Introduction

In this lab assignment we implemented a 1 Msps digital oscilloscope using our lab kit, and the TI EK-TM4C1294XL launchpad and the BOOSTLX-EDUMKII. This lab consisted of collecting 1,000,000 samples per second from the ADC into a buffer, adjusting the waveform and displaying it on launch pad screen. The screen also displayed the time scale, the voltage scale, the tigger slope, the trigger level, and the cpu load of the system. We displayed the triggered value in the center of the screen. We also allowed the user to modify the voltage scale and the trigger up/down setting through button input which used a FIFO buffer, to ensure that a button presses do not get missed. The CPU load of oscilloscope program was calculated and displayed on the screen as well. Mainly for this lab we made an oscillator.

Discussion

Overview

The intended design of this project was as follows.

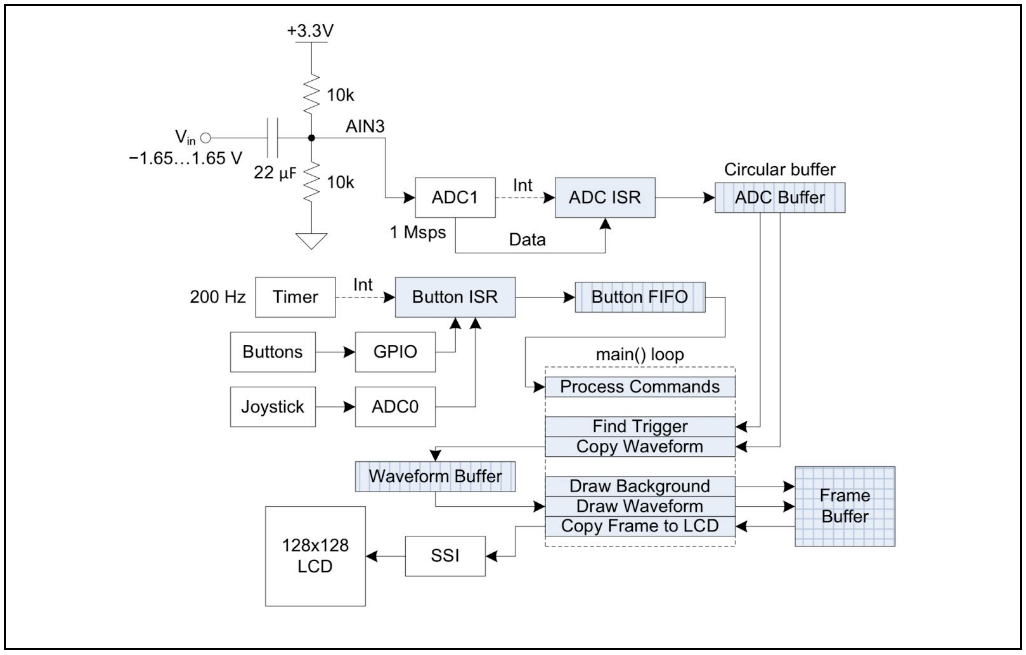

Figure 1. Layout of the components' interactions.

We create a oscillating source described by the Figure 2 that is capacitively coupled into a voltage divider seen in Figure 1.

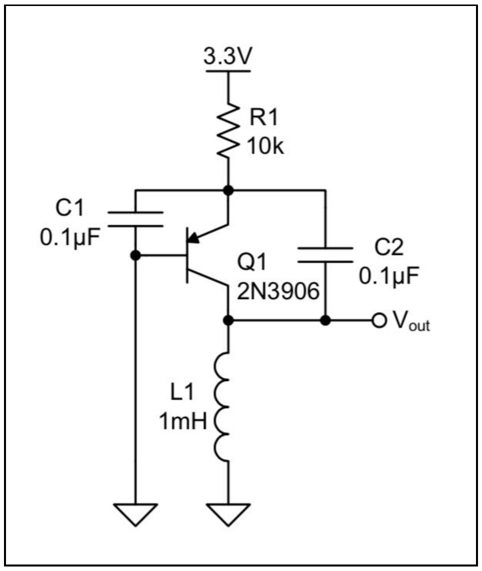

Figure 2. The oscillator circuit.

This circuit creates a sine wave at ~22.5 kHz with an amplitude of ~0.5v, this circuit is then capacitively coupled into a voltage divider, this moves the waveform from being centered around 0, to now being centered around 1.65 volts, this is important, because the onboard ADC only works on values from 0 to 3.3v, and values less than 0 will be interpreted as 0, this offset will be compensated in code, by setting that 1.65v value to 0. The sine wave is then sampled by the by the ADC at 1 Mbps. The ADC is continuously converting analog voltage into digital values, and is running as the highest priority interrupt. It is important as not to miss any values coming in from the signal and this ensures we don't.

ADC Sampling

The ADC generates a hardware interrupt setting a flag. The interrupt is then handled by our ADC_ISR() function. This function clears the flag set by the interrupt, verifies that we did not miss any deadlines from the ADC, and copies the value from the ADC register into our circular ADC buffer `gADCBuffer`. This ISR is set with a priority of 0, meaning that is the highest priority, and can interrupt other interrupts to make sure that this deadline is met. It also means that no other interrupts can interrupt it. These values are constantly being written to the

ADC FIFO, no other tasks or interrupts are writing to this FIFO to guarantee no shared data bugs. These buffer values are then used to generate our waveform.

Waveform generation

Now that the values are being sampled and stored into our buffer `gADCbuffer`, and we can guarantee that there are no missed deadlines, as well as no shared data bugs, we need to be able to read that values from the buffer and display the waveform on the screen. We can not read the values directly from `gADCbuffer` and display them since the buffer is getting written at 1,000,000 samples per second and this is too fast for our program to read all the values without being overwritten before finishing one loop. Also, writing to screen is relatively slow, so by the time that we try to write all 128 values to screen, the buffer will have looped, and our wave will not be continuous. In order to display the correct waveform on the screen, and make sure that we don't miss any deadlines we will search for the value we are trying to trigger on, and once that value is found, we will take 64 samples behind that value, and 64 samples ahead of that value from `gADCbuffer`, and copy them into a local buffer to be displayed. Our local buffer was named `locoADCBuffer`, and holds 128 values, which the pixel width of the screen.

The first thing that we need to do is find a trigger. For this lab we selected the trigger value to always be the center of the waveform, which after going through the voltage divider, is the ADC value that corresponds to 1.65 volts. We measured this value to be 2044. In theory, it should theoretically be 2048 since that is half of the maximum value of the ADC, however there are some internal sources of error due to the conversion between 0-3.3V and 0-4096, as well as the resistor values for our voltage divider not being perfect.

In `main` we call a function `get_waveform()`. In this function the first thing that we do is check to see if a wave form has already been found with the integer `triggered`, while `triggered` is 0 we will acquire a wave form, and once a waveform is acquired we will set `triggered` equal to 1 to print it. After the wave form is displayed we will set `triggered` back to 0 so that a new waveform can be acquired. Inside the while loop, we define triggered to be 64 values behind the head of the `gADCBuffer` a half screen behind, this is so that if the first value checked happens to be the `triggered` value we have enough frames in front of us to be able to print the waveform, if a waveform is not found we then traverse back half the ADC buffer in search of our triggered value using a for loop, if the value still is not found, we go back 64 samples behind the head and try again until we find one.

We are doing all of this in `main()` which is our lowest priority task, therefore this task is often interrupted by our other tasks like the ADC and button ISRs. Due to this, we only traverse back half of the buffer size once we find a satisfactory trigger value so as to not interfere with the forward values of the `gADCBuffer` while it is being written to. Once the value we are triggering on is found, we set the `triggered` boolean to one, and copy 64 samples behind the value we found, then the value we found, then 63 values in front of so that we can display the standing wave on the screen. After this is copied into a local buffer we check the trigger value against the next value in the buffer to see if this is the rising side of a waveform or the falling side. Based on user settings, we then determine if this waveform will be displayed on the screen or not.

Screen generation

When we draw the screen this takes a couple of components. The main consideration here is that we want to make sure we layer the components correctly so that the final product is showing us what we want to see.

The first thing we draw is the grid. This consists of drawing horizontal and vertical lines spaced 20 pixels apart with the middle ones lining up in the center of the screen. To ensure that the center crossbar is in the center we calculated that the first line needed to be offset by 4 pixels. Once we've drawn the grid, we want to amplify the center crosshairs so we draw those as a lighter blue so they stand out.

We will now populate the screen with information about the waveform on the upper and lower bars. We first draw a black bar at both the top and the bottom. For the bar at the top we then generate it with the timescale, the voltage scale and the current trigger direction. These values are all taken from global variables because the user can modify them over the course of running the program, so when the user does modify them, we are automatically getting the changes by accessing the global variable. These variables are the

```
// the direction of the trigger
int trigger_direction = 1;
```

```
// voltage possibilities
float voltages[4] = {.1, .2, .5, 1};
int voltage_index = 1;
const char * const voltage_strs[] = {
                                     "100 mV", "200 mV", "500
mV", "  1  V"
};
```

Code Snippet 1. Global variables used for determination of the direction to trigger and the voltage.

The lower bar is then generated with the CPU load. We calculate the CPU load by counting the loaded and unloaded values over an interval of 10 milliseconds and converting it to a float and multiplying it by 100. Then we display this on the bottom of the screen as a percentage with the '%' appended to the end.

The last component to draw is the waveform. We draw this by allowing each sample to be 1px wide, we draw rectangles for each sample to make a continuous line. To scale the y values correctly we used the function that was given to us in lab.

```
int y = LCD_VERTICAL_MAX/2 - (int)roundf(fScale * ((int)sample -
ADC_OFFSET));
```

Code Snippet 2. Calculation of the y-value for the screen display.

Where `LCD_VERTICAL_MAX` is the 128px, `sample` is the current sample value from the local buffer, and `ADC_OFFSET` was measured to be 2044. `fScale` is another expression that was given to us in the lab.

```
float fScale = (VIN_RANGE * PIXELS_PER_DIV)/((1 << ADC_BITS) *
voltages[voltage_index]);
```

Code Snippet 3. Fscale calculation.

`VIN_RANGE` = total Vin range in volts = 3.3, `PIXELS_PER_DIV` = LCD pixels per voltage division = 20, `ADC_BITS` = 4096, because the ADC uses 12 bits, and the 2^12 is 4096, and `float voltages[voltage_index]`= volts per division setting of the oscilloscope, which was varied from 100mV to 1V, defined by the user. The variable `voltage_index` is modified based on the users input and `voltages` is a statically defined array of the possible voltages stored as float values.

These two functions allowed us to accurately scale the data from our local buffer to be display the points properly on the screen. To make the points appear to be a continuous line, we stored the previous point, and averaged it with the current point to make the lower bound of that box, and we took the same average value and added it to the current y to set the bound for the top of the box, this gave averaging give us a continuous sinusoidal wave.

User modification

To allow users to modify the trigger value and the voltage scale we used the joystick and two user buttons on the educational booster pack board. To initialize the buttons for use we took concepts we learned from lab 0 and applied them again here. This specifically involved using the GPIO pins to setup the button presses for the right pins on the right port. We used ADC0 for the configuration of these. The debouncing was done by reading the button state over time to see how long the button was held then anding it with a mask. The priority for the button ISR was set to 32 because it is not as important as the ADC sampling, but more important than our `main()` function.

After they were configured we could use the buttons to modify the global variables we defined directly, shown in Code Snippet 1. However, since drawing on the screen and collecting samples takes so long, it's hard to make sure that the button presses are going to get used when they are pressed. To ensure that none of the button presses were missed we used a FIFO buffer. This is a buffer of ints where each int represents a different button / joystick trigger. The global values we used in our buffer are described below.

```
#define TRIGGER_UP      1
#define TRIGGER_DOWN    2
#define INCREASE_VOLT   3
#define DECREASE_VOLT   4

int fifo[FIFO_SIZE];  // FIFO_SIZE = 11
int fifo_head = 0; // index of the first item
int fifo_tail = 0; // index one step past the last item
```

Code Snippet 4. The variables used for the FIFO button buffer.

Using these predefined values as inputs and outputs from our buffer ensured we weren't using random values in our buffer that would then have to be decoded later.

We wait until after we draw the screen to check our button buffer to make sure that the user hasn't pressed any new buttons while we were drawing and calculating a new waveform. If there was a change in the buffer we handle that change by changing the associated variables that are used when drawing the screen. For example, if the user wants the waveform to be triggered on the rising edge we change the variable `triggered_direction`. Then, on the next draw of the screen, which will happen once we find a clean waveform that is triggering up, the change will take place in both the waveform view and the upper corner where the trigger up symbol is drawn.

If the user had changed the voltage scale then we would first make sure that they would still be within the bounds that we can show (100mV, 200mV, 500mV, 1V). If they are, we modify the the index associated with the global array `voltages` and increase or decrease it. The waveform draw function handles the connection of the pixels to make for a smoother waveform.

Conclusion

The most important outcome of this lab was to be able to create a working oscilloscope by implementing features necessary in real-time systems. By setting our ISRs to have priorities and handling shared data problems, we learned the concepts that are important when implementing real time operating systems. We set our task priorities so that the highest priority was the ADC collection because that was the most integral to the RTOS. Our mid level priority was registering user button presses, and our lowest level was the displaying of the waveform. We handled shared data by implementing FIFO buffers that we implemented for our button presses, and our ADC which accounts for re-entrancy in functions that access it. These concepts are important for real time operating systems and ultimately, this lab helped us learn how to implement these while creating an very functional 1Mbps oscilloscope.