosterpack button 1 pressed
            // increment
            button_char = 'u';
            Mailbox_post(mailbox0, &button_char, TIMEOUT);
        }

        if (presses & 8) { // boosterpack button 1 pressed
            // spectrum mode
            button_char = 's';
            Mailbox_post(mailbox0, &button_char, TIMEOUT);
        }
    }
}
```

Figure 8: Button Task Function Implementation

Spectrum Mode

In spectrum mode, the board displays the received data in the frequency domain. The received data is transformed using an FFT, and the resulting data is shown in decibels. In the processing task, the Kiss FFT package is used to compute the 1024 point FFT of the captured waveform. Then, the lowest 128 frequency bins of the complex spectrum are converted to magnitude in dB. Lastly, it saves these into the processed waveform buffer scaled at 1 dB/pixel for display. This tells us what frequencies are present in the original set of data, and they appear in the form of spikes in the frequency domain. For example, a sine wave signal with a frequency of 10Mhz would normally appear as a wave, but in the frequency domain the X axis would become frequencies and the data appear as a spike upwards at this point.

Difficulties

The processing task was most difficult to implement since it required specific knowledge of the Kiss FFT package, including the functionality of its associated functions. For example, figuring out where to place code converting the first 128 bins of out[] to dB for display was difficult. Also, not taking a semaphore or taking the incorrect semaphore was an issue since this resulted in issues that do not produce obvious errors such as deadlocks.

Conclusions

The most important result of this lab was porting a real-time application to RTOS using fundamental RTOS objects like tasks, hardware interrupts, clocks, semaphores, and mailboxes. Also, real-time debugging experience was gained and proved to be a great tool for identifying and fixing bugs. Lastly, critical sections were found and protected by implementing semaphores around them.