Traffic Light Controllers

Introduction

This project focuses on programming a traffic-light controller system. The traffic lights loop through, from red to green to yellow, in order to control the flow of traffic. During operation, each light is on for the same amount of time, ranging from one second to four seconds, each integer second making up one of the four cycling modes. The user can enter configuration mode by pressing a button, while the dial turn amount controls the operating state. The traffic-light controller system can also monitor the speed of the vehicles on the road as they pass through light barriers LB1 and LB2. A speedometer like display can represent speed estimates of up to 100km/hr. The behaviour of the drivers is also monitored through a red-light camera, triggering a camera when LB3 is breached while the traffic light is on red, and logging the total number of cars who do run a red light.

This program is programmed using C language on an Arduino Uno board with an ATMega328P microcontroller.

Methods

Normal Mode

Under normal mode, the traffic light will automatically change state every second. Timer/counter1, a 16-bit timer, is used to control the cycle period. The operation of TinkerCAD simulations is compromised unless in normal mode, so CTC mode is emulated by declaring the value of OCR1A as 15625 and leaving the bottom value as zero. When the prescaler is initialized as 1024, timer/counter1 will match the value of OCR1A every second. Therefore, the COMPA interrupt service routine (ISR) will run every second, rotating through the lights in order, and resetting the value of timer/counter1 to 0x0000.

The state of the system represents the number of seconds each light is on for as it is cycling. Cycling through the traffic lights based on this state occurs similarly to normal mode, except OCR1A value is adjusted by multiplying the 15625 counts by the state number, so that each light is toggled after the appropriate number of seconds has passed according to the state.

Configuration mode

The traffic lights continue cycling until a push button is pressed, entering the system into configuration mode only when the light is on red. The push-button initiates an aperiodic ISR, where the system checks to see the configuration status, and either begins configuration mode or exits if already in configuration mode. When the traffic light is in configuration mode, there are four states which could be chosen from using the potentiometer.

A fourth LED is used to display the current state of the traffic lights. If ‘x’ represents the state number, the fourth LED will flash ‘x’ times in one second then remain off for the next two seconds. For this LED, another timer, Timer/counter2, was used to develop another overflow ISR separate from the rotating traffic lights. The prescaler is set to 128 and the OCR2A value is 125. As a result, compare A matches every 1ms. Since timer/counter2 is only

an 8-bit counter, a variable ‘num_loop’ is used to count how many times the timer has matched the OCR2A value. At the beginning of each loop, num_loop is set back to zero. When num_loop is equal to 500/state, a variable ‘blue’ is increased by 1. ‘blue’ represents the index number in a cycle period. The fourth LED will only be on when blue % 2 = 1 but blue < state * 2. It remains off when blue is less than state *6. When blue is equal to state*6, this means it reaches the end of the third second. Then blue should be reset to 1. This method toggles the lights on and off, for the number of pulses per second to match the state. The logic is shown in Figure 1. When ‘blue’ reaches the end of the third second, ‘blue’ is reset.

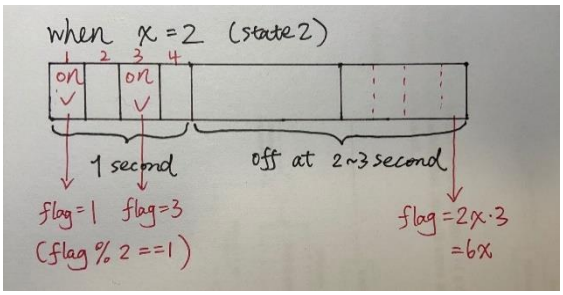

Figure 1 Example of state2

The potentiometer is used to determine the upcoming state. The potentiometer varies its resistance and outputs a voltage ranging from 0 to 5V, which acts as an input to the microcontroller. 5V Vcc is used as the reference voltage to map the input analog 5V range to a digital code ranging from 0 to 1024, a 10-bit number. The full range of the ADC outputs is 10 bits, and is used as specified, where every state is represented by 256 digital codes, ranging from 0 to 1023 in order of state number. A single analog to digital conversion is performed upon the exit of configuration mode during the second aperiodic button press. When the conversion is complete, the results are stored in data register ADCH and ADCL and the ADC conversion ISR is executed. The ADC ISR determines the chosen state from the register ADC, turns the normal mode blue light on, and updates the OCR1A compare register to control the cycle periods appropriately.

Speed Monitor

The speed of the vehicle is estimated from the time elapsed between passing LB1 and LB2, light barriers 20 metres apart simulated as push buttons. An oscilloscope receives a PWM signal whose duty cycle represents the speed of the vehicle just passed.

The first button triggers the interrupt ISR for INT0, which resets the timing parameters for Timer/counter2 to ‘start the stopwatch’. Timer/counter 2 (8-bit) has a prescaler of 256, where the count is incremented every 0.016ms and overflows every 4.096ms. The number of times Timer/counter 2 overflows is updated and stored in a 16bit integer in the Timer/counter2 overflow ISR. When the second button is pressed to signal the passing of LB2, the time elapsed is calculated by multiplying the number of overflows with ms/overflow, and adding the counter values in Timer/counter2 and multiplying this with ms/count. The speed (km/hr) is then calculated and converted to an appropriate value stored in OCR1A for the PWM signal which uses timer1. The PWM signal is updated upon LB2, to ensure that the signal persists until another vehicle passes through LB1 and LB2.

Timer1(16-bit) has a prescaler of 256, and makes use of the output compare interrupts A and B to emulate fast PWM mode. OCR1B is set to 62500, which triggers an ISR every second to achieve the 1s pulse period; OCR1A is dependent on the estimated speed, and is converted to a number that is a percentage of 62500 to form the duty cycle. Every second, the OCR1B ISR resets the counter to 0 and turns the signal high to mark the start of the PWM signal. When the counter value for Timer2 meets the compare value in the OCR1A register, the ISR turns the signal low. Signal changes are left as-is in the cases where the PWM signal is 0% or 100%.

Red-Light Camera

Upon a button press to signal the passing of a car, an ISR is run to check if the light is red. If it is, the number of cars caught is increased. Then the blue light is turned on to represent the camera trigger. Timer/counter2, an 8-bit counter, with a prescaler of 256, has the COMPA interrupt enabled and occurs when the count value in TCNT2 is at 125. The counter overflowing 125 times is equivalent to the passing of 250ms, a quarter of a second. The light is toggled on and off every 250ms to represent a 0.5s pulse period at 50% duty cycle. These counter values are reset at the start of the red-light breach.

A PWM signal representing the total number of cars that run a red light is written to an Arduino Uno digital IO pin, and displayed on an oscilloscope; the number of cars forms the percentage of the duty cycle for the PWM signal. Timer/counter0 was initialized as a top value of 125 and a prescaler of 256. When the compA ISR is served five times, the flag variable will plus one. The indication of the flag variable allows us to make changes to the PWM signal every 10ms (100 counts in 1 second), according to the number of cars that have been simulated by the tactile push button.

Results

Deadline 1 - Camera to respond within 10ms from when LB3 is reached

Since INT0 is the second priority interrupt vector, the external interrupt of the tactile push button will be served first. In the INT0 ISR, the blue light turns on, hence responding to the breach.

The interrupt execution response for all the enabled AVR ® interrupts is four clock cycles minimum. After four clock cycles, the program vector address for the actual interrupt handling routine is executed. During this four clock cycle period, the program counter is pushed onto the stack. The vector is normally a jump to the interrupt routine, and this jump takes three clock cycles (ATmega328p datasheet 6.7.1). This results in a minimum of seven clock cycles required in total. Since the clock frequency is 16MHz, one clock cycle takes 0.0625 microsecond. Therefore, the external interrupt response time is at a best case scenario of 0.4375 microseconds. Although this does not take into account other computational efforts, the system is very likely to respond within 10ms when light barrier LB3 is breached.

Deadline 2 - Speed estimates to have an error less than 1km/hr for speeds up to 150km/h

Deadline 2 is achieved by ensuring that the response time between the breach of LB1 or LB2 is to be less than 1.5ms. Similar to discussion on deadline 1, at 16MHz, each clock cycle takes 0.0625 microseconds.

A theoretical worse case is assumed where it takes 20 clock cycles for INT0 ISR to start executing, and 20 clock cycles for the ISR to run and turn on the blue light representing the camera trigger, and start the elapsed timer. The time taken for LB1 to respond will be in the order of microseconds, responding well in the performance range of 1.5ms.

For LB2, now assuming the time it takes INT1 ISR to start executing is in the order of @@@ microseconds. The time elapsed is the first operation of the ISR, and is calculated by performing two 16bit by 16bit multiplications and as well as an addition. 16-bit by 16-bit multiplication in ATMega328P requires 19 clock cycles. As multiplications are computationally more expensive, 3x19 clock cycles is an estimate of 57 clock cycles, and less than 4 microseconds. Although this does not account for other computation that may occur, it is much lower than the 10ms performance requirement.

The debugging tool on TinkerCAD was also used to verify this performance by looking at the simulation time provided. A breakpoint was placed at the start of INT0 ISR, to indicate the start of the time elapsed from LB1 breach (01.454s since the start of simulation). A second breakpoint was placed at the start of INT1 ISR to capture the time LB2 was breached (2.817s since start). According to TinkerCAD simulation time, the elapsed time is 1.363 seconds, and the value calculated from time elapsed is 1362.608 microseconds. Despite this method not capturing response time between button press and the ISR starting, it can still be seen that the elapsed time error calculated is less than 1.5ms (limited by 1ms resolution of TinkerCAD simulation times). Figure 2 details one of the timing tests done.

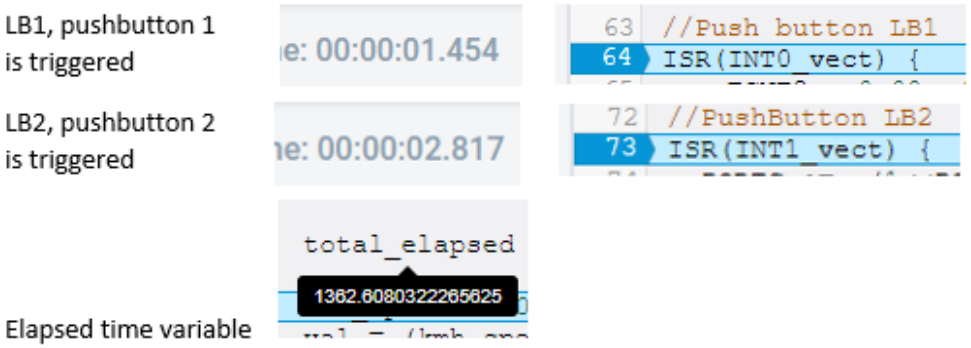

Figure 2 - Simulation result for elapsed time comparison for LB1 to LB2

Discussion

Operational design choices

The following are personal design decisions made to the project:

- When the traffic light is in normal mode, the ISR responsible for cycling through the lights occurs every second. Therefore the red light is turned on during the set up, so that the red light cycles to the green light on the second second.
- When the button is pushed to enter configuration mode, if the light is already red, the light will stay red and enter configuration mode, rather than waiting for the occurence of the next red light.
- The blue LEDs representing LB1 and LB2 for task three turn off briefly when its respective button is pressed, as if to indicate the barrier was broken.

Performance design considerations

Effort was made to maintain deadline 1 using the highest priority interrupt (after the reset interrupt) because the red light camera is a firm real-time system. The result has no utility after the deadline of 10ms has passed, but will not directly result in any severe consequences. The driver, unfortunately, might not get recorded, but if a car crash were to occur because of the breach, it would have happened regardless of the camera. That saying, it can lead to drivers thinking they could get away with running red lights, which is bad.

Speed measurement of cars is also a firm real time system. The utility of the result drastically decreases with increasing speeds, but would not result in any severe consequences. As mentioned in the results, a response time of 1.5ms was achieved. Counting the number of times Timer/counter 2 overflowed, and adding the values in the counter was used to maximise accuracy of time elapsed. Unfortunately, calculating the time elapsed by counting the number of overflows for a counter that overflows every millisecond resulted in an 13ms discrepancy between TinkerCAD simulation time and timeElapsed counter value, the timers perhaps not offering enough precision.

It is recognised that resource management (timers, external interrupts) could have been maximised, where one timer could have served more than one purpose. If all the subsystems were to be implemented on the same microcontroller, decisions around timer choices would change to allow for the entire system to operate using 3 timers. This would also apply to emulating CTC and fast PWM mode for the timers, as the functionalities may have been used to be more efficient with timer use. Normal mode was the only timer mode used due to TinkerCAD simulation constraints.Timer/counter2 was used over timer/counter0 based on which task was deemed to take higher priority in relation to timer/counter1. For example, in task 3, calculating the elapsed time accurately used timer/counter2 as it was more important than updating the PWM signal.

Programmed IO is not a perfect solution to this design. Programmed IO has single output and single input instructions. Therefore, programmed IO cannot handle two I/O transfers at the same time. Only once the first transfer has completed, can the second transfer start. If the user presses the push button when the processor is occupied by blocking code (such as the changing state in task 2), the processor may miss the upcoming signal (push button) entirely. However, IO interrupt allows the processor to serve interrupt routines according to their vector priority.Furthermore, I/O interrupt has a higher efficiency as well because interrupts enable the processor to handle events as they rise. This saves the processor from having to continuously poll. However, adding functionality to a polling program often comes at the cost of increasing worst-case response time. This response time may be accumulated to a sufficient high value when the design is relatively complicated