Lab Assignment 2

Interrupts and PWM

Luke Jaffe
jaffe.lu@husky.neu.edu

Brian Crafton
crafton.b@husky.neu.edu

Submit Date: 02/16/2016
Due Date: 02/16/2016

Instructor: Prof. Gunar Schirner
TA: Jinghan Zhang

Abstract

The purpose of Lab 2 was to take more efficient approaches to interfacing with the real world, and to investigate different methods of timing control. In this lab, concepts include interrupts as a more efficient way to interface with the real world, and different forms of timing control in the form of busy delay loops and timers/PWM. This assignment uses concepts from the previous assignment but improves on efficiency and explores new ideas.

Introduction

In Lab 1 software polling was used to check the states of buttons and switches and to then use their states as input to control LEDs (outputs). This is inefficient because it requires the processor to constantly poll inputs waiting for something to happen and if the processor isn't reading at the time of input then the input is lost. Fortunately external hardware can be used to catch these inputs and notify the processor when input has occurred. This technique is called an interrupt and it was used in Lab 2 as a more efficient method for allowing the processor to interact with the real world. Interrupts are more efficient because they do not require the CPU to constantly be polling the status of the inputs.

It is often the case in embedded systems that there needs to be a time delay before executing future instructions. In Lab 1 delay loops were used to add time delay in toggling LEDs. This is not efficient because it is blocking the processor for doing anything else until the number of ticks specified have occurred. This time could be spent doing other useful tasks but it will be wasted because the processor has to keep track of the delay itself. The first solution that was tried in Lab 2 was a busy delay loop. The idea was to calculate the number of instructions that would be necessary to perform the busy delay loop and use the input clock speed of the timer to delay for a certain amount of time. This calculation results in:

(1 / clock speed) * (# instructions in iteration) * iterations = time delay

This method works, but once again is inefficient because it requires the processor to constantly be keeping track of time. Also if another task is running concurrently the delay will not be accurate because it assumes the processor is using all its processing power on it. This method was also used to generate a PWM signal. All that was required to do this was to set the output low then delay for 18.5 seconds and set it high and delay for 1.5 seconds. The other method used to do this was a timer. A timer is a binary counter which will interrupt the processor when the specified number of ticks have occurred. This way the processor does not need to keep track of time itself and can do other useful tasks and react later. In this lab the timer was not even configured to interrupt the processor it simply updated the output signal without requiring the processor.

Lab Materials

- Hardware
  - Xilinx Zynq®-7000 All Programmable SoC
- Software
  - FreeRTOS
- Environment
  - Eclipse IDE
  - Xilinx SDK

Results and Analysis

I. GPIO Interrupts

To create a functional interrupt handler, three steps were necessary. First, the handler reads interrupt status by checking the mask register (to check which bits are enabled) and the status register (to check which bits are currently set):

```
u32 int_assert =
(*(volatile u32 *)GPIO_INT_STAT_2) & ~(*(volatile u32 *)GPIO_INT_MASK_2);
```

Then, the handler clears all set interrupts by setting the requisite status bits high:

```
(*(volatile u32 *)GPIO_INT_STAT_2) = int_assert;
```

Finally, the handler implements the necessary output logic by setting the LED bits equal to the switch bits in the data register. This logic should be implemented more safely in a more complex project, but this code is sufficient to solve the problem:

```
(*(volatile u32 *)GPIO_DATA_2) = ((*(volatile u32 *)GPIO_DATA_2)>>16) &
0x1F;
```

II. GPIO PWM

To create a software delay loop, we did calculations based on the number of instructions executed in the loop body of the busyWait() function:

```
static inline void busyWait(unsigned long delayCount)
{
      while(delayCount--)
      {
            ;/* Do Nothing */
      }
}
```

Using the clock speed, we determined how many loop iterations would create a 7.5% duty cycle for a
20ms period. We used the following values for the on and off portions of the wait:

```
#define ON_DELAY_COUNT (200100)
#define OFF_DELAY_COUNT (2467900)
```

One loop iteration was implemented as follows:

```
(*(volatile u32 *)GPIO_DATA_2) = (0x01<<21);
busyWait(ON_DELAY_COUNT_ADJ);
(*(volatile u32 *)GPIO_DATA_2) = (0x00);
```

```
busyWait(OFF_DELAY_COUNT_ADJ);
```

When we checked the PWM waveform produced by our calculated values on an oscilloscope, we found that the measured result differed slightly from our intended result. We adjusted our delay counts by multiplying the original values by a factor corresponding to the measured length vs. the target length. This yielded the following values:

```
#define ON_DELAY_COUNT_ADJ (166750)
#define OFF_DELAY_COUNT_ADJ (2056583)
```

When we ran the program using these values, the waveform had almost exactly 7.5% duty cycle and 20ms period.

When we enabled O2 level compiler optimization and checked the oscilloscope, the waveform appeared as a solid block, implying instantaneous oscillation. When we viewed the assembly code for the program and tried to find the busyWait() function, it did not appear in the code at all; it had been optimized out. This means that the code had been stuck in a while(true) loop with set_high(), and set_low() functions with no delay inbetween, which was consistent with what we saw on the oscilloscope.

III. GPIO TTC

First, we configured TTC0, one of the triple-timer counters, by setting the necessary registers. The clock control register was configured to enable the prescaler and set its value. The counter control register was configured to determine waveform polarity (low to high on match interrupt), setup the match register, and generate interrupts on regular intervals. The interval register was set to create a 20ms period, given the clock speed and prescaler. The Match register was set to give a 7.5% duty cycle, given the period.

```
//Set Clock Control 7bits, Enable Prescaler(Bit0) and give the Prescaler
Value(Bit4:0)
ClockCntl =  0x0B;
*((volatile u32*)(SET_CLK_CNTRL_VAL)) = ClockCntl;

/* Set Counter Control 7bits, Wave_pol(Bit6), Wave_en(Bit5), RST(Bit4),
Match(Bit3), DEC(Bit2), INT(Bit1), DIS(Bit0) */
CounterCntl = 0x4A;
*((volatile u32*)(SET_CNT_CNTRL_VAL)) = CounterCntl;

// Set 20ms Period count number
Period = 34688;
*((volatile u32*)(SET_INTERVAL_VAL)) = Period;

// Set 7.5% Duty Cycle count number
Duty_Cycle = 2602;
*((volatile u32*)(SET_MATCH_0_VAL)) = Duty_Cycle;
```

When we used an oscilloscope to verify our configuration, the results were as expected. Further, when the code was compiled with O2 optimization, no difference was present in the waveform. This demonstrates that hardware counters are not affected by software optimizations, since no optimization would interfere with setting/clearing of registers.

Finally, the hardware timer code was combined with our previous work with interrupts to produce a program for controlling a servo motor with push buttons. The details of this portion are discussed in post-lab question 3 below.

Post-Lab Questions

1. Interrupts allow the CPU to do more tasks rather than poll inputs, and are more power efficient because they do not require lots of polling time to catch inputs. Interrupts do have the downside of requiring additional hardware and needing to copy registers prior to jumping to the ISR function.

2. Delays created by loops rely on variables. This severely limits the usefulness of delay loops as a technique for time control. If the clock speed is changed, the assembly code for the loop block, or different tasks are running at once, then the actual delay time will be affected greatly. Because the technique relies on a constant instruction execution rate, if the speed of the processor changes or it is not the only task requiring processor time, it will take longer than expected. The main effect that changed in the lab was the delay code block. The compiler removed the delay function because it did not actually do anything and for this reason it was never delaying at all, it was just repeatedly toggling the output and never delaying.
A more portable way for implementing fixed time delays is to use a timer. Have the delay function that takes three parameters.

1. Clock speed of the timer.
2. Amount of time to delay.
3. A function to call back when the time has expired.

The correct number of ticks to wait will be determined by the timers speed and the amount of time to delay. The callback function will run the code that the user wanted to delay. Once the timer expires the ISR function for the timer interrupt used will be configured to call the callback function and execute the user's code.

3.
The FSM constructed to govern servo motor arm movement was created using simple if/else statements and state variables. Our interrupt handling function was updated to check for each of the three needed buttons (up, down, center), and execute the necessary response:

```
if (int_assert & (0x01<<20))
{
      pwm_shift(1);
}
if (int_assert & (0x01<<18))
{
      pwm_shift(0);
}
if (int_assert & (0x01<<17))
{
      pwm_reset();
}
```

The pwm_shift function was used to move the servo arm back and forth by changing the PWM duty cycle. The servo arm was moved in increments such that a full swing from its leftmost to its rightmost position would be composed of 15 steps. The control code below implements a FSM with three states: normal, minimum, and maximum. The current duty cycle is checked against the min and max, then a response is executed, which may include incrementing/decrementing the duty cycle, and turning on/off LEDs (to indicate system state to the user).

```
void pwm_shift(unsigned int dir)
{
      if (dir && Duty_Cycle < DUTY_CYCLE_MAX)
      {
            Duty_Cycle += DUTY_CYCLE_INC;
            *((volatile u32*)(SET_MATCH_0_VAL)) = Duty_Cycle;
            (*(volatile u32 *)GPIO_DATA_2) = 0x00000000;
      }
      else if (dir && Duty_Cycle >= DUTY_CYCLE_MAX)
      {
            (*(volatile u32 *)GPIO_DATA_2) = (0x0C);
      }
      else if (!dir && Duty_Cycle > DUTY_CYCLE_MIN)
      {
            Duty_Cycle -= DUTY_CYCLE_INC;
            *((volatile u32*)(SET_MATCH_0_VAL)) = Duty_Cycle;
            (*(volatile u32 *)GPIO_DATA_2) = 0x00000000;
      }
      else if (!dir && Duty_Cycle <= DUTY_CYCLE_MIN)
      {
            (*(volatile u32 *)GPIO_DATA_2) = (0x03);
      }
}
```

In order to transition out of the maximum or minimum states, the reset button (center button) must be pressed. Our implementation deviates slightly from the problem statement in the lab, and was approved by the TA. When the reset button is pressed, the duty cycle is set so the arm returns to neutral position, meaning the FSM has returned to the “normal” state.

```
void pwm_reset()
{
	*((volatile u32*)(SET_MATCH_0_VAL)) = DUTY_CYCLE_NEUTRAL;
	Duty_Cycle = DUTY_CYCLE_NEUTRAL;
	(*(volatile u32 *)GPIO_DATA_2) = 0x00000000;
}
```

Note that no variables are used to explicitly store the state of the FSM, since this would be gratuitous.

Conclusion

In this lab the goal was to introduce new ways to handle time control requirements and processor interaction with the real world. In the first part of the lab, polling was replaced and interrupts proved to be very useful since the processor could do other things rather than constantly check for interrupts. Next busy delay loops were used to replace a task delay. At first the theoretical calculation necessary for an 18.5 millisecond delay and 1.5 millisecond delay were off. After simply finding the fraction off they were from the desired value they were divided by the fraction and came out to be correct. However this technique was still inefficient because it required constant use of the CPU otherwise the delay time would be off. This is because it requires the number of instructions executed per second to remain constant, which is not the case when there are multiple tasks running at once. The next technique used was a hardware timer. The timer was effective because it did not require the processor to keep track of time and updated the PWM output when it was time to do so.

After completing the lab it is clear that both interrupts and timers are valuable techniques in the embedded world. Both require minimal additional hardware and allow the processor to spend its time usefully rather than polling inputs or keeping track of time based on its clock cycles.

References

[1] Xilinx, Zynq-7000 All Programmable SoC Software Developers Guide, v12.0 09/30/2015
http://www.xilinx.com/support/documentation/user_guides/ug821-zynq-7000-swdev.pdf

[2] Xilinx, Zynq-7000 All Programmable SoC Technical Reference Manual, v1.10, 02/23/2015
http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf