Lab Assignment 3

Queues, Debounce, Multi-Channel PWM

Luke Jaffe
jaffe.lu@husky.neu.edu

Brian Crafton
crafton.b@husky.neu.edu

Submit Date: 02/29/2016
Due Date: 03/01/2016

Instructor: Prof. Gunar Schirner
TA: Jinghan Zhang

Abstract

The purpose of Lab 3 was to understand issues in embedded systems that require a real time operating system like freeRTOS. Embedded systems solve different problems than laptops and for this reason have different requirements and implementations. In order to be low power, deterministic, and running multiple tasks or complex applications, an OS suited to these needs is required.

Introduction

In lab 2.3 a counter was maintained by pushing an up button or a down button. However, debounce was not dealt with properly and more than not an extra count (in either direction) was done due to the debounce. The debounce that happens is caused because mechanical parts due not create the crisp transition that is ideal. In order to deal with this problem a delay was used. A delay cannot be used from an interrupt handler because it has the highest priority and the OS will not context switch from it. In order to have a delay a message needed to be sent to a queue to in the GPIO task. From the GPIO task a delay could be used, that would allow the OS to run a separate task and not waste clock cycles.

In lab 3 part 3, 2 tasks were used. In order to handle delays for both the debounce and the delay for the PWM it was necessary to separate the two into two separate tasks. This way the PWM could run and not be disturbed if the queue had a message saying to check the status of a push button. This displayed the need for an operating system because the application required running two real time tasks. In the last part of the lab priority was experimented with. When giving a 'disturb' task (waits 500ms and delays 500ms) higher priority than the GPIO task it interfered heavily with the performance of counter. This is because the GPIO task was not getting the chance to react to queue information for half a second every second.

Lab Materials

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

Results and Analysis

I. Interrupt Counter

This part of the lab focused on sending messages between tasks using a queue. We first declared a queue as follows:

```
gCountUpdateQ = xQueueCreate( 10, sizeof(int));
```

We then configured the GPIO to generate an interrupt when on the rising edge of a button push (up and down):

```
/* disable interrupts before configuring new ints */
*(volatile u32 *)GPIO_INT_DIS_2 = 0xFFFFFFFF;

*(volatile u32 *)GPIO_INT_TYPE_2 = (0x01<<18 | 0x01<<20);
*(volatile u32 *)GPIO_INT_POLARITY_2 = (0x01<<18 | 0x01<<20);
*(volatile u32 *)GPIO_INT_ANY_2 = 0x00000000;

/* enable input bits */
*(volatile u32 *)GPIO_INT_EN_2 = (0x01<<18 | 0x01<<20);
```

We also defined a counter in the interrupt handler as static, so that it would be local but persistent across calls:

```
static int counter = 0;
```

To read and clear interrupts, we used the same code as in the previous lab:

```
// read interrupt status
u32 int_assert = (*(volatile u32 *)GPIO_INT_STAT_2) & ~(*(volatile u32
*)GPIO_INT_MASK_2);

// clear interrupts
(*(volatile u32 *)GPIO_INT_STAT_2) = int_assert;
```

Then, to check the up and down buttons, we checked the asserted interrupts for high bits in the correct positions:

```
if (int_assert & (0x01<<20))
{
      counter++;
}
if (int_assert & (0x01<<18))
{
      counter--;
}
```

Finally, we sent a message to a separate task to print the counter value:

```
xQueueSendFromISR( gCountUpdateQ,( void * ) &counter, NULL);
```

The task contained a loop which would continually check poll the queue and print upon receiving a message:

```
for(;;)
{
	/* Receive the counter value from Queue */
	int value;
	if( xQueueReceive( gCountUpdateQ, &value, 1000) )
	{
		printf("Value is %d \n", value);
	}
}
```

The counter did not always increase by exactly 1 each time a button was pushed. Sometimes the interrupt would be received multiple times, indicating a bounce in the signal. Further, an interrupt for a rising edge would also often be received when the button was released. This indicated a debounce, which is resolved in the next part of the lab.

II. Interrupt Counter Debounced

We originally completed this portion of the lab using xTaskGetTickCountFromISR(), and comparing the time against the last time an interrupt occurred. For solving the bounce issue, this worked for just 2 ticks (10-20 ms). However, this did not solve the debounce issue, which would require a different approach.

For handling bounce and debounce, our interrupt handler would check and clear interrupts, send a message to a task containing the asserted interrupts, and then disable interrupts:

```
// read interrupt status
u32 int_assert = (*(volatile u32 *)GPIO_INT_STAT_2) & ~(*(volatile u32
*)GPIO_INT_MASK_2);

// clear interrupts
(*(volatile u32 *)GPIO_INT_STAT_2) = int_assert;

xQueueSendFromISR( gCountUpdateQ,( void * ) &int_assert, NULL);

*(volatile u32 *)GPIO_INT_DIS_2 = 0xFFFFFFFF;
```

In the task, the queue would be checked in a loop, and a task delay would be called if anything was received in the queue. Then, if the value in the gpio data register matched the value in the previously asserted interrupts, the counter value would be changed. Interrupts were re-enabled after checking both buttons:

```
for(;;)
{
	u32 value;
	/* Receive the counter value from Queue */
	if( xQueueReceive( gCountUpdateQ, &value, 1000) )
	{
		vTaskDelay(2);
		u32 gpio_data = (*(volatile u32 *)GPIO_DATA_2);
		if ((gpio_data & (0x01<<20)) & (value & (0x01<<20)))
		{
			counter++;
			printf("%d\n", counter);
		}
		if ((gpio_data & (0x01<<18)) & (value & (0x01<<18)))
		{
			counter--;
			printf("%d\n", counter);
		}
		*(volatile u32 *)GPIO_INT_EN_2 = (0x01<<18 | 0x01<<20);
	}
}
```

This approach worked well for handling both bounce and debounce, and also successfully handled simultaneous button pushing.

III. Interrupt Counter LED

To not block upon receiving when the queue is empty, the xQueueReceive function was called with the xTicksToWait parameter set to 0:

```
xQueueReceive( gCountUpdateQ, &value, 0)
```

When a new value was received in the queue, after the delay and check from the previous section, the bit pattern would be calculated upon a true button push:

```
if( xQueueReceive( gCountUpdateQ, &value, 0) )
{
	vTaskDelay(2);
	u32 gpio_data = (*(volatile u32 *)GPIO_DATA_2);
	if ((gpio_data & (0x01<<20)) & (value & (0x01<<20)))
	{
		counter++;
		printf("%d\n", counter);
```

```
            getPWMChangeBitPattern(counter, patternArray);
      }
      if ((gpio_data & (0x01<<18)) & (value & (0x01<<18)))
      {
            counter--;
            printf("%d\n", counter);
            getPWMChangeBitPattern(counter, patternArray);
      }
      *(volatile u32 *)GPIO_INT_EN_2 = (0x01<<18 | 0x01<<20);
}
```

The actual code to produce the bit pattern was the same as that used in the prelab:

```
unsigned char shift(unsigned char val, int mag)
{
      if (mag < 0)
            return val >> abs(mag);
      else
            return val << mag;
}

void getPWMChangeBitPattern(unsigned int value, unsigned int
patternArray[4])
{
      int s = (value / 4) - 1;
      unsigned int p = 2 * (value % 4);
      unsigned int d0 = shift(((0b11111101 >> p) & 0b11), s);
      unsigned int d1 = shift(((0b10110101 >> p) & 0b11), s);
      unsigned int d2 = shift(((0b10000101 >> p) & 0b11), s);
      unsigned int d3 = shift(((0b00000001 >> p) & 0b11), s);
      // assume all LEDs are off at the beginning of a period
      unsigned int m1 = (~(d1 ^ d0) << 16) & 0xFFFF0000;
      unsigned int m2 = (~(d2 ^ d1) << 16) & 0xFFFF0000;
      unsigned int m3 = (~(d3 ^ d2) << 16) & 0xFFFF0000;
      patternArray[0] = d0;
      patternArray[1] = m1 | d1;
      patternArray[2] = m2 | d2;
      patternArray[3] = m3 | d3;
}
```

A separate task was implemented to update the LED values (though this likely was not necessary given that the queue receive was changed to be not blocking). This task contained a

4 step for loop within an infinite for loop. The bit pattern array it used to update was declared globally, so that it could be modified by the receive task and read by this task. The code was modified to use the MASK_DATA_2_LSW register, so that GPIO pins would only be modified on change.

```
static void led_task( void *pvParameters )
{
	for (;;)
	{
		int i;
		for (i = 0; i < 4; i++)
		{
			vTaskDelay(1);
			(*(volatile u32 *)MASK_DATA_2_LSW) = patternArray[i];
		}
	}
}
```

IV. Priority Check

The disturbTask was written to alternate between executing busyWait and vTaskDelay for 500 ms each in an infinite loop:

```
static void disturbTask( void *pvParameters )
{
	for(;;)
	{
		busyWait(BUSY_WAIT_COUNT);
		printf("DISTURB1\n");
		vTaskDelay(50);
		printf("DISTURB2\n");
	}
}
```

To increase/decrease the priority of a task, the uxPriority parameter of xTaskCreate was adjusted.

When the priority of the disturbTask was higher than that of the gpio_task, button events were lost, and the LEDs had 500ms gaps in displaying (during busyWait). This is because gpio_task and led_task were being pre-empted by the disturbTask to execute busyWait. When the priority of the disturbTask was lower than that of the gpio_task, the program functioned normally as in the previous section.

Bonus: Night Rider Scanner

To show the Night Rider scanner lights going left and right automatically, the gpio_task was modified to contain the following code in place of the loop checking the queue:

```
const int MIN = 0;
const int MAX = 32;
u32 dir = 1;
for(;;)
{
	vTaskDelay(4);

	if (dir && counter <= MAX)
		counter++;
	else if (dir && counter > MAX)
		dir = 0;

	if (!dir && counter >= MIN)
		counter--;
	else if (!dir && counter < MIN)
		dir = 1;

	printf("%d\n", counter);
	getPWMChangeBitPattern(counter, patternArray);
}
```

Simple if/else logic was used to check against the min and max values, and to increment or decrement the counter accordingly.

Conclusion

In this lab the goal was to investigate the advantages offered by a real time operating system in applications that require low operating power, multiple tasks, and deterministic behavior. Debounce was handled properly due to the use of message queues between tasks. This was done simply by firing a message to the GPIO task saying that a button event occurred and it needed to check to wait and see if it was in fact a button press or just debounce. Next two tasks were used in order to handle two functions that each needed to be handled with time constraints. Had these been implemented in the same task, they would have conflicted every time one called a task delay, because they were in fact two separate tasks. Lastly the problems that can occur with priority were investigated. When the function ‘disturb’ task had higher priority it caused lots of button events to be lost because the GPIO task did not have time to read and react to messages in its queue. The LEDs also did not duty cycle correctly since the led_task was being pre-empted for 500 ms periods. As soon as the priority for disturb task was lowered, GPIO task was able to perform as expected. After completing the lab it is clear that message queues and being able to run multiple tasks with time constraints is a necessary feature of a

real time operating systems. These features allow critical time constraints to be met and allow the processor to be more efficiently used.

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