Lab 2: RTOS, Spectrum Analyzer

ECE 3849

Samantha Comeau, sjcomeau@wpi.edu, Mailbox 733

William Schwartz, wrschwartz@wpi.edu, Mailbox 300

15 April, 2018

Table of Contents

Introduction

The goal of this lab was to port all of the lab 1 oscilloscope functionalities into a TI-RTOS through breaking up the project into tasks, Hwis, and clock threads. We ensured that all the previous code we had written would still run with the new timing constraints of using the TI-RTOS. The first thing that we did was convert the ADC ISR into a Hwi, and preserve the low latency through modifications to the TI_RTOS GUI. We then modified the button handling. To do this a clock instance in the RTOS GUI that calls a function every 5 milliseconds to post to a semaphore. We then have a task then pends on this semaphore that gest the user input from the buttons, and posts this information to a mailbox. The mailbox is then pended on in a separate task, the process the button task, and updates the globals appropriately to cause minimal shared data bugs. We then created a Waveform task, the searches for the trigger, and copies the waveform into a local buffer. The waveform task then signals the processing task using a semaphore. The processing task is responsible for getting all the data scaled and ready to be displayed. The Processing task then signals the Waveform task the find a new waveform, as well as the Display task to draw the processed waveform. The Display task is solely responsible for drawing the waveform on the screen. After all this functionality was working, we added another button for user input to allow the user to also see the frequency, to do this we modified the waveform task to get the values for the FFT, the processing that's to calculate the FFT, and the display task remained mostly unchanged.

Discussion

Overview

For this lab we took the starter code and modified it to integrate our code from Lab 1. This involved setting up tasks, events and Hwis to modify our code to implement the TI-RTOS (Texas Instruments Real Time Operating System). Most of our existing code from lab one remained unchanged. However this code was divided into different tasks that were signalled using semaphores, and a mailbox to keep track of the user input. Once these we had al the code from lab one working correctly, then then added Spectrum (FFT) mode, which allows the user to

view the frequency analysis of the waveform. Which for the circuit that we created in lab 1 consists of a large spike at around 20 kHz and aliasing on adjacent frequencies.

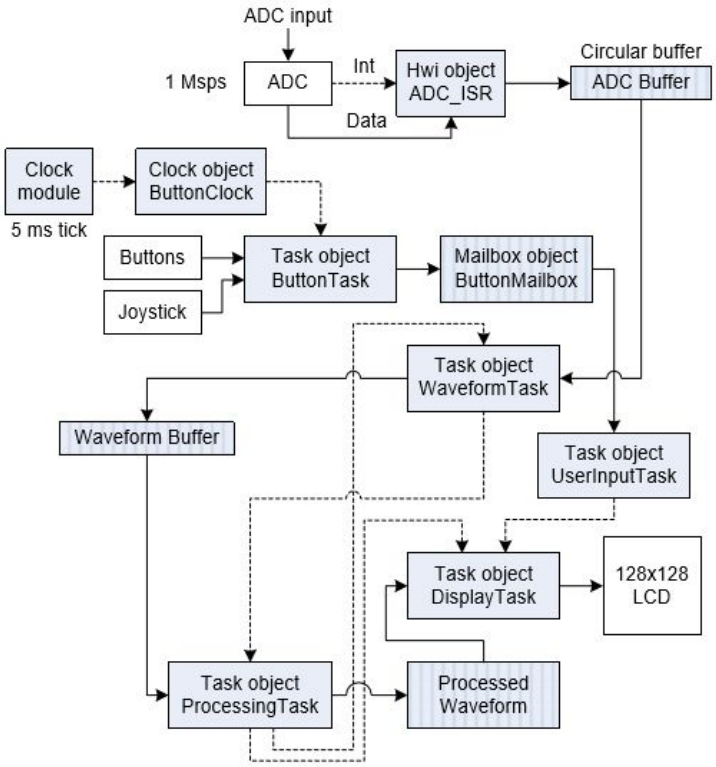

Figure 1. Overview of the implemented system

General Approach

Our general approach to this lab was to take our code from lab 1 and section it off into different tasks corresponding to what the code was responsible for doing. For example, our waveform task function was deemed responsible for getting the waveform and posting to the processing semaphore because the processing task relies on the waveform data to be available. Our code below demonstrates our approach.

```
void WAVEFORM_TASK_FUNC(UArg arg1, UArg arg2) // high
Priority!!
{
    IntMasterEnable();

    Semaphore_pend(WAVEFORM_SEM, BIOS_WAIT_FOREVER);

    if (display_mode == TIME_DOMAIN){
        get_waveform();
```

```
    } else {
        get_fft_waveform();
    }

    Semaphore_post(PROCESSING_SEM);
}
```

Code snippet 1. Demonstrating our general approach to modifying our code from lab 1, this excludes the code we added for the extra credit portion of the assignment.

ADC ISR

We modified our ADC interrupt service routine to instead be a hardware interrupt. This involved going into the TI-RTOS configuration file and adding a new Hwi. We set the priority threshold for Hwi disable to be 32 for all modules which will leave higher priority ISR's to execute without being interrupted by the RTOS, to make sure that this event doesn't miss its extremely tight deadline of collecting 1,000,000 samples per second. This hardware interrupt is triggered whenever the ADC captures a new samples a new from the waveform. To make sure that the RTOS hardware interrupt function triggered on the ADC, we had to add the Vector Number of ADC0 to the RTOS hardware interrupt module, we found in the data sheet that this value was 62. This conversion was the only required modification from lab one to get our waveform to continue to be sampled, into our global buffer `gADCBuffer`. This Hardware interrupt has the absolute highest priority of our entire system.

Button Handling

To change the way we handle buttons, we began by creating a clock module. This clock instance has the initial timeout and period set to 1. This is called every 5 milliseconds and it calls the `clk1Fxn` function that posts to our `BTN_CLK_SEM`. Our `BTN_CLK_SEM` is a binary FIFO.

Our `BTN_TASK` is the task that handles button presses. It is a priority 6 task, which is the highest priority, and has a stack size of 1024. The function `BTN_TASK_FUNC` is signaled by the semaphore and it calls our original code from lab 1. The code from lab 1 was also modified to use a mailbox instead of our original FIFO buffer. This involved setting up a mailbox which we coined `BTN_PRESS_MAILBOX`. Our mailbox takes in 4 chars and a

maximum of 11 messages. This allows us to add all of the `presses` variable to the mailbox for us to handle later by the `USER_INPUT_TASK`. We added a button press for switching between frequency and time domain. We used the joystick select for this functionality. The rest of our code for debouncing and such was the same.

User Input Task

The user input task, `USER_INPUT_TASK`, has a priority of 4 and a stack size of 512. This task pends on the mailbox that handles the button presses. Once a button is pressed and processed into the mailbox, the user input task will take that button press in and determine which button was pressed. The code that we use to do this is shown below.

```
if(presses & 4){ // trigger up
    trigger_direction = 1;
} else if(presses & 8){ // trigger down
    trigger_direction = 0;
} else if(presses & 128){ // increase voltage
    if(voltage_index <= 2){
        voltage_index++;}
} else if(presses & 256){ // decrease voltage
    if(voltage_index >= 1){
        voltage_index--;}
} else if(presses & 16){ // change the display mode
    display_mode = (display_mode+1) %2;
}
```

Code snippet 2. The code for determining button press reaction.

Waveform Task

Our waveform task, `WAVEFORM_TASK`, was set to have a priority of 5 and a stack size of 512. This calls our `WAVEFORM_TASK_FUNC` which pends on the `WAVEFORM_SEM`. The semaphore is a binary FIFO which is initialized with an initial count of 1. This is so that we can get the first waveform when the program starts running initially. Based on the display mode, either time domain or frequency domain, we call the corresponding function to generate the waveform. When the time domain function is called, we search for the trigger, grab 64 samples behind the trigger and 63 samples ahead of the trigger and copy them into a local buffer. When the frequency domain function is called, we simply copy the most recent 1024 samples and copy

them into a local buffer to perform the FFT on. After a waveform is sampled, we post to the PROCESSING_SEM so that the waveform samples can then be processed.

Processing Task

The processing task, PROCESSING_TASK, has a priority of 1, which is the lowest priority, and a stack size of 2048. This is used to call the PROCESSING_TASK_FUNC which pends on the PROCESSING_SEM. After the semaphore has been posted to, we can then calculate the proper waveform given the domain, either time or frequency. Once we have processed a valid waveform, we post to the DISPLAY_SEM so that the display can be generated, and also to the WAVEFORM_SEM, so that we can trigger the generation of a new waveform. The processing task handles all of the calculations involved with transforming the waveform to fit on the screen, as well as the calculations to perform the FFT.

Display Task

Our display task, DISPLAY_TASK, is a priority 2 task with a stack size of 2048. This task pends on the DISPLAY_SEM which is posted after a processed waveform is ready to be displayed. Based on the domain, time or frequency, we display the proper gridlines, header/footer and waveform on the screen. We don't need to post to any semaphores after we've displayed on the screen, because the processing task, kicks off the processes of collecting a new waveform to be displayed.

Spectrum FFT mode

Inside our processing task function we had to process the waveform for spectrum FFT mode. The code that we used is shown below.

```
for (i = 0; i < NFFT; i++) {    // generate an input waveform
    in[i].r = locoFFTBuffer[i]/4096.0f; // 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 (j = 0; j < 128; j++){
    pixel_y = (20 - roundf(10.0f * log10f(out[j].r*out[j].r +
out[j].i*out[j].i))) - 65480;
```

```
    processedBuffer[j] = pixel_y;
}
```

Code snippet 3. The code to generate the FFT waveform.

As this snippet shows, we used the provided `kiss_fft` function provided in the instructions with this lab. This function does all of the hard calculations for us, and we just had to figure out how to use the function appropriately. We copy our sampled waveform into the `in[]` buffer of the kiss fft function, were then call the function, and copy the calculated frequency domain information out of the `out[]` buffer, we only take the first 32 samples as specified by the lab instructions. This information is the scaled appropriately in the `pixel_y` variable and copied into our processed buffer to be displayed on the screen.

Extra Credit

To complete the extra credit for this lab we were required to time the latency response time and relative deadline of the Button task, we did this using timer one, and finding at what point the button task was run after it was called. The approach used to find this information was almost verbosely copied from an online example. The relative deadline of this task was known, as we set it to 5 milliseconds to check if a button has been pressed, making our deadline 5 milliseconds.

For the second part of the extra credit we found the response time and calculated the relative deadline of the trigger search and waveform copy. This again was code that was almost verbosely copied from an example posted online. We do this by calling `SysTickValueGet()` this function allows us to know how many systems ticks have been counted on a 24 bit counter, we call this function before our waveform code, after our waveform code, and one sequentially after the other. The first is to get the start time, the second is the get the end time, and the sequential count is to get the bias to be subtracted off. With this information we can time how long the function takes to run. We calculated the relative deadline. We know that the ADC sampling at 1,000,000 samples per second, and we have a buffer that can hold 2048 samples, so our relative deadline is less than 2 milliseconds. To be more specific with this calculation we can say that we must trigger and copy before our 128 samples are overwritten, and we allow for a trigger search to go half the size of the array, so our real deadline is a little over 1 microsecond.

Conclusion

In this lab we revised our lab 1 in order to make this a real time operating system. This involved setting up hardware interrupts, tasks, events, semaphores and clocks so we could keep all the same functionality. We also implemented an FFT mode and another button which were additions of code from lab 1. We learned how to properly set priorities and use semaphores to ensure our code flow was reflective of how we wanted the program to be running. For example, we had to ensure that the processing task relied on the waveform task which we did through the use of posting and pending on a semaphore. Overall in this lab we learned how to use the configuration file GUI to setup the different components of a RTOS while also ensuring that our code met deadlines properly.