# 1. Introduction

The objective of this project is to program and simulate a traffic light controller, hereinafter referred to as the “system”. The system is to be coded in embedded C language on an STK500 board. The simulated system is required to act in ways similar to real-life traffic flow control systems. The lights are to phase between green, yellow, and red at a period configurable by the user. Using light barriers (LB1 through LB3), the system will also implement select safety features seen in real traffic light systems such as speed monitoring and a red-light camera to detect when a vehicle passes through the intersection illegally.

At times, the system will need to perform several different tasks simultaneously, effectively performing “parallel” operations. As the system is to be used in an application where human safety is at risk. accurate timing and fast response are important factors that must be considered throughout the entire system design. This report outlines in detail the solution that our team produced in response to the project brief.

Our team defined goals for producing a successful solution to the brief outline:

* All tasks meet functionality requirements,
* All tasks run in “parallel” as concurrent operations,
* Events are timed accurately,
* The system responds swiftly to inputs,
* and, Don’t blow up the board, because that is also very important.

# 2. The System Design

## 2.1 The Design Approach

Our team considered two design approaches when designing the functionality of the system: creating a system that solely satisfies the brief versus creating a realistic system that would reflect the operation of realistic intersection and traffic flow control systems.

The core difference between these two system designs is the restriction of all other functionality when in the system’s configuration mode. In a realistic scenario, the intersection would be shut down when configuring the system to avoid any potential hazards caused by irregularities the system may exhibit. When designed for realism, the system would use less processing power and require a less complex design. It wouldn’t have to measure the speed of vehicles passing through the intersection, nor would it need to record vehicles that ran the red light.

Ultimately, our team decided to design the system for brief fulfillment as it proved to require a more complex system design. Our system was designed to encapsulate each different functionality specified by the brief. This allows the user to add and remove complexity from the system at will. If the system were to be implemented in a real-world case, it would be very accepting of any modifications required with other functionality remaining unaffected.

## 2.2 Periodic Tasks

The basis of our system is an infinitely operating “superloop”, looping as fast as the processor allows it to until the program is killed. Execution of periodic tasks is controlled by timer overflow counters.

### 2.2.1 Timer0 Overflow Counters

To implement periodic tasks effectively, we configured the Timer0 overflow interrupt to increment global variables which track the number of overflows. This allowed us to define seconds and half-seconds in terms of number of Timer0 overflows; 3906 overflows per second and 1953 overflows per half second respectively. As there were 3906 overflows per second, this method of time measuring was deemed appropriate for our purposes, accurate to values in the microseconds range.

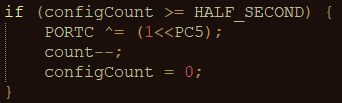
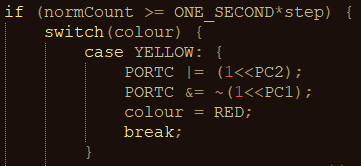
The Timer0 method of time tracking works by comparing the number of overflows to the defined variables for any number of seconds in “if” statements. These statements are only entered once a certain amount of time passes, shown in Figure 1 to the left, achieving periodic operation without the need for blocking functions such as “delays” or single condition “while” loops. These iterating global variables can be set and reset as the user requires, allowing for complete control of system timing.

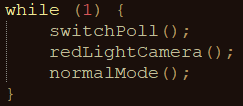
Figure Overflow counter entrance condition

### 2.2.2 normalMode()

The “while(1)” loop within the main() function continuously calls the normalMode() function, which is the basis for periodic system operation. The function serves as a “fancy light switch”; recording the previous state of the system as a global enumerated variable and acting through a switch case to transition to the next defined state. This operation is comparable to that of a finite state machine.

Being the base function for the operation of our system, all other system functionality had to be built normalMode(). This meant that correct blocking avoidance had to be achieved so the system would be as efficient as possible. After much brainstorming and many function iterations, an appropriate structure was formulated. Using the appropriate entrance condition shown in Figure 2, the function can iterate, resetting the reference count once complete.

Figure normalMode() entrance condition

Although it is called many thousands of times in the while(1) loop per phase, it is only allowed to perform functionality when the entrance condition is met. This is the basis for the rest of our system design, allowing for a “concurrent” system to operate on functions piece by piece when the system code grows more complex.

Although many functions are called consistently within the while(1) loop, shown to the right in Figure 3, giving the illusion of periodicity, their entrance conditions are based on discrete, aperiodic events, rather than periodic timing overflows. These are explored in the next section.

Figure Superloop function calls

Our team classified normal mode as a hard real-time operation. The normalMode() functionality runs the entire real-time system. If applied in a real-world scenario, the failure of this functionality would be catastrophic, leading to the complete failure of the intersection and an increased potential for harm to come to road-users.

## 2.3 Aperiodic Tasks

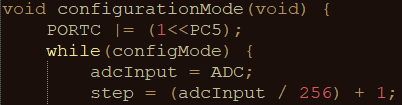
As mentioned in the previous section, the periodicity and aperiodicity of a function is defined by the type of entrance variable it uses. It is clear to see in Figure 3 in section 2.2.2that the entrance variable is periodic as it is based on a timing interval. However, if we consider the configurationMode() function shown to the right in Figure 4, it is clear to see that it is only entered once a specific condition is satisfied. This is the basis for our system’s aperiodic operation.

Figure Aperiodic function entrance variable

### 2.3.1 configurationMode()

The function configurationMode() is implemented as its own function, as it is defined as a “different state” to the normalMode() operation. As specified in the project brief, the light state is to remain red when in configuration mode. As such, configurationMode() implements blocking “while” functions, ceasing the light cycling that occurs in normalMode(). It makes logical sense that configurationMode() and normalMode() cannot run simultaneously, so in this situation, blocking was deemed appropriate to use.

The function of configuration mode is to continuously read and evaluate a potentiometer input controlled by the user. The input range is divided into four sections, each relating to a step size from 1 to 4. This step size is then returned to the user in a light flashing sequence to notify them of the step they have selected.

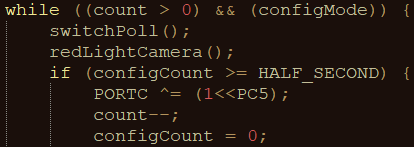
Despite normalMode() and configurationMode() having this unique relationship, all other functionality is expected to run irrespective of the current system mode. Therefore, the functions switchPoll() and redLightCamera() are included in the configurationMode() function code, as shown on the left in Figure 5.

Figure Functionality embedded in configurationMode()

This function is classified as a soft real-time operation as, although its failure would degrade system quality, it would still be tolerable, and the system would still run effectively without any loss of functionality.

### 2.3.2 switchPoll()

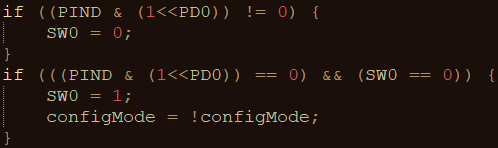
Switch polling had to be implemented due to the restricted number of hardware interrupts available on the STK500 board. switchPoll(), a function called each time the superloop is run, is used to check if specific switches have been pressed. At the time of polling, if the switch is not being pressed, then the switch condition will not be met. If the switch is being pressed then the switch condition will be changed to true, but only if the switch was not pressed last time it was polled. This makes the function slightly more efficient as we estimate that polling will occur many times over the duration of a human pressing the switch. This logic is shown to the right in Figure 6.

Figure switchPoll() logic

### 2.3.3 Speed Monitoring Function

The speed monitor functionality is the only code that doesn’t have its own dedicated function. To record an accurate speed measurement, the response time of each light barrier had to be less than 1.5 milliseconds. This meant that polling had the potential to be too slow depending on the system load. The light barrier functionality is therefore implemented in the two hardware interrupt functions; LB1 in INT0 and LB2 in INT1.

The speed monitor calculates the speed of a vehicle travelling through the traffic lights. The speed is calculated by measuring the time taken between the triggering of two light barriers, 20 meters apart. In this system, the light barriers are represented by SW5 and SW6, connected to the INT0 and INT1 pins respectively. The speed in kilometers per hour is then represented by the duty cycle of a PWM output. A vehicle travelling at 24 kilometers per hour should give a PWM of duty cycle of 24% after passing through the traffic light system. A vehicle travelling at any speed greater than 100 kilometers per hour will give a duty cycle of 100%, saturating as a PWM cannot have a duty cycle larger than this.

The first light barrier for the speed monitor is represented by a switch which when pressed, triggers an interrupt. In this interrupt vector, the speed counter is reset, and the speed PWM is set to zero. A flag variable SW5 is also set to true, to signal that the first speed barrier has been crossed. The second light barrier is also represented by a switch which when pressed, triggers an interrupt. Before carrying out any actions, the interrupt vector checks to make sure the first barrier has been crossed, using the flag variable. Once both light barriers have been crossed, the speed is converted into percentage duty cycle for the PWM, which represents the car’s speed in kilometers per hour. The speed monitor can be considered aperiodic, since the light barriers can be crossed at any time and are not dependent on timers to operate.

The speed monitoring functionality has been defined as a firm real-time operation by our team. Due to the varying delay times between pressing LB1 and LB2, a significant delay in functionality response would cause the loss of speed information, and in some cases, no execution of the task. Furthermore, if the system were to be used to fine people travelling at excessive speeds, a relatively firm response time would be required.

### 2.3.4 redLightCamera()

The red light camera should be triggered when a car crosses LB3 during a red light. When the red light camera is triggered, LED3 flashes twice, and the duty cycle of the output PWM increases by 1% per car. This is a relatively simple functionality with a much greater response time of 10 milliseconds, so it is implemented in the superloop with switch polling rather than as a hardware interrupt.

Once the red light switch has been pressed, the switch poll checks if the current light is red. If the light is red and the switch is pressed, a flag is raised. Once back in the superloop, the entrance condition for redLightCamera() is satisfied. The output PWM duty cycle increases by 1% for each time the switch has been pressed. LED4 will flash twice, and if LB3 is triggered during the flash sequence, it will restart.

The redLightCamera() has been also classified as a firm real-time operation as a significant delay in the function executing would render computed results useless. Although updating the PWM itself isn’t firm real-time, in a real-world scenario, a delay in notifying the system that a vehicle ran the red light would produce inadequate timing for a photo of the vehicle’s number plate to be recorded. Thus, a delay in performing the flashing sequence would render the function’s results useless.

## 2.4 Blocking Avoidance

All timing was done using the Timer0 overflow interrupt, which iterated all global counter values each overflow. This approach meant that all periodic tasks were carried out using if statements that checked these counter values, rather than using while loops and delays, which blocked the free-flowing functionality of the superloop. With this approach, the average time through the superloop was 95 microseconds. The code running in this manner gives an illusion that all tasks are running parallel to each other.

## 2.5 Hardware Interrupts vs. Polling

Ideally interrupts would be used for every switch for the fastest response times. The ATmega8 board is limited to only two hardware interrupts, which meant these interrupts were used for tasks with the fastest response deadlines. The remainder of switches were processed using polling. We used the two hardware interrupts for SW5 (LB1) and SW6 (LB2), as they both had 1.5-millisecond response deadlines, while SW7 (LB3) has a 10-millisecond response deadline and the configuration switch, SW0, has no specified deadline. Each time through the superloop SW0 and SW7 are polled to check if they are being pressed. We assumed that the average press time of a switch is less than the time taken through one superloop iteration.

## 2.6 Task Priority

The priorities of each task were ranked by their response deadlines. The speed camera task was the highest priority, having a 1.5-millisecond deadline for both light barriers. To ensure that this task had the highest priority in the code, the light barriers were triggered using hardware interrupts, executing regardless of where in the code the program is running.

The next highest priority task is the red light camera, with a response deadline of 10 milliseconds. As both configuration mode and the red light camera function are triggered by switch polling, they are treated as having the same priority. Both switches are polled at the same in the switchPoll() function, however in the superloop, redLightCamera() is called before configurationMode() to ensure that the red-light camera is given priority. The normal mode operation of the system is given lowest priority; all other tasks will interrupt normal mode to execute. It is not important if the light cycling takes slightly longer to respond (in the range of milliseconds), so long as the cycling stays in sync with the clock.

Task priority is not to be confused with the “hardness” of the real-time operation. In fact, our “most hard” task, normalMode(), also holds the lowest priority interrupt-wise. This is not an issue as the robustness of our entire code ensures that the hard, firm, and soft real-time definitions are fulfilled.

# 3. Results

## 3.1 Speed Monitor Time Response

The LB1 and LB2 timestamp responses are to each be less than 1.5 milliseconds to satisfy the requirements set by the brief. The LB1 timestamp response is defined as the time between LB1 being crossed and the variable counter starting iteration, and the LB2 timestamp response is defined as the time between LB2 being crossed and the speed calculation to complete. These requirements are drawn from the maximum error tolerable when measuring vehicle speeds of up to 150 kilometers per hour. In achieving these time responses, the percentage error of the system calculation should consistently be less than 1%.

The results achieved by our system, tabulated in Appendices I and II, were average response times of 31.6 microseconds and 661.2 microseconds for LB1 and LB2 respectively. LB2 had a significantly slower response time as the speed calculation had to be carried out before the timestamp recorded.

## 3.2 Red Light Camera Time Response

As specified by the brief, the LB3 timestamp response between the light barrier being crossed and the first camera flash is to be less than 10 milliseconds for an appropriate “photo” of the perpetrators number plate. Ten response times under different system loads were recorded and averaged, as displayed in Appendix III. The average response time was found to be 70.6 microseconds, well within the 10-millisecond limit.

## 3.3 Superloop Time Response

The time taken to complete a superloop in normal mode wasn’t defined as a necessity in the brief. We decided that it was important to be kept as small as possible. This is because most of the functionality, such as the polling and red light camera functions are called once per superloop. Essentially the superloop time needed to be kept as small as possible to give the effect of concurrent operation. Ten response times under different system loads were recorded and averaged, as displayed in Appendix IV. The average response time was found to be 94.6 microseconds. We are satisfied that this super loop time is sufficient enough to carry out all tasks concurrently and make the system reactive.

# 4. Conclusion

# The final system that was designed by our team is successful in meeting all requirements for the project. The system was created using embedded C programming on the AVR ATMega8. This board allowed us to use functionality of interrupts and polling to meet deadlines and ensure tasks were run according to their priorities.

# The functionality of how all tasks work individually and together was unclear, so we had to make assumptions on how the system should work in a real-world context. Based off these assumptions, the code was written to satisfy all requirements of the traffic light system. All tasks met the required response deadlines, having significantly smaller times than the deadlines. The system that we have designed is robust and functional in theory, however if it was implemented in reality it could be improved further to account for unexpected issues and improved hardware capabilities. We are confident that our system has been designed to easily accommodate for extra functionality if required. Ideally, every switch would be paired with a hardware interrupt to ensure fast responses.

# 5. Appendices

## 5.1 Appendix I: LB1 Timestamp Response

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| LB1 timestamp response time (µs) between LB1 crossed and counter started | | | | | | | | | | | |
| Test | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Avg |
| Time us | 26 | 68 | 27 | 26 | 26 | 31 | 27 | 26 | 32 | 27 | 31.6 |

## 5.2 Appendix II: LB2 Timestamp Response

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| LB2 timestamp response time (µs) between LB2 crossed and calculation completed | | | | | | | | | | | |
| Test | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Avg |
| Time | 692 | 676 | 684 | 640 | 676 | 648 | 632 | 640 | 632 | 692 | 661.2 |

## 5.3 Appendix III: LB3 Timestamp Response

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| LB3 timestamp response time (µs) between LB3 crossed and the first camera flash | | | | | | | | | | | |
| Test | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Avg |
| Time | 70 | 122 | 56 | 27 | 47 | 56 | 140 | 81 | 38 | 69 | 70.6 |

## 5.4 Appendix IV: Full Superloop Runtime

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Full superloop runtime | | | | | | | | | | | |
| Test | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Avg |
| Time | 65 | 66 | 138 | 66 | 65 | 139 | 65 | 66 | 138 | 138 | 94.6 |