ECE3849 Lab 2

Nam Tran Ngoc
Yigit Yuan
Mailbox 388

December 5, 2016

1 Introduction

Our previous assignment was to build a real-time digital oscilloscope that is capable to gather 500.000 samples each second and display the signal on the built in display. In this lab, we take this a step further by implementing the same features using TI's SYS/BIOS Real Time Operating System (RTOS). We created various tasks and RTOS objects to mimic the foreground - background system from the previous lab. Furthermore, we integrated the Kiss FFT package to compute and display the spectrum of the sampled signal.

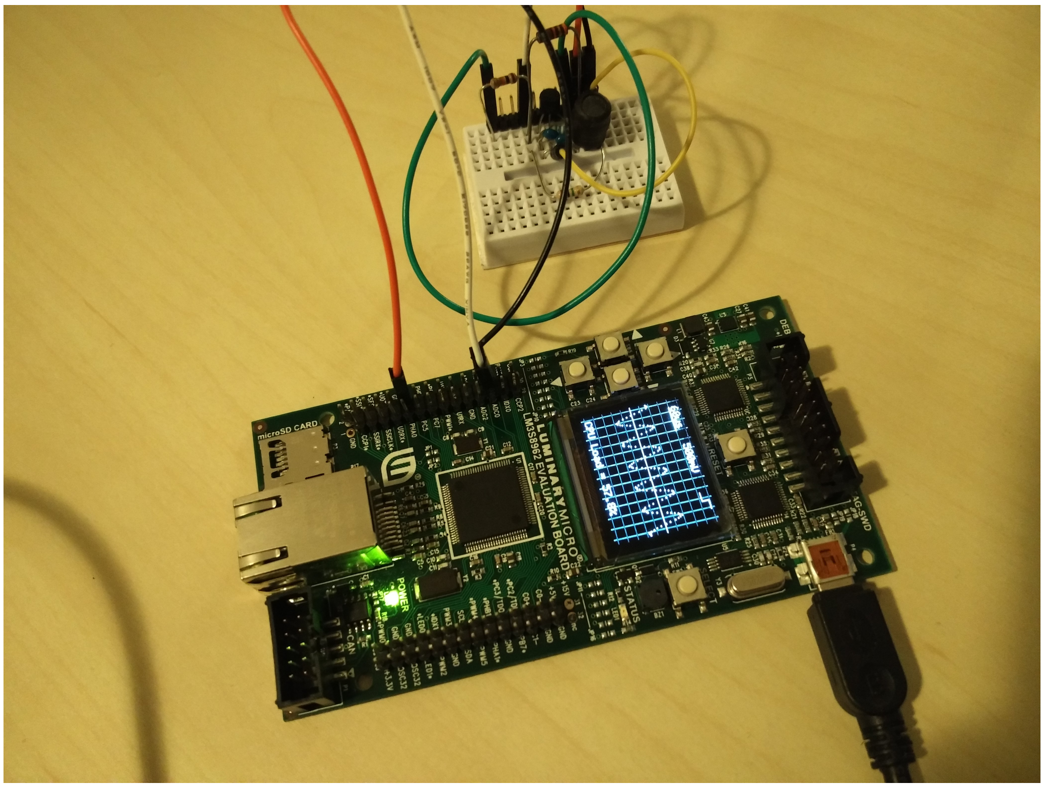

Figure 1: Overview of the circuit

2 Discussion and Results

In this part we will discuss the methodologies used during the projects, along with their results.

2.1 Creating a new SYS/BIOS project

We started out by creating a new sample SYS/BIOS project, with the basic files to get started and an `app.cfg`

Before moving on to the project, we configured our debugger to automatically reset the board every time we upload by checking the "Reset target during program load to Flash memory." option. However, for some unknown reasons, this lead to our board being disconnected from the IDE Debugger. In the end we opted not to use this option, and have not experienced any crashes so far.

2.2 Initializing

Our previous code relied heavily on timers and interrupts to achive real time functionality. Our current task is to replace the previous foreground – background system with an implementation using SYS/BIOS provided functions and features. For this, we first started by going through the code and take out all the timer and interrupt related code. Timers are set by RTOS so our implemetation of timed functions are not recognized. Also RTOS disables all interrupt related functions (except IntMasterEnable and IntMasterDisable), as they would interfeare with RTOS provided real time features. After cleaning, our setup pretty much looked like this:

```c
Void main()
{
    IntMasterDisable();
    initializeButtons();
       // Initialize Buttons
    initializeADC();
       // Initialize IRQ for ADC
    initializeScreen();
    RIT128x96x4Init(3500000);

    BIOS_start();
}
```

assets/main.c

The timer initialization function is taken out. The button initialization function hasn't changed at all and the ADC code changed slightly (interrups are taken out) as below.

```c
void initializeADC()
{
    SysCtlPeripheralEnable(SYSCTL_PERIPH_ADC0); // enable the ADC
    SysCtlADCSpeedSet(SYSCTL_ADCSPEED_500KSPS); // specify 500ksps
    ADCSequenceDisable(ADC0_BASE, 0); // choose ADC sequence 0; disable before configuring
    ADCSequenceConfigure(ADC0_BASE, 0, ADC_TRIGGER_ALWAYS, 0); // specify the "Always" trigger, highest priority
    ADCSequenceStepConfigure(ADC0_BASE, 0, 0, ADC_CTL_CH0 | ADC_CTL_IE | ADC_CTL_END); // in the 0th step, sample channel 0
     // enable interrupt, and make it the end of sequence
    ADCIntEnable(ADC0_BASE, 0); // enable ADC interrupt from sequence 0
    ADCSequenceEnable(ADC0_BASE, 0); // enable the sequence. it is now sampling
//  IntPrioritySet(INT_ADC0, 32); // set ADC interrupt priority in the interrupt controller - lower than main ISR
//  IntEnable(INT_ADC0); // enable ADC interrupt
}
```

assets/main.c

2.3 Creating the ADC Hwi

Next we created an Hwi Object for the ADC, instead of declaring/initializing ISR like Lab 1. We created the Cortex-M3 specific Hwi module, make the priority 0 and the interrupt number as 30 - the interrupt number corresponds to ADC0 on the LM3S8962 board. Then we refer to the `ADC_ISR` function by reference.

Declaring the Hwi Object allows us to sample data from the ADC every $2\mu s$, and at priority 0 (highest), the Hwi is zero-latency.

2.4 Button Scanning

We then port the button scanning functions to SYS/BIOS. In the last lab, we used timer interrupt to periodically scan the buttons and put them in a buffer array. Since we are using SYS/BIOS, we will be creating a Clock object instead.

We created a Clock instance with a period of 5ms, following the lab instruction, which then call the `Button_Clock` function on every clock tick.

```c
void Button_Clock(UArg arg) {
        Semaphore_post(sem_button);

        if(displaySignal)
            Semaphore_post(sem_waveform);
        else
            Semaphore_post(sem_fft);
}
```

assets/main.c

We also added a flag check for `displaySignal`, which tells us if Spectrum mode is active or not, and re-render accordingly.

Next, we created three more objects: A Task, for button scanning, A Semaphore, for signaling the button Task, and a Mailbox, for posting button IDs to. We would then have the Button Task looking like this.

```c
void Button_Task() {
    IntMasterEnable();
    while(1) {
        Semaphore_pend(sem_button, BIOS_WAIT_FOREVER);

        unsigned long presses = g_ulButtons;
        ButtonDebounce((~GPIO_PORTE_DATA_R & (GPIO_PIN_0 | GPIO_PIN_1 | GPIO_PIN_2 | GPIO_PIN_3)) << 1);
        ButtonDebounce((~GPIO_PORTF_DATA_R & GPIO_PIN_1) >> 1);
        presses = ~presses & g_ulButtons; // A List of Button Presses

        char buttonPress;

        if (presses & 1) {
            buttonPress = 'S';
        } else if (presses & 16) {
            buttonPress = 'R';
        } else if (presses & 8) {
            buttonPress = 'L';
        } else if (presses & 4) {
            buttonPress = 'D';
        } else if (presses & 2) {
            buttonPress = 'U';
        }

        if (presses != 0) {
            Mailbox_post(Mailbox_ButtonQueue,
               &buttonPress, BIOS_NO_WAIT);
        }

    }
}
```

assets/main.c

The internal button scanning logic mostly stayed the same, with the addition of a semaphore pending at the beginning of the Task while look, which serves as a blocking call while waiting for the Clock signal. The same pattern can be seen in every other Tasks as well.

2.5 Creating Task Objects

We added three more Tasks to the mix, porting over from the last lab, taking care so that shared data is being protected. The three Tasks behave the same as the last lab, however, there was a slight change in each Task with the addition of the new Spectrum mode.

For UserInput_Task, we added a check for the Select button press, as seen from the Button_Clock function above.

```c
        if(currentEntry == 'S' || currentEntry == 's')
        {
            if(displaySignal)
```

```c
                displaySignal = 0;
            else
                displaySignal = 1;
    }
```

assets/main.c

For the Display_Task, we also added a check, drew the Overlay for Spectrum Mode, and draw the transformed data. The decision of using `DrawPoint` over `DrawLine` is purely aethestic, as the waveform looks better in our opinions.

```c
        if(displaySignal)
        {
            drawGrid();
            drawOverlay();
            drawTrigger();
        }
        else
        {
            drawFFTOverlay();

            int i = 0;
            while(i < 128) {
                DrawPoint(i, -60 + fftBuffer[i],
                   COLOR_SIGNAL);
                i++;
            }
        }
```

assets/main.c

Lastly, for the Waveform task, we used the same check to copy the buffer over and redirect the Task to FFT.

```c
        case 0: //Spectrum mode
            i = 0;
            j = 0;
            for (i= g_iADCBufferIndex - 1024; i !=
               g_iADCBufferIndex; i++){
                g_tempBuffer[j] =
                   g_psADCBuffer[ADC_BUFFER_WRAP(i)];
                j++;
            }
            break;
```

assets/main.c

2.6 Spectrum Mode

The spectrum mode turned out to be the more straightforward than we initialy thought. The first few lines serve as initialization, as suggested by the lab instruction.

```c
    static char
       kiss_fft_cfg_buffer[KISS_FFT_CFG_SIZE];
    size_t buffer_size = KISS_FFT_CFG_SIZE;
    kiss_fft_cfg cfg;
    static kiss_fft_cpx in[NFFT], out[NFFT];
    int i;

    cfg = kiss_fft_alloc(NFFT, 0,
       kiss_fft_cfg_buffer, &buffer_size);
```

assets/main.c

Then, during the while loop, we convert the data from the buffer to a struct that the KISS_FFT library can understand, with the imaginary and real portions, which is then fed into the FFT function. Finally, we go through the whole screen width (128 pixels), and fill the data in the display buffer. We are using a separate buffer from the Waveform buffer so that we don't have to recalculate every time we switch back and forth between the two modes.

```c
        for (i = 0; i < NFFT; i++) {
            in[i].r = g_tempBuffer[i];
            in[i].i = 0;
        }

        kiss_fft(cfg, in, out);

        // Update the fftBuffer:
```

```
        i = 0;
        while(i < 128)
        {
            fftBuffer[i] = (log10(out[i].r*out[i].r +
               out[i].i*out[i].i)*(-10))+180;// VALUE
               FROM FFT BUCKET
            i++;
        }
```

assets/main.c

Last finishing touches have us change the scale and the offset (currently at -10 and 180, respectively) so that the transformed waveform fits nicely on the screen. Since we opted not to work on the extra credit (partially owe to a lack of sleep), we don't have to worry about accurately scaling the waveform on the screen, although it should not be too hard to calibrate by using the software debugger.

3 Conclusion

In this lab we have implemented real time features of a digital oscilloscope using an RTOS distribution SYS/BIOS, running on a Stellaris LM3S8962 microcontroller. The foreground-background system from the previous assignment is completely replaced with the RTOS tasks and shared data objects. We have observed significant increase on the performance and responsiveness of the device, as the CPU usage is now dynamically scaling to 100% at all times, not wasting any flops available on the system.