ECE3849: Digital Oscilloscope
B-Term 2020

Adam Yang
Prudence Lam

November 10, 2020

1 Introduction

The objective of this lab was to implement a 1 Msps digital oscilloscope on the LCD display with selectable voltages and trigger slope. Calculations on CPU load were performed and outputted to the display. Implementation of the lab required several peripheral devices to be configured, such as the ADC. The user interface was provided by two buttons on the Launchpad and one on the Boosterpack. The lab enabled work with timing constraints, interrupt preemption and prioritization, and shared data to create a real-time application.

2 Discussion and Results

The digital oscilloscope required several peripherals to be configured, namely the ADC and the Launchpad and Boosterpack buttons. Both of these required ISRs. An ADC buffer was used to store the raw ADC data for later use, whereas a FIFO was implemented for the buttons for both read and write operations. A fixed time scale of 20 microseconds was used. The following block diagram summarizes the system from a top level.

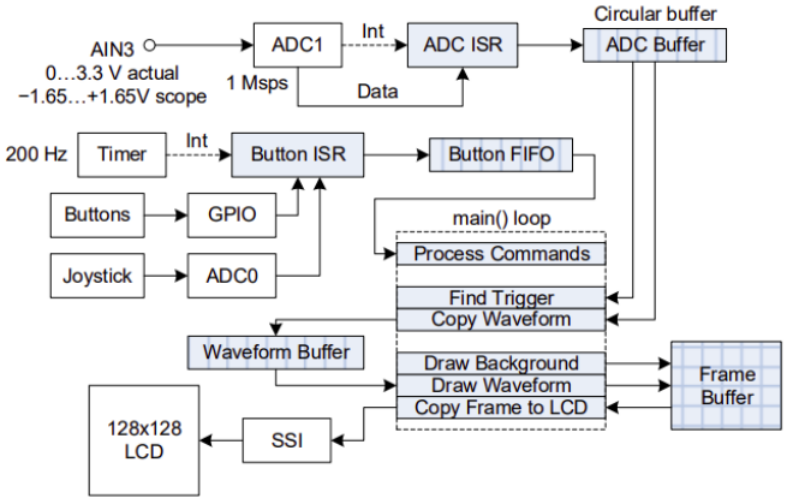


Figure 1: Block diagram of final implementation of how data is shared between the ADC, the Buttons, and main().

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

2.1 System Requirements

1. **Signal Source**

   A 20 kHz PWM square wave was produced on the PF2 and PF3 outputs with a 40% duty cycle. Using the driver libraries, the GPIO pins and the PWM signal source were initialized in `main()` through a call to `SystemInit()`. To obtain a period of 20 kHz, the number of cycles was obtained by dividing the PWM clock frequency with 20k, and passing it as a parameter to `PWMGenPeriodSet`.

2. **ADC Sampling**

   To configure the ADC on the launchpad to acquire ADC samples at 1,000,000 samples per second, the TivaWare Peripheral Driver Library User's Guide was consulted to determine the appropriate function arguments. **Fig. 2** depicts the process.

```
SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOE);
GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_0); // GPIO setup for analog input AIN3
```

Figure 2: This figure depicts a code snippet of the configuration method implementation. The analog input AIN3 was used, which corresponded to the GPIO pin E0.

The configuration function SysCtlPeripheralEnable() enables a given peripheral, or GPIOE. Thus, `SYSCTL_PERIPH_GPIOE` was passed as the function argument. Similarly, the configuration function `GPIOPinTypeADC(GPIO_PORTE_BASE, GPIO_PIN_0)` sets the pin modes of the desired pin(s). `GPIO_PORTE_BASE` and `GPIO_PIN_0` were subsequently passed as function arguments.

For configuration of the ADC interrupt service routine, the ADC interrupt flag was first cleared to allow an ADC sample to be read from the ADC FIFO. Similar to previous configurations, the TM4C1294NCPDT Datasheet was consulted to determine the appropriate register definitions to use. The ADC1 required sequence0 port in the ADCISC register to be used (see Fig.3 for implementation).

```
ADC1_ISC_R = ADC_ISC_IN0;          // clear ADC1 sequence0 interrupt flag in the ADCISC register
```

Figure 3: This figure depicts the code snippet used to clear the interrupt flag from the ADCISC register. The macro ADC ISC IN0 reads and clears the status of the sequence0 interrupt flag.

To read the ADC samples from the ADC1 sequence0 FIFO, The following definition was used:`ADC1_SSFIFO0_R`.

3. **Trigger Search**

   To display the general shape of the waveform, triggering was implemented. To extract the ADC samples from the buffer that adheres to a trigger, or the waveform crossing the specified voltage in a given direction, a trigger search function was implemented. In particular, the rising edge trigger search function was created using the algorithm suggested in the lab 1 document.

   To begin, the trigger index was initialized at half a screen behind the most recent sample (i.e. the end of the gADCBuffer buffer. To complete this step, the expression `LCD_HORIZONTAL_MAX / 2` was subtracted from the `gADCBufferIndex`. Then, the index was decreased until the trigger level was in the desired direction. This was achieved using a `for` loop that compares the current sample with an older sample. For a rising edge trigger, the current sample must be greater than or equal to the `ADC_OFFSET` value, whereas the older sample must be less than the offset value. Finally, the trigger index was reset back to how it was initialized in the case that the trigger level was not found (writing `gADCBufferIndex - LCD_HORIZONTAL_MAX / 2` to x). These three steps determined where in the `gADCBuffer` the index of a viable trigger level was. A `for` loop and the recently found trigger index was used to copy samples from half a screen before and after the trigger index of the `gADCBuffer` to a local buffer (see **Appendix A** for code implementation).

   Similarly, the falling edge trigger search function employs the same algorithm. However, the current sample must be less than the `ADC_OFFSET` value, while the older sample must be greater than or equal to the offset value.

4. **ADC Sample Scaling**

   As the raw ADC samples are not processed in a way that can be properly outputted, scaling was implemented. The middle of the ADC range was expressed by the value corresponding to 0 Volts. Appendix A shows the code used to scale the samples.

   The scale factor was determined as the ratio of the product of the voltage range (3.3 V) and the number of pixels per division on the LCD (20) to the product of ADC bits and the desired volts-per-division. To draw the pixels on the waveform, a ‘for‘ loop was set to iterate through the number of pixels on the screen, or 128. To prevent

shared data bugs and the ADC values from going out-of-bounds, the raw ADC samples were first copied into a local buffer of size 128. The y-position of the pixel was found by subtracting half of the height of the LCD display with the scaled individual ADC samples (to the nearest integer). For interconnected lines, the function the line drawing function `GrLineDraw` was called, with the X and Y coordinates of a sample along with the one after it passed as parameters.

5. **Buttons**

   Two buttons on the Launchpad board and one on the Boosterpack provided the user interface. To do this, a FIFO with a maximum capacity of 10 events was used. In the ISR used for reading and debouncing the state of the buttons, the function `fifo_put` was called to write three characters, a, b, and e, into the queue. Then, using `fifo_get` in `main()`, the values were read. As the read operation in the FIFO is faster than the write operation, the FIFO should in theory never be full.

   Depending on the character read, the index of the voltage scale array, or the volts per division (`vpd`), increased or decreased. The voltages in the array were ordered from lowest to highest, meaning that an increase in vpd meant an increase in voltage, and vice versa. Set to 4 on default (2 V), the index increments if the character 'a' is read, or the value corresponding to SW1 on the Launchpad board. The index decrements if 'b' is read, or when the Launchpad SW2 button is pressed. Finally, if 'e' is read, meaning the SW1 button on the Boosterpack was pressed, the trigger slope changes.

6. **Measuring CPU Load**

   To measure the CPU load, `ece3849_int_latency` was followed (see **Appendix C** for code implementation). `Timer3` was configured to be in one-shot mode, whereas its timeout interval was set to 10 ms. With the one-shot mode, the only parameter that needed to be changed was the time-out interval argument. From the example project, the timeout interval argument of `TimerLoadSet` with the value `gSystemClock - 1` corresponds to a 1 second timeout interval. It was inferred that an argument value of `gSystemClock/100 - 1` essentially corresponds to a 10 millisecond timeout interval.

   Calculation of CPU load consisted of reading from the load count function before and after the `while` loop in `main()`. After that, one minus the ratio of `cpu_load` to `cpu_unload` was used to represent the CPU load of the system.

2.2 Difficulties

The first difficulty encountered in this lab was due to the absence of the "extern" keyword when using variables from other files. This resulted in errors much like the one below,

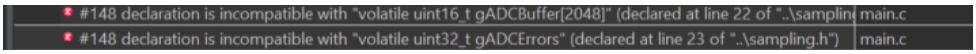


Figure 4: This figure depicts two errors due to the absence of the "extern" keyword.

After some research, it was concluded that the extern keywords must be applied to all files except for the .c file in which the variable was defined.

The second difficulty encountered was due to the `ADC_ISR` not being included in the interrupt vector table. This issue was first discovered when there was a black screen at the program startup. The debugger tool was used to determine that the program was essentially going to the `IntDefaultHandler()` right when `MasterIntEnable()` was called. This indicated an issue within the interrupt vector table, where it was revealed that the `ADC_ISR` was not included in the vector table.

The final difficulty encountered was relatively minor, but very noticeable: the displayed signal had periodic flickering. After countless tests to determine the cause of the issue, it was determined that a mere counter variable had to be moved inside the `ADC_BUFFER_WRAP` function. This fixed the issue because prior to this discovery, the `gADCBuffer` was essentially trying to access values outside of the `ADC_B_FFER_WRAP`, causing uncontrolled spikes in our PWM signal.

2.3 Results

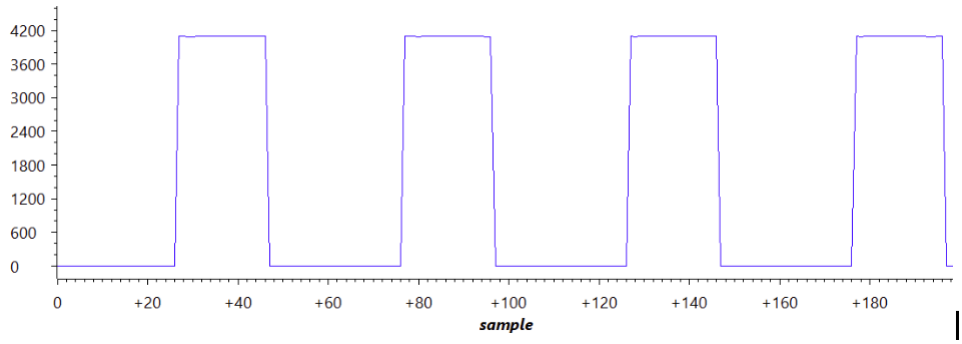
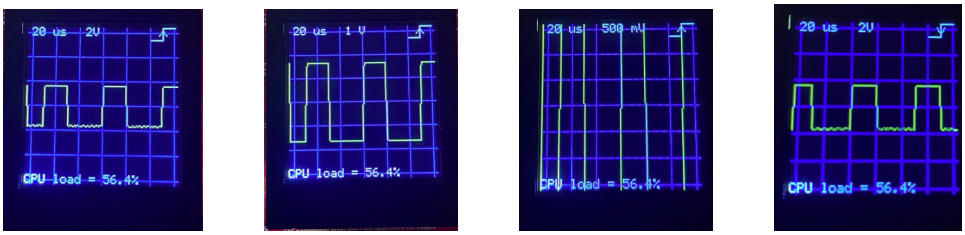


Figure 5: Sample data from the gADCBuffer. It is clear that correct information is being written into the gADCBuffer.



(a) Rising edge, 2 Vpd

(b) Rising edge, 1 Vpd

(c) Rising edge, 500 mVpd

(d) Falling edge, 2 Vpd

Figure 6: Displayed PWM signal for various trigger slopes and voltage scales.

3 Summary and Conclusion

The lab accomplished several things, namely the construction of a real-time application using timing constraints, interrupts, and shared data patterns. Interrupt Service Routines were constructed for the peripherals, and different data structures were used for read and write operations. For an oscilloscope with greater detail, more complex data structures could be used to process all the raw ADC data samples. Despite the difficulties, a digital oscilloscope with a voltage range and trigger slope that can be toggled was implemented on the board.

4 Appendices

4.1 Appendix A

Rising edge trigger search function implemented in the digital oscilloscope. The algorithm starts near the end of the gADCBuffer and moves backwards until it detects the trigger level in the desired direction.

```
int RisingTrigger(void)
{
    // Step 1
    int x = gADCBufferIndex - LCD_HORIZONTAL_MAX / 2; // Half screen width; don't use a magic number

    // Step 2
    int x_stop = x - ADC_BUFFER_SIZE/2; // Half the ADC buffer
    for (; x > x_stop; x--) {
        if (    gADCBuffer[ADC_BUFFER_WRAP(x)] >= ADC_OFFSET &&
                gADCBuffer[ADC_BUFFER_WRAP(x-1)]< ADC_OFFSET) // Older sample
            break;
    }
    // Step 3
    if (x == x_stop) // for loop ran to the end
        x = gADCBufferIndex - LCD_HORIZONTAL_MAX / 2; // reset x back to how it was initialized
    return x;
}
```

4.2 Appendix B

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

```
for (i = 0; i < LCD_HORIZONTAL_MAX - 1; i++)
{
    // Copy waveform into local buffer
     sample[i] = gADCBuffer[ADC_BUFFER_WRAP(trigger - LCD_HORIZONTAL_MAX / 2 + i)];

    // draw lines
    y = LCD_VERTICAL_MAX / 2 - (int)roundf(fScale * ((int)sample[i] - ADC_OFFSET));
    GrLineDraw(&sContext, i, y_prev, i + 1, y);
    y_prev = y;
}
```

4.3 Appendix C

Timer3 configuration that configures a timer in one-shot mode and with a timeout interval of 10 ms.

```
//Initialize CPU Load Timer
SysCtlPeripheralEnable(SYSCTL_PERIPH_TIMER3);
TimerDisable(TIMER3_BASE, TIMER_BOTH);
TimerConfigure(TIMER3_BASE, TIMER_CFG_ONE_SHOT);
TimerLoadSet(TIMER3_BASE, TIMER_A, gSystemClock/100 - 1); // 10 ms
```