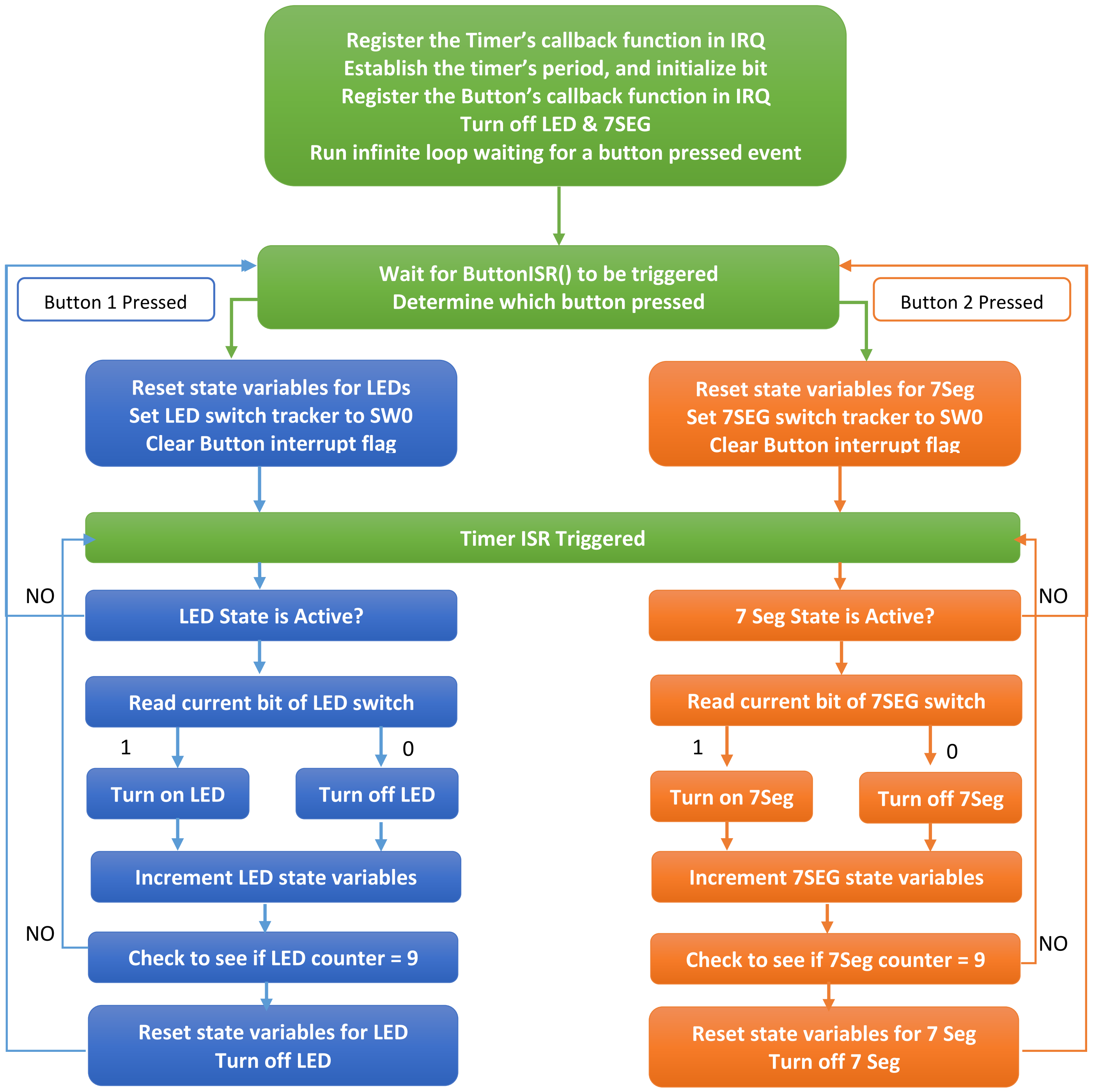
ECE 224: LAB 1

Ramie Raufdeen

Joey Keum

Phase 1

Overview of software design


Register the Timer's callback function in IRQ
Establish the timer's period, and initialize bit
Register the Button's callback function in IRQ
Turn off LED & 7SEG
Run infinite loop waiting for a button pressed event
Wait for ButtonISR() to be triggered
Determine which button pressed
Button 1 Pressed
Button 2 Pressed
Reset state variables for LEDs
Set LED switch tracker to SW0
Clear Button interrupt flag
Reset state variables for 7Seg
Set 7SEG switch tracker to SW0
Clear Button interrupt flag
Timer ISR Triggered
NO
LED State is Active?
7 Seg State is Active?
NO
Read current bit of LED switch
Read current bit of 7SEG switch
1
0
1
0
Turn on LED
Turn off LED
Turn on 7Seg
Turn off 7Seg
Increment LED state variables
Increment 7SEG state variables
NO
Check to see if LED counter = 9
Check to see if 7Seg counter = 9
NO
Reset state variables for LED
Turn off LED
Reset state variables for 7 Seg
Turn off 7 Seg

```
//State Variables
alt_u8 currLED = 0x00;
alt_u8 currSEG = 0x00;

int ctrLED = 0;
int ctrSEG = 0;

int stateLED = 0;
int stateSEG = 0;

//Called when timer completes a period/pulse
void TimerISR(){
    Clear the Timer

    if(stateLED is Active){
        Display the value of currLED
        Increment ctrLED and move onto next switch
    }
    if(stateSEG is Active){
        Display the value of currSEG
        Increment ctrSEG and move onto next switch
    }

    if(ctrLED is max switches)
        Reset state variables related to LED and turn off LED
    if(ctrSEG is max switches)
        Reset state variables related to 7Segment Display and turn off 7 Segment Display
}

//Called when any of the buttons were pressed on the board
void ButtonISR(){
    Read Which Buttons were Pressed
    //LED display
    if(button 1 pressed){
        currLED = Read from the address of switch 1; //Initial
        Activate stateLED flag and reset ctrLED counter
    }
    //7SEG display
    if(button 2 pressed){
        currSEG = Read from the address of switch 1; //Initial
        Activate stateSEG flag and reset ctrSEG counter
    }
    Clear button interrupt flag
}

int main(void) {
    Register timer with appropriate frequency
    Clear the bit in the timer, and activate it
    Register the button's callback in IRQ
    Turn off LED/7SEG
    Infinite loop to keep listening for interrupts
    return 1;
}
```

Which synchronization method performed the best?

The advantage of interrupt methods as opposed to polling for this phase of the lab is that it suits an event-based program. In this case, the timer and the button pressing are both events (raising flags in the bit) that are handled by the interrupt method. Polling would have been an advantage because it was easier to implement and debug (since the program is timed and deterministic). Although polling might have been easier to implement, it would be inefficient as it would continuously use the processor when not needed; such as either button not being pressed for a while. The optimal synchronization method to implement for Phase 1 would have been the interrupt method.

Pros and Cons of Synchronization using Tight Polling

Pros:

- Easy to implement and debug
- Program execution would just be to follow the state diagram above (easy to code as well)
- There was no need to raise flags or manage them

Cons

- As mentioned in the previous question, it is inefficient as it uses processor to poll when not needed

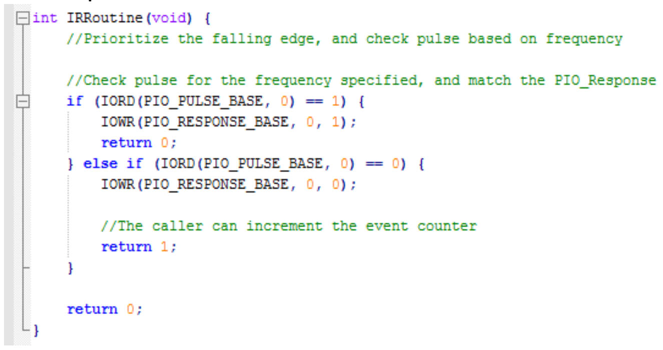
Testing Strategy

The testing strategy we used was coupled with how we implemented the solution. Upon adding a piece of the flow we would test the new feature; and use regression testing to test previous features. There was a debugger in the Eclipse UI that we were able to use to step through code. In addition, the console logging was really helpful to know what the program was doing. For example, after every crucial step; we used the *printf* function to output that the step was completed. This way, we knew where the program broke down, if it did. With the sample file provided to us, we began implementing the 7 Segment display portion of the phase. First we wrote code to ensure the display turns on, we then tested that functionality. Then we wrote code to listen for the button push and turn on the display, we tested that as well. After testing the hardware and software interaction, we programmed the appropriate logic to ensure that the state variables were taken into account. We then cumulatively tested to see if the full functionality works with the 7 segment display and the switches. Upon successful tests, we repeated the process for the LEDs and regression tested the 7 segment display. Our testing strategy ensured full functionality of our program and met the requirements for phase one.

Phase 2

Implementation of Periodic Polling & Interrupt Synchronization Methods

For this part of the lab, the two techniques of synchronization that was used was periodic polling and interrupts.

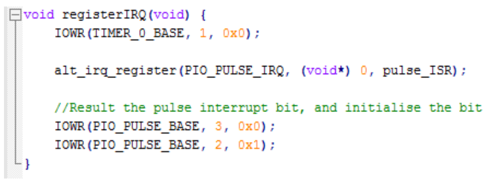


```
int IRRoutine(void) {
    //Prioritize the falling edge, and check pulse based on frequency

    //Check pulse for the frequency specified, and match the PIO_Response
    if (IORD(PIO_PULSE_BASE, 0) == 1) {
        IOWR(PIO_RESPONSE_BASE, 0, 1);
        return 0;
    } else if (IORD(PIO_PULSE_BASE, 0) == 0) {
        IOWR(PIO_RESPONSE_BASE, 0, 0);

        //The caller can increment the event counter
        return 1;
    }

    return 0;
}
```

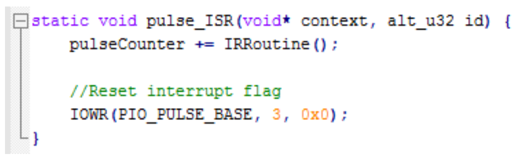
We start off by matching the pulse P_{IO} to the response P_{IO}. Since this function was both used in the interrupt and in the polling, we decided to make it into its own separate function shown above. To keep track of pulse during the two synchronization techniques, we increment the pulse value on a falling edge to indicate a full pulse has passed.

Interrupt Method

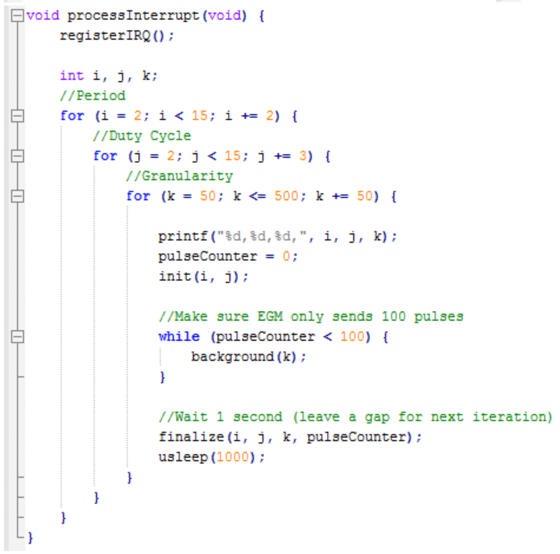
The interrupt method of synchronization involved the interrupt notifying the program with an event when the bit is set. The three methods below, along with the base interrupt method above was used for the interrupt synchronization. The *pulse_ISR* method keeps track of the 100 pulses. The *processInterrupt* method loops various test cases that vary in period, duty cycle and granularity. The *registerIRQ* method sets the interrupt and resets the edge capture register.



```
void registerIRQ(void) {
    IOWR(TIMER_0_BASE, 1, 0x0);

    alt_irq_register(PIO_PULSE_IRQ, (void*) 0, pulse_ISR);

    //Result the pulse interrupt bit, and initialise the bit
    IOWR(PIO_PULSE_BASE, 3, 0x0);
    IOWR(PIO_PULSE_BASE, 2, 0x1);
}
```



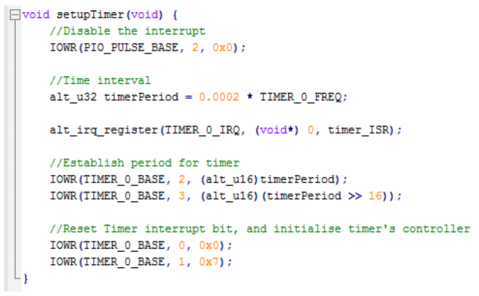
```
static void pulse_ISR(void* context, alt_u32 id) {
    pulseCounter += IRRoutine();

    //Reset interrupt flag
    IOWR(PIO_PULSE_BASE, 3, 0x0);
}
```



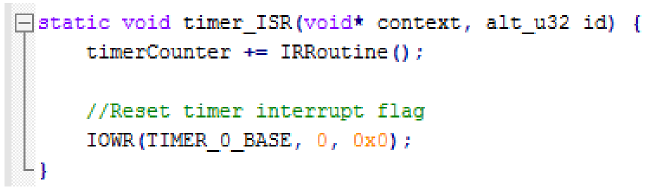
```
void processInterrupt(void) {
    registerIRQ();

    int i, j, k;
    //Period
    for (i = 2; i < 15; i += 2) {
        //Duty Cycle
        for (j = 2; j < 15; j += 3) {
            //Granularity
            for (k = 50; k <= 500; k += 50) {

                printf("%d,%d,%d,", i, j, k);
                pulseCounter = 0;
                init(i, j);

                //Make sure EGM only sends 100 pulses
                while (pulseCounter < 100) {
                    background(k);
                }

                //Wait 1 second (leave a gap for next iteration)
                finalize(i, j, k, pulseCounter);
                usleep(1000);
            }
        }
    }
}
```

Polling Method

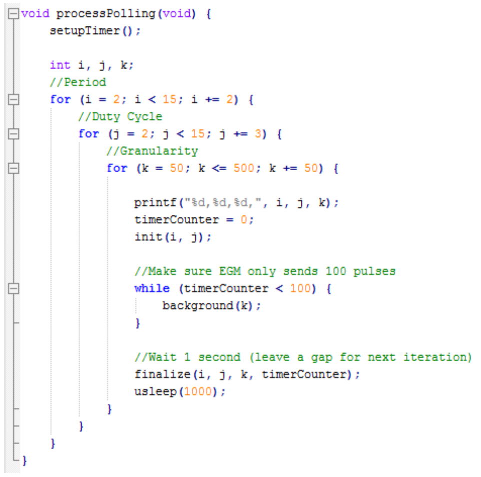
The below code indicates the polling synchronization technique. This involved continuously checking (polling) the interrupt at an established period. The way we implemented it was very similar to how that was done of the interrupt. The only difference was for the *setupTimer* method where we have more to declare than the interrupt. We had to disable the original interrupt as recommended by the TA, and then establish a timer interval/period.



```c
void setupTimer(void) {
    //Disable the interrupt
    IOWR(PIO_PULSE_BASE, 2, 0x0);

    //Time interval
    alt_u32 timerPeriod = 0.0002 * TIMER_0_FREQ;

    alt_irq_register(TIMER_0_IRQ, (void*) 0, timer_ISR);

    //Establish period for timer
    IOWR(TIMER_0_BASE, 2, (alt_u16)timerPeriod);
    IOWR(TIMER_0_BASE, 3, (alt_u16)(timerPeriod >> 16));

    //Reset Timer interrupt bit, and initialise timer's controller
    IOWR(TIMER_0_BASE, 0, 0x0);
    IOWR(TIMER_0_BASE, 1, 0x7);
}
```



```c
static void timer_ISR(void* context, alt_u32 id) {
    timerCounter += IRRoutine();

    //Reset timer interrupt flag
    IOWR(TIMER_0_BASE, 0, 0x0);
}
```

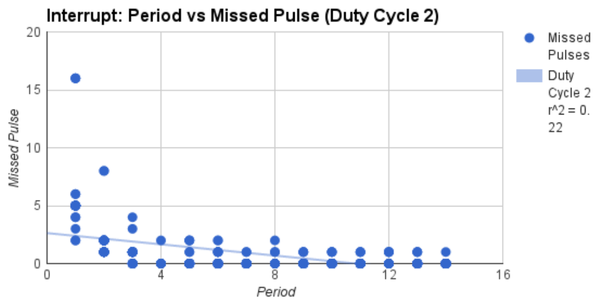


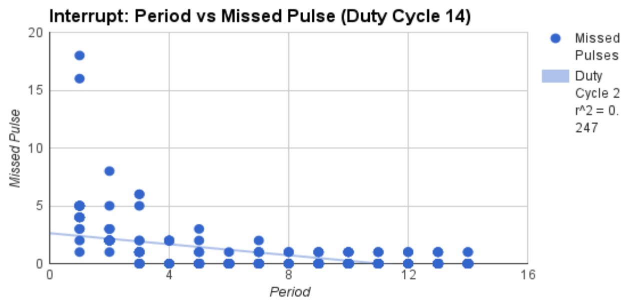
```c
void processPolling(void) {
    setupTimer();

    int i, j, k;
    //Period
    for (i = 2; i < 15; i += 2) {
        //Duty Cycle
        for (j = 2; j < 15; j += 3) {
            //Granularity
            for (k = 50; k <= 500; k += 50) {

                printf("%d,%d,%d,", i, j, k);
                timerCounter = 0;
                init(i, j);

                //Make sure EGM only sends 100 pulses
                while (timerCounter < 100) {
                    background(k);
                }

                //Wait 1 second (leave a gap for next iteration)
                finalize(i, j, k, timerCounter);
                usleep(1000);
            }
        }
    }
}
```

Experimental Results

Period vs Missed Pulses

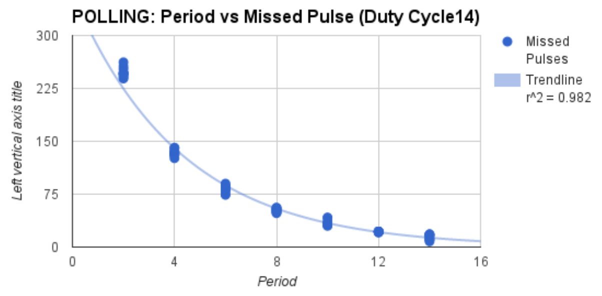
Interrupt

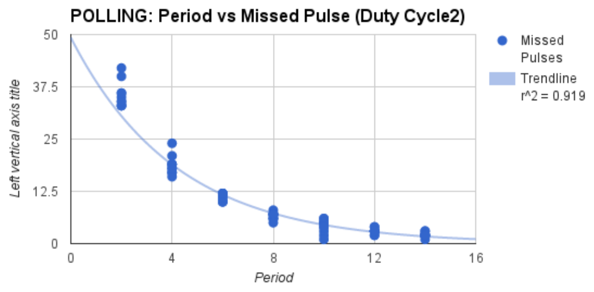




The graphs above show Period vs The Missed Pulse relationship for each duty cycle for the interrupt synchronization technique. According to the data, it shows that for majority of the periods show a very little missed pulse. This make sense since the interrupt should be active at every edge. The duty cycle seems to have little effect on the missed pulse. The trend line is very similar for the two end cases of the duty cycles.

Periodic Polling

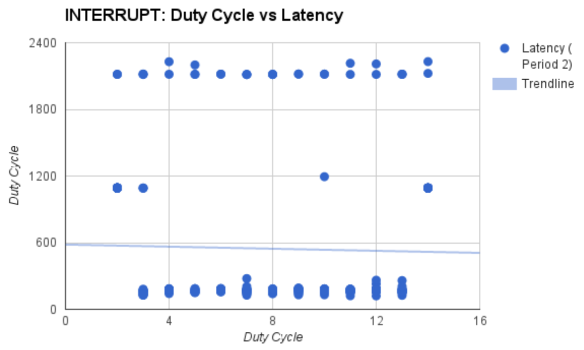


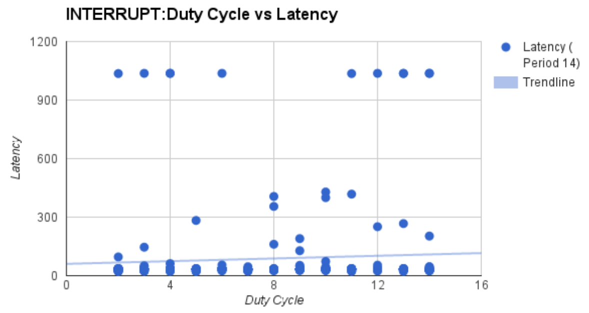


For the periodic polling, you can see a dramatic decrease of missed pulse as the period increases. This may be because when we have a short period, there may be changes that are too quick for the timer to pick up. As we allow for a longer period, we increase the timer's ability to accurately catch the pulse and miss less pulses. Unlike the interrupt there is a noticeable difference when the duty cycle change for the periodic polling. Although it approaches zero as period increase, at a lower period, a higher duty cycle had a much greater number of missed pulses. This could be because at a higher duty cycle it is much harder to catch the changing edge due to it being shorter.

Duty Cycle vs Latency

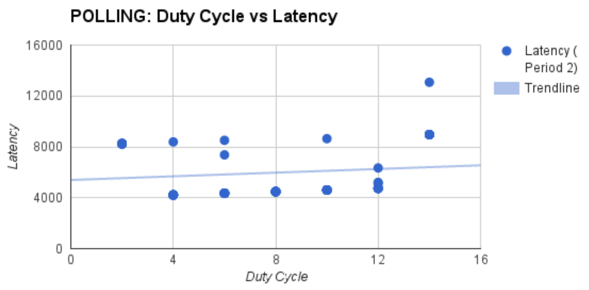
Interrupt

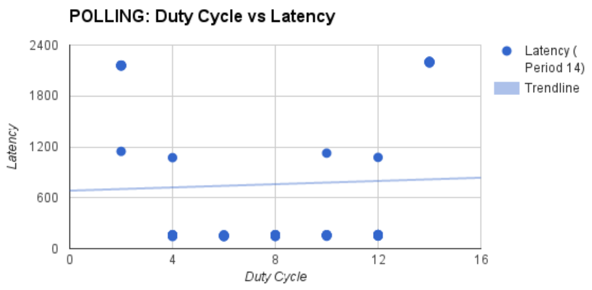




The two graphs above shows the duty cycle vs latency results for the interrupt at their respective periods (2 and 14). It shows that the duty cycle does not really affect the latency. However, the increase in period seems to lower the latency. This may because there are a fewer missed pulses as the period increased as shown in the graph *INTERRUPT: Period vs Missed Pulses*.

Periodic Polling



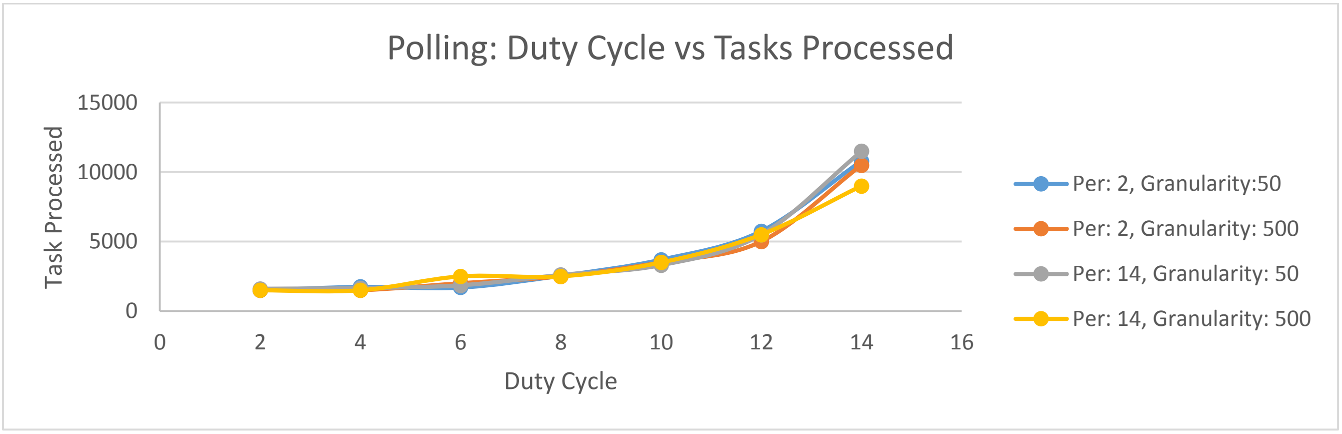


The two graphs above show the Duty cycle vs latency for the polling method. It seems as though the polling method has a higher latency than the interrupt. Like the interrupt method, we can see that increasing the period can also reduce the average latency. This is because we also see that increasing the period in the polling method can lower the amount of missed pulses.

Tasks Processed

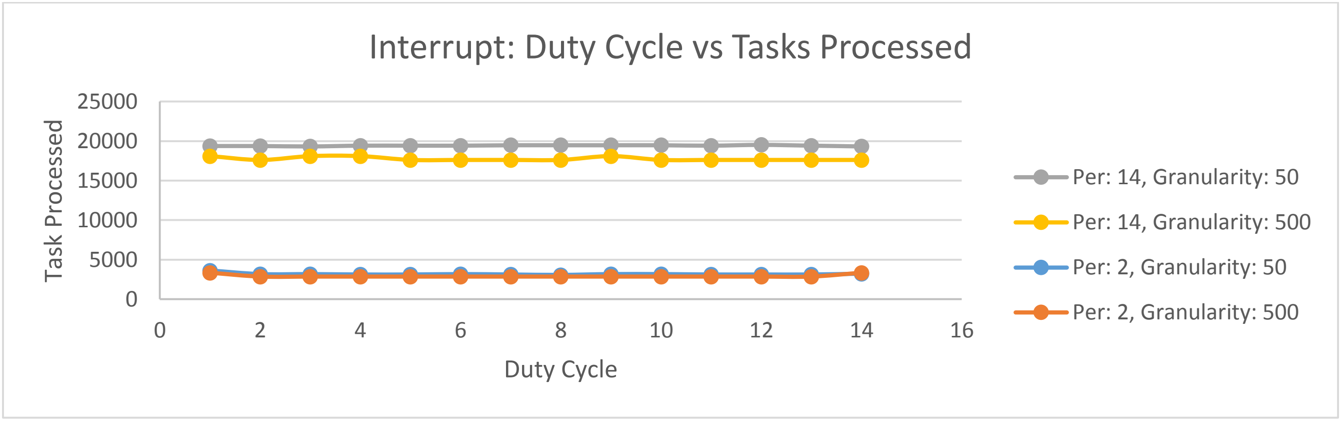
Polling

We notice that there is an increase in tasks processed as the duty cycle is increased. We also know from the previous section that the latency affects the duty cycle. The duty cycle increases tasks processed because it determines the percentage of time the polling method can spend processing the tasks. This can help conclude that the criteria for comparing the latency and background task progress is fairly consistent; and dependent on the duty cycle.



Interrupt

We notice that there is no increase in tasks processed for the interrupt method when there is an increase in duty cycle. However, there is a huge increase in tasks processed when the period is increased using the interrupt routine. This helps us conclude that a higher period will allow for more tasks to be processed using interrupt synchronization. This could primarily be because the interrupt is independent on time. The duty cycle has no impact on the interrupt method because it is not spending time polling the processor to see if there are any tasks to be processed.



Contribution

Ramie Raufdeen

With my programming experience, I was able to contribute to the software aspects of the lab; while learning a bit of the hardware from Joey. Programming in C was something new for me but I was able to pick it up relatively fast. In addition, I learned how to develop and debug code written for the NIOS II hardware. For the lab as a whole, it helped me better understand various synchronization techniques and how to benchmark them. I was able to relate the theoretical aspects of the course with the practical challenges in this lab. The lab instructions were pretty clear and the TAs were helpful during the lab sessions.

Joey Keum

I contributing in majority of the hardware side of this lab. When it came to coding, Ramie took the lead, and I assisted him in finishing the code. The most important thing I learned from this lab came from the post lab. After processing and compiling all the data at the end of the lab, it has really allowed me to understand the two processes better.

Notes for TA

- Our data for Interrupt method (csv file) included more test cases because we went period 1-14, and duty cycle 1-14 without skipping any intervals. However, in polling; we used the data that had period and duty cycle incrementing by 2 and 3 respectively. This made no difference to our plot as we simply neglected the extra data we had for interrupt method.
- The data is averaged out, following the trend-line would be ideal to see the patterns that we saw
- The TA had marked our polling missed pulses as too high (65000) or so, but we realized that he had accidentally read the wrong column off the output. We re-ran it after the lab session, and by then it was too late. It’s not a big deal, but we misunderstood and thought there was a problem with our code.