ECE3849: RTOS Spectrum Analyzer
B-Term 2020

Adam Yang
Prudence Lam

November 24, 2020

1 Introduction

The objective of the lab was to port the 1 Msps digital oscilloscope from Lab 1 to a TI-RTOS. Using objects in the RTOS, the oscilloscope was broken up into multiple threads. Additionally, a spectrum mode (FFT) was implemented to provide new functionalities to existing task threads. These changes were reflected in the oscilloscope display on the LCD.

2 Discussion and Results

All of Lab 1's functionalities were implemented using TI-RTOS threads (tasks, Swi, Hwi). The ADC ISR was converted into an Hwi, and the `main()` code in Lab 1 was divided into Task threads. Additionally, the state of the buttons were read and processed in two separate tasks, and communicated with each other using a Mailbox rather than a FIFO. **Figure 1** provides the block diagram summarizing the top level design.

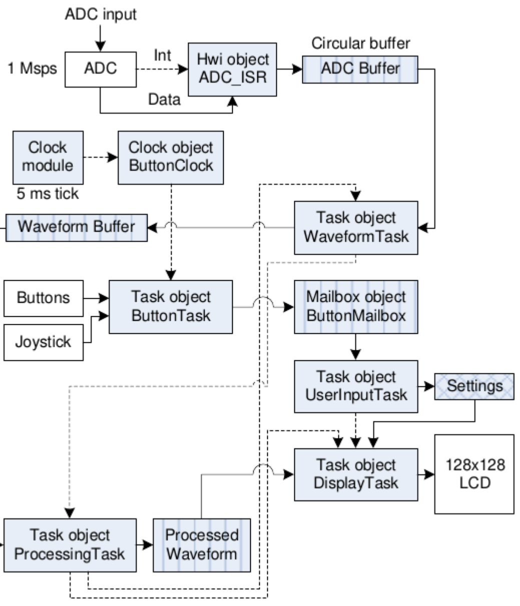


Figure 1: *(Referenced from the Lab 2 Document)* Block diagram depicting the structure of the ported oscilloscope. The smooth shaded boxes represent software threads and modules, whereas textured shaded boxes represent important shared data and inter-task communication objects. The solid lines depict data flows and the dotted lines represent signaling or synchronization. The boxes are arranged such that their priority is represented from high to low vertically

Each feature is explained in further detail in **System Requirements**.

2.1 System Requirements

1. **ADC Hwi**

   The ADC hardware interrupt (Hwi) was configured using the TI-RTOS Graphical User Interface (GUI). An ADC Hwi object was created and set to priority 0, giving it the highest priority on the interrupt controller. The vector number in the TM4C1294NCPDT datasheet was used as the Hwi object interrupt number.

   Table 1: An entry in the Interrupt Table. The number on the far left is the Vector Number, which is used to specify the interrupt number of the ADC Hwi object.

62	46	0x000.00F8	ADC1 Sequence 0

2. **Waveform Processing**

   The `main()` code in Lab1 was divided into three separate task threads (from highest to lowest priority): Waveform, Display, and Processing. Using semaphores, the waveform task searches for the trigger and copies a small sample into a waveform buffer. It then posts to a semaphore signaling the Processing task to begin (see **Appendix A** for a code snippet of implementation).

   Similarly, a buffer was created to store 1024 samples from the Waveform Task for spectrum mode, which essentially follows the same waveform task processing as oscilloscope mode.

   In the processing task, the waveform from Waveform Task is scaled for the LCD display and stored in a global buffer, `displayBuffer`. It then posts to two semaphores to signal the Display and Waveform tasks for display and data capturing respectively (See **Appendix B**). Similarly, for spectrum mode, memory was allocated for a second buffer, `out_db`, for the FFT. The lowest 128 frequency bins of the spectrum were converted to their respective magnitudes [dB]. The bins were scaled at 1 dB/pixel to accommodate the display by logarithmically scaling the output values.

   The display task is responsible for displaying the signal onto the LCD screen. After being signaled by other tasks, the display task draws one frame with the appropriate waveform and display settings. Depending on which display mode the TI-RTOS is on, different waveforms are processed onto the LCD screen. While the oscilloscope mode displays a scaled version of the square PWM wave, the spectrum mode displays a frequency breakdown of the waveform with modified grid and display settings. The code for this remains the same as Lab 1 for the oscilloscope mode, with slight modifications for spectrum mode.

3. **Button Scanning**

   Periodic button scanning was implemented with a 5ms clock, which provided a signal for the button task thread (`buttonTask`) to begin. During execution of this task, the state of the buttons are read and their respective character IDs stored in a Mailbox for data protection, contrasting the use of a FIFO in Lab 1. The individual button states are processed by a lower priority task, named `userInputTask`, and changes the voltage scales, mode, or trigger edge accordingly.

   An additional button (Boosterpack button SW1) was added to the user interface to decide whether the system is in oscilloscope or spectrum mode. Similar to Lab 1, the state of the Launchpad buttons SW1 and SW2 determined if the voltage scale increments or decrements respectively. As the Launchpad SW1 button was assigned to the mode, the Launchpad SW2 button was configured to toggle the trigger slope.

4. **Spectrum (FFT) Mode**

   The Spectrum mode was implemented using the Kiss FFT package. In this mode, a Fast Fourier Transform (FFT) was performed on the ADC data. As mentioned above, memory was allocated for 1024 samples in the Processing task using the function `kiss_fft_cpx` from the Kiss FFT library. The lowest 128 frequency bins were then converted to decibels using the decibel equation,

   ```
   out_db[i] = 160 - ((10.0f) * log10f((out[i].r * out[i].r) + (out[i].i * out[i].i)));
   ```

   (see **Appendix B**). For display, the data points were scaled at 1 decibel per pixel.

   Instead of time and voltage scales, the Display task prints frequency and decibel scales in this mode. For the lab, 20kHz and 20 dBV were used. Additionally, the parameters to draw the vertical lines of the grid were modified from Lab 1 such that the first vertical line is situated at zero frequency (x = 0). Instead of iterating from -3 to 4, the `for` loop ranges from 0 to 7.

2.2 Difficulties

The first difficulty encountered in the lab occurred within the individual Task threads. Using a debugger, it was determined that all tasks were being blocked because no semaphores were being posted at startup. Subsequently, the system entered the following idle loop:

```
Void Idle_loop(UArg arg1, UArg arg2) {
    while (1) {
        Idle_run();
    }
}
```

The solution to this was to set the initial count of the waveform semaphore to 1 instead of 0.

Another difficulty occurred in displaying the pwm signal at varying voltage scales. The pwm signal would disappear at and below the 0.5 V-per-division level. From the debugger, it was determined that the intermediate buffer `displayBuffer` was of type `uint16_t`, meaning that any negative values would cause an overflow and result in an offscreen signal.

The solution was to change the type of `displayBuffer` to a one of a primitive `int`, which is signed.

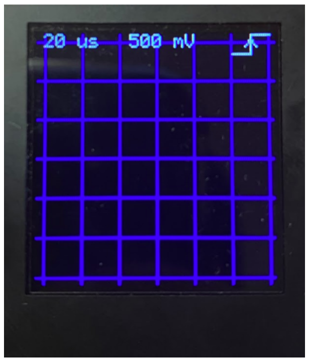


Figure 2: An offscreen signal that resulted from displayBuffer being an unsigned 16-bit integer

2.3 Results

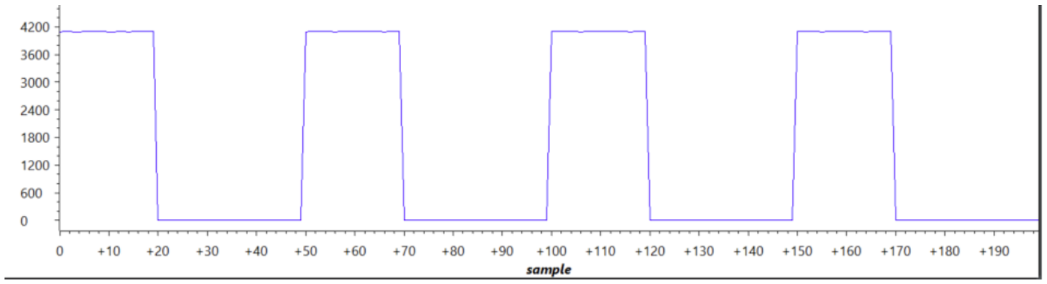


Figure 3: Signal output when Oscilloscope mode is toggled. The square wave indicates that $gADCBufferIndex$ is being written and read from properly.

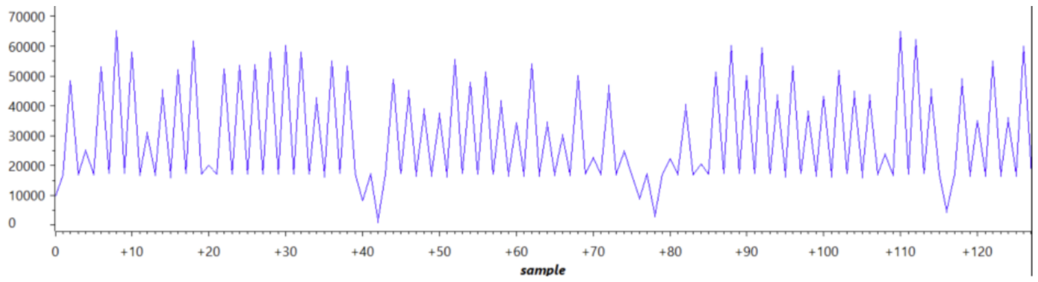
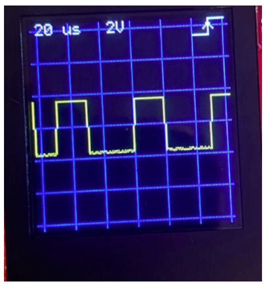
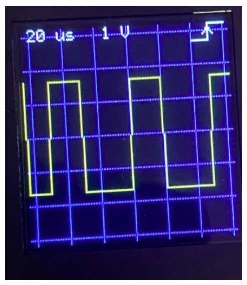


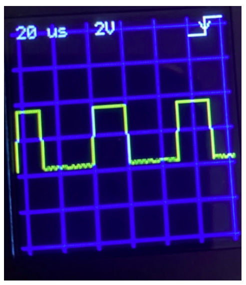
Figure 4: Signal output when Spectrum mode is toggled. Here, values from *out_db* are being read after a Fast Fourier Transform (FFT)



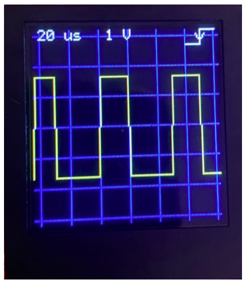
(a) Oscope mode
Rising edge, 2 Vpd



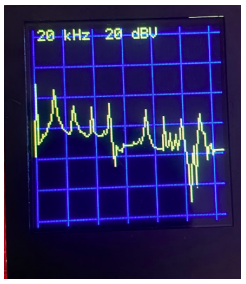
(b) Oscope mode
Rising edge, 1 Vpd



(c) Oscope mode
Falling edge, 2 Vpd



(d) Oscope mode
Falling edge, 1 Vpd



(e) FFT mode
20kHz, 20 dBV

Figure 5: Displayed PWM signal for various trigger slopes and voltage scales for both oscilloscope and spectrum mode

3 Summary and Conclusion

Upon completion of the lab, the 1 Msps digital oscilloscope application from Lab 1 was ported to a TI-RTOS. An second, toggleable mode allowed the system to process the ADC data using the Fast Fourier Transform to create a spectrum analyzer. Lab 2 enabled work with fundamental RTOS objects, such as Tasks, Hwi, Semaphores, Clocks, and Mailboxes, in multiple threads. To improve the spectrum analyzer, a Blackman window could be implemented to reduce spectral leakage and produce smoother curves. Although many difficulties were encountered during the process, RTOS provided numerous debugging features that allowed for close examinations of RTOS objects. The ability to observe whether Tasks or Semaphores were blocked proved crucial to the success of this lab.

4 Appendices

See code files for full implementation.

4.1 Appendix A

Waveform task reading a small sample of the ADC buffer and stores it in a sample buffer.

```
void waveformTask(UArg arg1, UArg arg2) {
    <Initialize variables>

    while(1) {
        Semaphore_pend(semaphoreWaveform, BIOS_WAIT_FOREVER);
        trigger = trig_slope ? RisingTrigger() : FallingTrigger();

        if(oscillo_mode) {  // oscilloscope mode
            for (i = 0; i < LCD_HORIZONTAL_MAX - 1; i++) {
                sample[i] = gADCBuffer[ADC_BUFFER_WRAP(trigger - LCD_HORIZONTAL_MAX / 2 + i)];
            }
            Semaphore_post(semaphoreProcessing);
        }
        <Spectrum mode>
        }
    }
}
```

4.2 Appendix B

After being signaled by another task, the Processing task scales the waveform to a form ready to be displayed on the LCD display.

```
void processingTask(UArg arg1, UArg arg2) {
    <Init locals>

    while(1) {
        Semaphore_pend(semaphoreProcessing, BIOS_WAIT_FOREVER);

        if(oscillo_mode) {  // oscilloscope mode
            fScale = (VIN_RANGE * PIXELS_PER_DIV) / ((1 << ADC_BITS) * fVoltsPerDiv[vpd]);
            for (i = 0; i < LCD_HORIZONTAL_MAX - 1; i++) {
                // Copy waveform into another buffer
                displayBuffer[i] = LCD_VERTICAL_MAX / 2 - (int)roundf(fScale * ((int)sample[i]
                                                                            - ADC_OFFSET));
            }
            Semaphore_post(semaphoreDisplay);
            Semaphore_post(semaphoreWaveform);
        }  else {  // spectrum mode
            for (i = 0; i < NFFT; i++) { // generate an input waveform
                in[i].r = fft_sample[i]; // real part of waveform
                in[i].i = 0;             // imaginary part of waveform
            }

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

            // 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)));
            }
            Semaphore_post(semaphoreDisplay);
            Semaphore_post(semaphoreWaveform);
        }
    }
}
```