Lab 4 Report

INPUT OUTPUT COMPARE

C. Ragona and J. Buss | EE 2361 | 3/18/2018

Documentation of Functions

There were quite a few functions created during this lab. Some of them are in different c files other then main. For the sake of the lab report I will just describe each one as if they were all in one c file.

SETUP()

This function simply sets our clock to 16 MHz, all pins to digital, it sets our RP8/9 pins to input and the rest to output, it starts RB5 to 0 for our game over LED, it unlocks/locks the pps, it enables the pins Rp8/9 to become input captures and it sets RP6 to be an Output compare for our servo.

_IC1INTERRUPT/IC2INTERRUPT

This function does a few things. First we set the global variable leftStart to 1. This is due to us wanting the game to start when both interrupts (or buttons) have been entered. These start variables are checked in the main. Next it initializes a: prevEdge to 0, currentEdge and count variables. It then moves onto setting this interrupt capture 1F flag back to 0 (since it changes to 1 once an input capture has occurred). It then moves to check if the current edge has a greater than 125 cycle count. This is a debouncing technique based off the initialized prescaler for the function of 256. Therefor if the button is giving an input capture reading less than 2ms we will not enter the main part of the function. The currentEdge is updated to with whatever value is in the ICxBUF. After entering the main part of the function the average (leftavg[] for ic1, rightavg[] for ic2) is put into the equation: $||currentEdge - prevEdge)| - (overFlow * PR2)|$. This equation takes the absolute value of the two edges and subtracts from the overflow calculation. Then we take the absolute value again. This is then put into an array which holds the time between presses called ****Avg[]. The rightAvg array is moved through using count which is updated a few lines later. The prevEdge is then updated to become the current edge and count is incremented by 1. Note count will not be higher than 4 as we have used a count &= 3. The next line initializes i for the for loop, and the global rightPersonScore to 0. We wanted to be sure we were always getting the correct score so we just cleared it before the next part of the function which calculated the average. The rightAvg array is then iterated through and added to the rightPersonScore. After they are added it is then divided by 4. This is because we are to take the last 4 button press averages. Overflow is then set to 0 and we exit our function.

_T1INTERRUPT

Here is where we start to figure out who is winning the game. This is setup to update every 1 second. However, this function will be called every 1 second due to how T1 is setup. First upon entering the _T1IF flag is set back to 0. A static count is then incremented by 1. The next statement says if count is 1000 enter the loop. Hence this will happen after 1 second. So once the 1 second check is entered some calculations take place. The formula we used to find out the winner was a percent-type based one: $\frac{difference\ of\ scores}{sum\ of\ \ scores} * 1000$. The 1000 comes from how the servo calculations work. The main point of this calculation is to find out how much we should move in either direction. Our servo pulse calculation states that at an OC1R value of 3000 the servo will be at 0 degrees, neither pointing left or right. So, what we did was take our calculation value and put it right into our setServo function. Note here that the if statement will check who is winning and swap the values in difference of scores depending on who has a smaller average. After the motor is updated a win variable is update for the winner by 1 and the loser win variable is set back to 0. Note in a tie the servo is set to its initial position of 3000 and both win variables are set to 0. Before exiting the 1000 count if statement the left score, right score and count are set back to 0. The scores are set back to 0 because we want a fresh new calculation each time. Also setting them back to 0 considers the fact if either player has not pressed a button in the last second, their score should be set to 0. Count is set to 0 to restart the 1 second count down.

_T2INTERRUPT

This function is really just set up to capture when the timer rolls over so we can update our overflows for our calculations. When the timer interrupt is enable we inter the function and we set our _T2IF flag back to 0. We then increment both overflow values. Exit function.

ENDGAME

This function is set to signify the end of game. First we set the _ICxIE to 0. We then set our servo to its initial position of 3000, or 0 degrees. The last item in the function is a while(1) loop that constantly sets our LED to an on state.

MAIN

There are a few things going on here. The timer 1 is set up to be 1 millisecond. The start variables are set to 0. The setup, button initialization and servo initialization are called here. The forever loop is entered, during each iteration of the loop we are checking if the left and right start variables are 1. This would only happen if both ic1/2 interrupt functions have been called. The _T1IE is set to 1 to start the enable flag to start checking the average and updating the servo. After both the start variables are 1 and the timer flag is set, the function then checks if either the left or right variables are 5. If they are we enter our end game function.

INITTWOPUSHBOTTONS

This is the function to initialize our timer, pull up resistors and input captures for our game. First we pull up the resistors in the corresponding RP8/9 pins. The timer 2 pre scaler is then set to 256 and pr2 to FFFF. This is because we want to maximize our time and minimize overflows. The timer interrupt flag is then set to 1. The ic1con is initialized and the buffer is cleared for each interrupt. Note that both interrupts are set to capture events on the falling edge.

INITSERVO

This function sets up the timer for use with output compare as well as initialize the output compare register components. First we initialize timer 3 to a prescaler of 8. Next we set the pr3 value to 39999 in order to get our 20 ms pulse. The output comapare OC1R is set to 3000 ms for an initial value. OC1RS is also set to 3000 as a placeholder. We also enable our output compare to use timer 3 and to use pulse width modulation while disabling OCf1.

SETSERVO

This function takes in an unsigned long int and sets the OC1RS value so we can update our pulse for our output compare to move the servo motor. The OC1RS value needs to be between 2000-4000 inclusive to keep our pulse within the range specified in the servo documentation. Anything from 2000-2999 is going to be between -75 degrees and 0 exclusive. Anything from 3001-4000 is going to be from 0 exclusive to 75 inclusive.

The Stages of Writing the Code

1. First the timer 3 calculations where figured out first.
2. This was figured out first to set up the output compare.
3. Next the calculation for the PWM was figured out to change the servo position.
4. Moving from this the button setup was created.
5. This included figuring out the timer 2 calculations and which bits to turn on for the input captures to set up for a falling edge.
6. We missed a step here which included setting the pull up resistors.
7. The next stage was creating the setup for the pic.
8. After that the interrupts were created and the calculation on how to take the average was made.
9. We stopped here as we were stuck and our interrupt wasn't working.
10. When then figured out that we needed to set our initial flag for our input capture interrupts.
11. After this was done we were able to see our interrupt working.
12. We then figured out that we needed to create a debouncing mechanism and within that we would execute our edge calculation.
13. This was followed with the idea of creating the circular buffer to store the averages along with updating the score at each interrupt.
14. We then moved on to creating the method to start the game.
15. After this we created the method to update the overflows.
16. During this time we used printf functions and the uart in simulation to figure out if we were getting the right numbers for our ic1 buffer.
17. Once this was figured out as true we moved on to creating the calculation for moving the servo to the winner's side.
18. There was a lot of debugging for this section as the motor was turning in wrong directions.
19. The was then debugged with printf again and was fixed with type casting.
20. We then created the end game mechanism.
21. This was coupled with trying to figure out when to update the scores.
22. We figured out we needed to use a timer to update our servo winner every second as well as we needed to add some sort of time out that would reset the servo when no buttons where pressed in the last second.
23. After that all everything begin to work and the game was finished.

Note: I left the print statements in the code tot show where we needed to debug it.

TEST CASES

There were a few test cases that we used for figuring out:

Checking if the left and right average interrupts where being placed in and if the calculation was correct:

```
//	printf("\n\nLeft Person Score: %d \n", leftPersonScore );
//	printf("Left Avg 0: %d \n", leftAvg[0] );
//	printf("Left Avg 1: %d \n", leftAvg[1] );
//	printf("Left Avg 2: %d \n", leftAvg[2] );
//	printf("Left Avg 3: %d \n", leftAvg[3] );
//
//	printf("\n\nrRight Person Score: %d \n", rightPersonScore );
//	printf("Right Avg 0: %d \n", rightAvg[0] );
//	printf("Right Avg 0: %d \n", rightAvg[1] );
//	printf("Right Avg 0: %d \n", rightAvg[2] );
//	printf("Right Avg 0: %d \n", rightAvg[3] );
```

Figuring out if we had the correct ICxBUF:

```
//printf("\n\nIC2BUF: %lu \n", IC2BUF );
rightAvg[count] = abs(abs(currentEdge - prevEdge) - (rightOverflow*PR2));
// printf("count: %d \n", count );
//printf("RightAvg[count]: %lu \n", rightAvg[count] );
```

Determining how to fix our calculations for determining the servo position:

```
//printf("\nleft score: %lu", leftPersonScore);

//printf("\nright score: %lu", rightPersonScore);

double denom = (double)(leftPersonScore + rightPersonScore);

//printf("\ndenom: %.2f", denom);

//printf("\nleft win calc: %d", (int)calculation);
```

Testing the servo:

We pressed one button faster then the other and vice versa to check our calculation

Observation of Performance

After the game was done and built there were not any type of errors or anomalies. This was a very great lab to piece together on how to debug while figuring out the components of the lab along the way.

However, there were errors while creating the game. While the calculation for setting the servo was correct the numbers being pass to the setServo function were not being passed correctly. The servo would turn it various wrong directions and was not being put in the intended test positions. Also our interrupt was not working at first, but we figured out we needed to enable the pull up resistors and interrupt the input capture interrupt flags.

The game works out well. The update every 1 second is smooth. The game timeout function was simple to write and works well with the aesthetic of the game. So the game starts after both buttons are pressed. If no button of either side is pressed the game will timeout back to the servo initial position. The game is still running, or in a play state. Testing one side compared the other to be the winner worked out well. The game enters the 5 second on one side win state rather smoothly.

All in all I think the lab was a rather success and it taught us a lot on how to work output compare and input captures events.