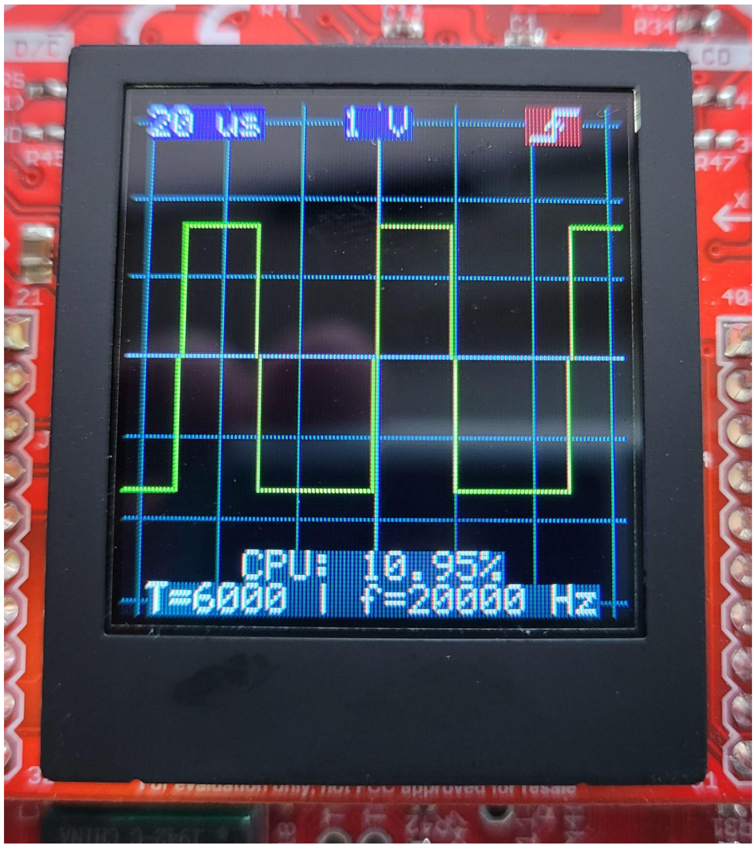
Slater Campbell
ECE3849 Lab 3
5/13/2021

Objective:

The goal of this lab was to take our existing oscilloscope code built in the TI-RTOS, and optimize it using Direct Memory Access. Then, using the CPU overhead we gained, we had to configure a timer to measure the period of the input wave. We also had to use PWM with an adjustable duty cycle to generate an audio waveform with the buzzer. I was successful in implementing all of these changes.



Discussion and Results:

The first thing I did was port my CPU usage measurement code from Lab 1. I left it unchanged, except I moved the loaded measurements to the display task, since it is the lowest priority task. This way, the counting will be interrupted by all the higher priority tasks, and the measurement will represent the loaded result. With the only change being adding CPU usage measurement and display code, I measured 67%-70% utilization. (Full measurements in Table 1 below).

Next, I implemented the DMA as an optimization for the ADC ISR. I opted for conditional compilation: if “DMA” is defined, then it will use the DMA mode, otherwise it will perform like labs one and two. I added the DMA initialization code to the same function that initializes the ADC. If

Slater Campbell
ECE3849 Lab 3
5/13/2021

DMA is not defined, then ADCIntEnable() is compiled. If DMA is defined, then ADCSequenceDMAEnable() and ADCIntEnableEx(ADC1_BASE, ADC_INT_DMA_SS0) are both compiled. I added the DMA ISR to the ADC ISR code, also conditionally compiled. When DMA is defined, the ADC ISR will clear the interrupt flag ADC_INT_DMA_SS0 in ADC1_BASE. Then, I completed the primary and alternate channel reset code. The argument for uDMAChannelTransferSet() is the same as in the setup code, so I copied it from there. For the alternate channel, I replaced UDMA_PRI_SELECT with UDMA_ALT_SELECT, and the start address of (void*)&gADCBuffer[0] got replaced with (void*)&gADCBuffer[ADC_BUFFER_SIZE/2], to signify that we are starting halfway through the 2048 sample buffer rather than at the start. Then, I changed the priority of the ADC_Hwi in the RTOS config to be 64. This means it is no longer a zero latency interrupt, which is okay because it has a significantly longer relative deadline. After running the code, I came up with the CPU usage measurements shown below in Table 1.

Table 1

ADC Configuration	Sampling Rate	CPU Load	ISR Relative Deadline
Single Sample ISR	1 Msps	67-70% fluctuating	1 usec
DMA	1 Msps	1.07%-1.13%	1024 usec
DMA	2 Msps	1.66%-1.73%	512 usec

Calculating the relative deadline for each configuration is simple. For the single sample mode, it interrupts every sample. At 1 million samples per second, this translates to a deadline of 1 usec. For the DMA modes, the interrupt is called when one of the two 1024 element buffers fill up. At 1 million samples per second, or 1 sample per microsecond, this will take 1024 microseconds. At 2 million samples per second, this will take 512 microseconds.

The CPU usage results I got make sense. The interrupt is called 1/1024th as many times, but when the interrupt is called, it takes longer to complete than before. Also, this change only affects the ADC ISR, so other parts of the system would use just as much CPU time. Thus, the CPU usage is not exactly 70%/1024. When we halve the DMA ISR deadline, the CPU usage increases proportionally, when we also consider the above factors.

Next, I configured Timer0 to be used for measuring the period of the input wave. For initializing the timer, I used the starter code and filled out the missing pieces. For the GPIOPinConfigure command, I searched through the header file and found GPIO_PD0_T0CCP0, referring to Port D, pin 0, timer input T0CCP0. I configured the timer to raise an event on the positive edge only, which means the value returned will be the full period of the wave. I configured the timer to interrupt on TIMER_CAPA_EVENT, which refers to the timer captureA event. Then, I designed an ISR for Timer0. The ISR is configured as a zero latency Hwi, with priority 32. The first thing the ISR does is clear the interrupt that called it, which is the same TIMER_CAPA_EVENT. The interrupt initializes a static variable at 0, which will hold the previous timer value. Then, it reads TimerA from TIMER0_BASE using TimerValueGet(). The period is then calculated by subtracting the previous value from what was just read, and bitwise ANDing the result with 0xFFFFFF. This resolves rollover conditions. This period is then saved to a global value, and the prevTimerValue static variable is updated with the last timer read.

The period returned by the Timer0_ISR is a count of 120MHz clock ticks. In a function called by my displayTask, I convert the period to a frequency in Hz. This frequency is displayed on the LCD, along with the current requested period from the PWM signal generator. This period is adjustable by moving the joystick left and right. I added a global that stores the desired period. When the joystick input is detected, it adjusts that period by 10 and calls a function which updates the PWM period by calling PWMGenPeriodSet(). It supplies the arguments PWM0_BASE, PWM_GEN_1, and the period value.

Next, I set up the PWM generator for audio output. I imported the audio waveform files. Then, using the square wave signal generator code, I changed the GPIO configuration to use port G pin 1. The peripheral is still PWM0, but the PWM generator is now generator 2. Generator 2 is used for outputs 4 and 5, and output 5 is the one which I am using for the buzzer. I configured the period to be 258 clock cycles as specified, and the 50% duty cycle is produced by halving that period to set the pulse width. I configured it to count down, interrupting when the count reaches zero. Next, I made the PWM_ISR. This ISR is configured as a zero latency Hwi, with priority 0. This makes it the highest priority interrupt in the system. I filled in the ISR code, first by setting the PWMGenIntClear() to clear PWM_INT_CNT_ZERO from PWM_GEN_2 in PWM0_BASE. I set it to disable the interrupt when index i became greater than gWaveFormSize. This would signify that all elements had been played. In my userInputTask, I configured User Button 2 to call PWMIntEnable(PWM0_BASE, PWM_INT_GEN_2). This enables interrupts in generator 2, which will cause audio to start playing.

Conclusion

This lab was more challenging than the others, as it involved implementing more features and required more thought about how each one worked in order to implement it properly. I really enjoyed it! Once I took the time to understand how everything worked, it became relatively simple. It forced me to think deeply and I found that very valuable. One issue I ran into was that DMA breaks the FFT functions of the lab. I think this is because the index that the DMA is writing to is not always known exactly. Since the FFT needs to use 1024 elements of the buffer, it can end up reading outside of the half of the DMA buffer that it should. Since the FFT functioning was not ever mentioned in this lab, I'm assuming that this is expected. One improvement to the lab would be to address this. Unless, of course, this bug is specific to me. In either case, I want to figure out how to solve it!