MECHENG 313: Real Time Software Design

Assignment 1 – AVR Traffic control

William Li 916720181
Hao Kang 562418291

1.0 Introduction

The 'AVR Traffic Control' assignment is a student project aimed at teaching mechatronics engineering students the principles of real time software design. To complete the assignment, students are to learn how to program an ATmega8 chip in C. More importantly, students are required to work with the hardware limitations of the microprocessor and design a program that meets all the functional and time requirements.

The specific tasks required are functions which phase traffic lights, alter the switching time for those lights in a configuration mode, measure the speed of traffic through two light barriers, and a red light camera that activates when a car runs the red light. All of these tasks were completed and the following report will detail the implementation and design decisions taken.

2.0 Classification

Task 1 was considered soft real-time systems. Soft real-time systems can sometimes miss deadlines without seriously impacting the intended functionality. Task 1 will maintain most of its functionality even if the lights do not change state exactly according to the specified time. Part of task 2 can also considered to be soft real-time system. The timing of each of the 4 period steps can withstand a certain degree of error for the same aforementioned reasons.

Task 3 and 4 were considered as firm real-time systems. Missed deadlines will not result in catastrophic events, however will render the results useless. If task 3 failed to perform within the time limit, the speed will be incorrect; while if task 4 fails to meet its deadline image captured will be blurry and hence useless. The part of task 2 which enters configuration mode can be considered a firm system. If the button is pressed however the system fails to enter configuration mode, the functionality will be rendered useless, however it's unlikely danger will arise from this failure.

Periodic tasks are ones that execute at periodic intervals. As such, polling tasks such as task 4 is considered a periodic task. The button's state is checked upon every entry of the superloop. Another example of periodic task will be the flashing of LED's throughout the assignment, such as the continuous cycling of lights at regular intervals in task 1.

Aperiodic tasks are ones that can happen randomly or unpredictably. Task 3 is implemented as aperiodic task due to the use of button interrupts, this can happen any time depending on when the user presses the buttons. The nature of interrupts means that it is not regularly checked, instead only executes the commands upon triggering of the pin.

3.0 Priorities and Partitioning of Peripherals

Task 3 has the highest priority as it has the tightest time requirements, both LB1 and LB2 need to respond within 1.5ms. As the ATmega8 has two hardware pin interrupts, we used both for Task 3 to minimise response time. Hardware interrupts have the highest precedence in the microprocessor, so this ensures task 3 executes as soon as the user presses the associated buttons.

Task 4 has the second highest priority as it also has a time response requirement, however, it is lower at 10ms. We have therefore decided to poll for the triggering of LB3. We ensured task 4's response time meets criteria by optimising our code such that the worst case response is always lower than the limit. Please refer to the results section to see exact performance achieved.

Task 1 and 2 were decided to have equally low priorities as neither have strict time requirements.

4.0 Timers

For this project we used ATmega8 microcontroller which has two 8-bit timers (Time/Counter0, Time/Counter1) and one 16-bit timer (Time/Counter2). Only Time/Counter1 and Time/Counter2 are used.

The ATmega8 is set to run at an internal frequency of 1MHz. We set Timer2 to CTC mode with a prescaler of 8 with a top value of 255, this results in the timer overflowing every 2 milliseconds. Time counter variables are incremented during the interrupts so these counters keeps track of time elapsed. Every 500 counts represent 1 second.

Timer1 is used in fast PMW mode with prescaler set to 256. The OCR1A pin will be set to a value between 0 and 255 producing a duty cycle that is equivalent to the speed of the car in kilometres per hour. OCR1B pin, is used to represent the number of cars that has crossed the red light, each additional car increments the duty cycle by 1%.

To ensure all tasks can run without affecting the functionality of the rest of the system, we included three timing variables. One for task 1 and 2 (as their operations are mutually exclusive), and one each for task 3 and 4.

5.0 Implementation

5.1 Task 1

Task 1 requires the board to cycle between red, yellow and green lights at designated intervals. This is achieved by using the remainder function to overflow a variable on 3, on entering the loop, the algorithm turns the next light in the cycle on, and the other 2 off. Refer to Figure 1 for task 1's code (the +3 is to shift to the correct ports as pb1/2 are used for PWM).

```
void normalMode(){
    light++;
    PORTB &= ~(1 << (light%3 + 3));
    PORTB |= (1 << ((light+1)%3 + 3));
    PORTB |= (1 << ((light+2)%3 + 3));
    PORTD &= ~(1 << 5);
    timeCounter = 0;
}
```

Figure 1 Task 1

5.2 Task 2

Task 2 allows the user to enter configuration mode and change the time interval at which task 1 changes state. If switch0 is pressed while the light is on red, system enters configuration mode, where task 1's cycle will stop. If switch0 is pressed on another light, system will enter configuration mode upon the next red light. Within configuration mode, LED4 will begin flashing to indicate the period based on ADC inputs.

ATmega8 features a 10-bit successive ADC, where input values will be between 0 and 1023 depending on the state of the potentiometer. To achieve four periods evenly divided through the range, we used the formula $\text{period} = \left(\frac{\text{adcInput}}{256}\right) + 1$. Since integer division floors, adcInput from 0 - 255 will return 1, 256 - 511 will return 2, 512 – 767 will return 3, and 768 – 1023 will return 4.

We then devised an algorithm to achieve the blinking of LED4, the polling time for the loop dynamically changes based on the current period. After the light has been turned on and off the same number of times as the period the polling time will be changed from 0.5 to 3 seconds, after which, the cycle will reset. Refer to Figure 2 for this algorithm.

```
void configurationMode(){
    ADCSRA |= (1<<ADSC);
    while (ADCSRA & (1<<ADSC)){}
    uint16_t adcInput = ADC;
    period = (adcInput/256)+1;
    if (timeCounter >= 250*configCycle)
    {
        if (configCount == 0) {configPeriod = period;}
        PORTD ^= (1<<5);
        configCount++;
        if (configCount >= (2*configPeriod - 1)) {configCycle = 6;}
        if (configCount >= (2*configPeriod)) {configCycle = 1; configCount = 0;}
        timeCounter = 0;
    }
}
```

Figure 2 Task 2

5.3 Task 3

Task 3 measures the speed of traffic by timing how long it takes a car to pass by two speed gates 20 meters apart. As previously mentioned, this task has the strictest timing requirements, so we decided to use a button interrupt for each speed gate. Since all switches on SKT500 are active low, INT0 and INT1 are both set to be falling edge triggered.

The assignment specifies that the two speed gates will always be triggered in the correct order. Therefore, we assigned our first button to reset time counter, and second one takes that time stamp and calculates the speed based on it. This speed is outputted as the duty cycle of OC1A pin, and it's capped at 100km/h. Refer to Figure 3 for implementation.

```
ISR(INT0_vect)
{
    timeCounter2 = 0;
}

ISR(INT1_vect)
{
    speed = 36000/timeCounter2;
    if (speed >= 100) {speed = 100;}
    OCR1A = ((speed*255)/100);
}
```

Figure 3 Task 3

We assume the slowest car to go through the light barriers will be going faster than 0.55 km/h, which we believe is a reasonable assumption. Since timeCounter2 variable is 16bits the longest time period it can count for is roughly 2.2 minutes, after this value is reached it overflows and starts counting back from zero.

5.4 Task 4

Task 4 requires a red light camera to be triggered when a car crosses a light barrier during red light. The camera is represented with flashing of LED3, and can only be activated when switch7 which activates the camera is triggered while the traffic light is red (LED0). When triggered the LED flashes half a second on, half a second off for two cycles. We also needed to keep track of the number of cars that ran the red light and output that as PWM, with the percentage duty cycle being the number of cars. The number is outputted onto OC1B pin, as this does not require setting up of an additional timer. Refer to Figure 4 for implementation of task 4.

```
void redLightCameraMode(){
    PORTB ^= (1<<0);
    OCR1B = redLightCount*2.5;
    redLightTimer--;
    if (redLightTimer == 0) {redLightCamera = 0;}
    timeCounter1 = 0;
}
```

Figure 4 Task 4

6.0 Results

The timing requirement for task 3 is to have an overall error of smaller than 3ms for each speed reading. This includes the resolution in our timing, and response time of the two buttons. As we only used overflow values for our timing, the worst case error is 2ms (timestamp is taken right before overflow happens), this means our button response times are required to be under 1ms collectively. As Figure 5 and Table 1 below show, our light barriers 1 and 2 react within 40 microseconds each. This means our task 3 will always meet the 3ms specifications. A consideration was made into reading the current timing value from TNTC2 to increase timing resolution from 2ms to 8us. However, as our current implementation already meets the specifications and adding more precision will not in any way change our outputs, we decided to omit it for better performance for the rest of the system.

Task 4 requires a response time of under 10ms between when LB3 is triggered to when LED3 first lights up. Since we've used both pin interrupts on task 3, we used polling to check for the triggering of LB3. Our worst case response time for this task is approximately 250 microseconds (Figure 6, Table 2), while best case is around 50 microseconds, which meets the required specification.

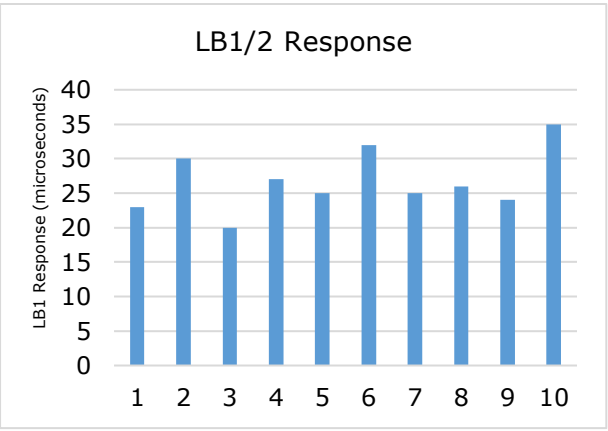

Table 1 LB1/2 response time

Figure 5 LB1/2 response time

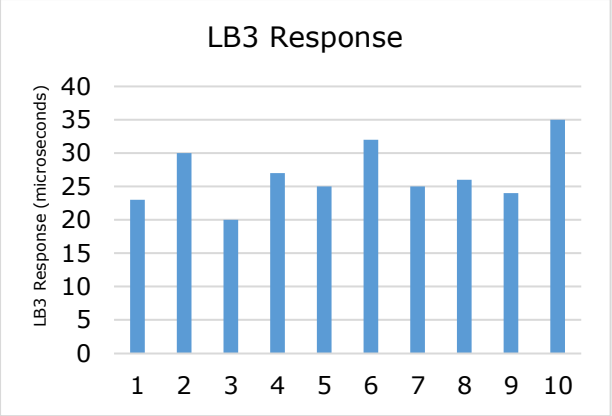

Table 1 LB3 response time

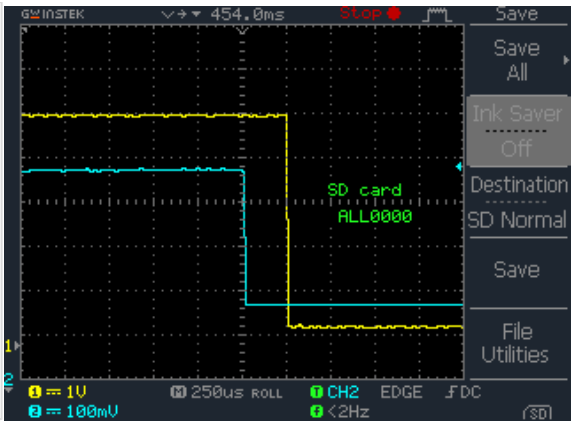

Figure 6 LB3 response time

7.0 Design decisions

Throughout the assignment, we strived to strike a balance between readability, performance, and shortness of code. First and foremost, we avoided long polling statements or nested loops for a more consistent runtime. This also helps with debugging as each line of code is more or less independent to each other. We then devised algorithms such as shown in figure 2 to shorten our code, these algorithms avoids the need to have duplicate code for each variation or case. In some places we traded some performance for cleaner code, an example of which is task 1, where we used the remainder function as opposed to conditional statements.

Finally, we abstracted the implementation of the four tasks into functions such that people who only wish to understand the behaviour of the program and not how it's implemented can just read the superloop. To make variables easier to access, we initiated all our variables to be global variables. As this assignment has no security concerns associated with using global variables, we decided it was an appropriate implementation for easier code maintainability and reusability.

8.0 Conclusion

Overall, the 'AVR Traffic Control' assignment has been a great learning experience, it has given us practical experience implementing a real time embedded system. The experience has highlighted the importance of design sacrifices when working with a relatively low end hardware such as the ATmega8 microprocessor. The right partitioning of hardware peripherals and design decisions allowed our system to seemingly run parallel processes on a single thread microprocessor.