Embedded Software Engineering – Smart City

\*Note: Sub-titles are not captured in Xplore and should not be used

Rodriguez Gutierrez Luis Fernando   
Master Embedded Systems Engineering  
FH DortmundDortmund, Germany

Mohammed Al Salihi  
Master Embedded Systems Engineering  
FH DortmundDortmund, Germany Leander Hackmann  
Master Embedded Systems Engineering  
FH DortmundDortmund, Germany

Hazhir Amiri  
Master Embedded Systems Engineering  
FH DortmundDortmund, Germany

*Abstract*—This paper introduces a smart city initiative using cameras to provide certain signals to a controller; to swiftly recognize and prioritize emergency vehicles. The system adjusts the traffic intersection waiting time for a rapid passage of vehicles and dynamically adjusts the time upon certain constraints. In this paper, we will only cover the theoretical applications and advantages of implementing this system in a city ecosystem.

Keywords—Smart City, End Node, Traffic Manager,

# Introduction

Normally we understand a “Smart City” as something that’s far from us as if it is something to be implemented in the future. However, the concept of a smart city is already being implemented. Cameras play a crucial role in today’s life, just as they are normally used for security measures.

The key idea of this implementation of a smart city is deploying a new traffic light that is going to be able to have a camera. The cameras in this implementation are going to play a crucial role; firstly, in recognizing emergency vehicles, which can be ambulances, law enforcement, or fire trucks. The other purpose of these cameras will also be to optimize the traffic depending on the number of vehicles at the intersection.

What makes this initiative more impactful is the adaptability that it could offer. The system could be able to interact with the city system as soon as it is connected, as once trained to recognize the scenarios, and interact with them, it will become a “simple plug and play” system.

The system will also be able to interact with the city to optimize the power consumption as needed depending on the scenario or situation.

# Analysis & Design

1. Intersection

A screenshot of a computer

Description automatically generated

Figure 1. Traffic Light Case Scenario.

In the most basic case scenario that was implemented for the smart city traffic intersection, it was considered an emergency vehicle needing to cross an active intersection, for this we mean that the vehicle does not have a green light when it arrives at the intersection. The system will recognize this vehicle through the camera to give priority, so it does not need to stop and wait.

The system will provide a warning to the other direction so they know that a change in the traffic light will occur.

1. Traffic Light

A traffic light is a device at street intersections that controls the traffic flow using a color code (RED, YELLOW, GREEN). The colors of a traffic light change in a timely manner, this usually is implemented using a scheduler. This needs to be coordinated to avoid collisions at the intersection.

* 1. Analysis

The traffic light will be the component that the actors will interact with, in this case mainly vehicles. The traffic light is to give the actors the proper instructions using a color code (RED, YELLOW & GREEN) so the actors can act depending on the color code.

A diagram of a company

Description automatically generated

Figure 2, State machine diagram.

* 1. Design

The implemented traffic light was implemented as an end node, where it will receive which lamp to ignite depending on the information received by the traffic manager.

In addition, this module will have a camera to monitor in case an emergency vehicle is approaching the intersection. In such a case, a signal will be sent to the traffic manager to prepare the intersection for this event.

A screenshot of a computer

Description automatically generated

Figure 3, Traffic light block diagram.

1. Smart Power

Street illumination is an important part of every city; however, the power consumption is usually high, normally this is the result of the current implementation of street lighting where it is either on or off.

* 1. Analysis

The streetlights should be an extension of the traffic manager, this can simplify the whole integration of our system into a running city while allowing a modular implementation, where this can be added to the traffic manager or not.

* 1. Design

The design involves an intelligent system where streetlights respond not only to the presence of vehicles but also to the time of day, specifically after it gets dark. When a vehicle is detected, regardless of whether it's an emergency vehicle or not, and it's past the configured time indicating nighttime, the streetlight is triggered to activate. The unique feature of this system is that the streetlight intensity is modulated, reaching a maximum of 80% brightness. This adaptive lighting approach ensures that the road is adequately illuminated for safety without causing unnecessary light pollution during nighttime hours. By integrating vehicle detection technology with time-sensitive activation, this system contributes to both enhanced safety for drivers and pedestrians and energy efficiency by optimizing streetlight usage based on the prevailing conditions.

A screenshot of a computer

Description automatically generated

Figure 4, Smart Power internal block diagram.

1. Traffic Intersection

This implementation is contemplating an intersection where 4 streets are going all in different directions. For each direction, there is a traffic light, just as in reality. For each intersection, there is going to be a sensor, which for this case of study is a camera. All traffic lights will be connected to another device, this device we are going to refer to as traffic manager. The traffic manager is going to process all the information coming from the sensors, and this will need to go through the traffic lamp.

* 1. Analysis

The traffic lights will need to interact with each other, so collisions are avoided. At the same time, the traffic lights will need to send the information to a central module, this to be analyzed and depending on the information, create an appropriate response.

* 1. Design

For this logic, we need the traffic lights to be “end nodes”. For end nodes, we mean that this module will perform no transitions or analysis of the system. The end nodes will simply receive the information needed from the traffic manager and report if an emergency vehicle is detected.

A computer screen shot of a diagram

Description automatically generated

Figure 5, Traffic Intersection block diagram.

For this implementation, the traffic manager will consider 2 scenarios, one of them is where an emergency vehicle is approaching, and the second one is for normal vehicles.

* Emergency Vehicles

A diagram of a computer

Description automatically generated

Figure 6, Emergency Vehicles Sequence diagram.

In the case of emergency vehicles, like ambulances or fire trucks, the system operates in a straightforward manner for efficient response. When the system's camera detects an approaching emergency vehicle at an intersection, it triggers a rapid response. The traffic controller, responsible for managing all the traffic lights, immediately signals a warning state by turning all lights yellow. This yellow light serves as an alert to everyone on the road, indicating the presence of an emergency vehicle and prompting caution.

Following the warning phase, the traffic controller takes a proactive measure by temporarily stopping all traffic. This temporary traffic block ensures a clear and unobstructed path for the approaching emergency vehicle, allowing it to navigate through the intersection smoothly and without delays. The synchronized orchestration of the warning signal and subsequent traffic control measures not only prioritizes the urgent needs of emergency services but also contributes to overall road safety. By swiftly adapting to critical situations, this system effectively manages the flow of traffic, ensuring a swift and unimpeded route for emergency vehicles through the urban road network. In essence, it's a simple yet powerful approach that emphasizes both responsiveness and safety in emergencies.

* Non-Emergency Vehicles

A diagram of a computer program

Description automatically generated

Figure 7, Non-Emergency Vehicles Sequence diagram.

In the context of non-emergency vehicles, the system employs a scheduling logic for efficient traffic flow. The traffic controller utilizes a scheduler that dynamically assesses the number of cars in each road direction at the intersection. When making decisions about which direction should move next, the scheduler prioritizes the road direction with a higher volume of vehicles, considering predefined time constraints.

Here's how it works: The scheduler constantly evaluates the traffic load in real-time. If one road direction has a larger number of cars waiting, the traffic controller gives it priority during the next green light phase. This prioritization is subject to predefined time limits to ensure fairness and efficiency. Essentially, the system adapts to the current traffic conditions, allowing the road direction with more cars to proceed first within the allocated time frame.

By implementing this logic, the system optimizes traffic flow, reduces congestion, and minimizes waiting times for drivers. It's a proactive approach that enhances the overall efficiency of the intersection by dynamically responding to the real-time demands of traffic, ensuring a smoother and more balanced movement of non-emergency vehicles through the road network.

1. Traffic Accident.

Collisions are an inevitable side-effect of the commotion of cars if human drivers are behind the wheel and pedestrians crossing a road. Although traffic systems may work efficiently, traffic accidents still happen. What can be done is to prevent further damage when one occurs; preventing and minimizing chain collisions is one way to achieve this end. Using the current infrastructure of cameras within an intersection, the traffic accident system monitors the intersection for any collisions and reports it to the traffic manager. Afterward, the traffic manager will decide accordingly, minimizing the chance of chain accidents.

* 1. Analysis

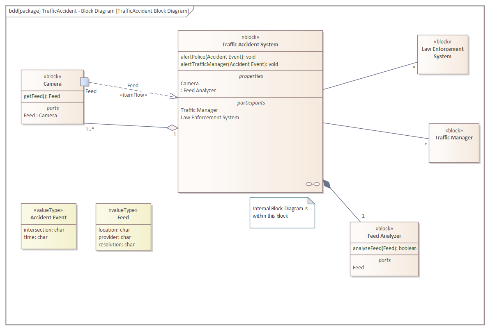


Figure 8 - Traffic Accident System block diagram

The traffic accident will make of use the existing infrastructure of cameras as data. It will analyze said footage and if it decides that an accident has occurred, it will alert the traffic manager of the relevant intersection and law enforcement. For this to happen, the system can make use of 5 recent footage for analysis. If there is no accident, it will discard the footage. Otherwise, it will save the footage for later in case the government or insurance companies need it. The system must react to an accident within ten seconds to reduce damage.

* 1. Design

For the analysis of footage, a Feed Analyzer component was defined is responsible for reviewing the footage and deciding if an accident has occurred. The analyzer needs footage from cameras that are presented as Feed objects. To correctly analyze footage, some constraints were for the Feed type defined. For example, the analyzer needs footage with a resolution of at least 1280 pixels wide in 720 pixels long to be able to properly analyze the footage. This among other constraints ensures that the analyzer receives viable footage.

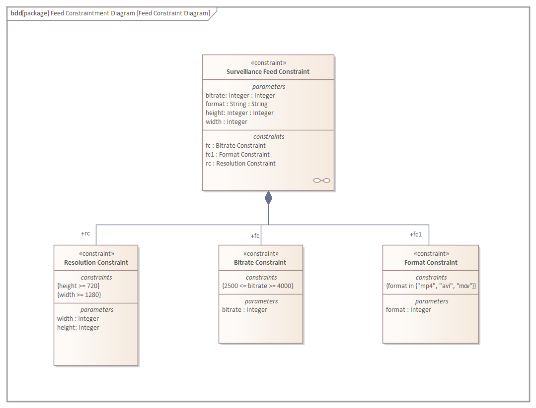


Figure 9 - Feed constraint diagram.

After each round of analysis, if the component detects an accident, an Accident Event object with information regarding the intersection and time of the accident will be created. This object will be passed to the relevant Traffic Manager and law enforcement agencies for further decisions.

1. Fault Tree Analysis

For this part, an analysis of two failure events was carried out: the event of a Traffic Light not working and the event of a collision within the intersection. After identifying the root events for these two top events, their possibility was checked against the implementation.

* 1. Traffic Light Not Working Fault Tree

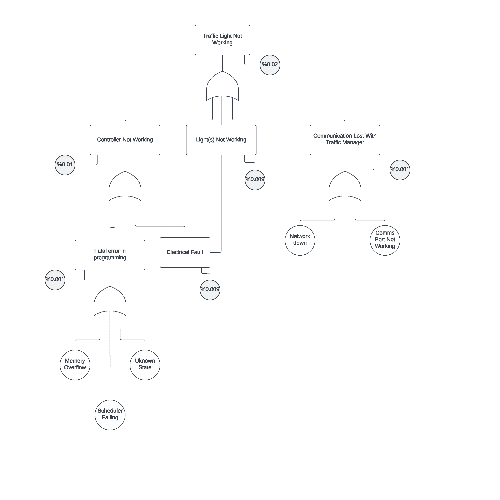


Figure 10, Fault tree diagram.

In this fault tree, the possibility of hardware and software failures that could result in the malfunctioning Traffic Light was analyzed. For a Traffic Light does not work, either its Controller, its Light(s) or its communication with the Traffic Manager would have to malfunction. Furthermore, the malfunction of a Controller could be traced back to either a software or electrical failure. For the malfunction of Light(s), only an electrical fault was identified. Lastly, an error within communication would either result from the network being down or the communication port of the Traffic Light not working.

As the case study had no actual hardware involved, the probability numbers assigned to the events are only meant for demonstration purposes and do not reflect real probabilities. The top probability of %0.02 is calculated from the union of the probabilities of its lower events; the intersection of the probabilities was neglected and considered to be 0 due to their insignificant value.

* 1. Collision in Intersection Fault Tree

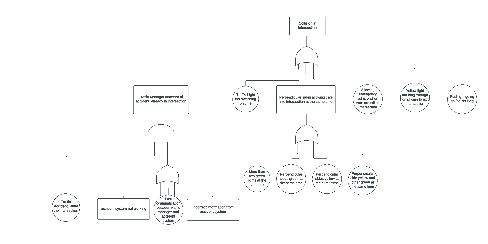


Figure 11, Fault tree diagram.

This fault tree includes the logical events that could result in a collision within the intersection. For example, an accident within the intersection and the malfunction of the Traffic Accident system would lead to a chain accident. Or, Traffic Lights not being switched on on time, allowing an emergency vehicle into the intersection while there is still traffic within the intersection, and much more. All these events could lead to a possible collision within the intersection and the specification of the model, and the implementation need to be checked against each event.

# Implementation

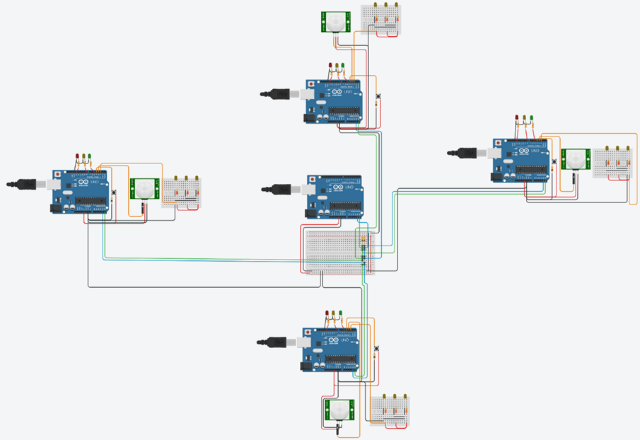
## Introduction

For the implementation, two use cases, the acceleration of emergency vehicles and the smart street lighting, were selected. The decision was made in favor of these two use cases because both run on one shared system and show the advantages of a smart city environment in an easily understandable way.

The topology of a simple traffic intersection with four arms was chosen, because it resembles the most universal example without any special boundary conditions. The goal was to simulate an intersection that controls the flow of the vehicles in a simple time-fixed manner with an ensured acceleration of emergency vehicles. The acceleration was realized by allowing a fast switch to a green signal for the direction in which the emergency vehicle is approaching and by holding this state until it passed the intersection. The smart lighting aspect was integrated by turning on the lights when a citizen that crosses the intersection is detected.

As the whole system should be simulated, a simulation environment for multiple embedded systems was needed. The online electronics simulation suite “TinkerCAD” offers a free and straightforward possibility of simulating prototypes of simple embedded systems that are based on Arduino microcontroller platforms. It was therefore used in the implementation.

Because of the earlier developed design, the whole system needed four Arduinos for the end nodes located at each arm and one Arduino acting as the Traffic Manager. To replicate the intersection and to make the working principle more understandable, the four end nodes were placed in the pattern of a cross while the Traffic Manager was placed in the middle.  
 As stated in the design, each end node consists out of the controller and peripherals connected to it. The peripherals of each end node in the selected use cases (traffic lights, street lights and cameras) were simulated by LEDs, a PIR sensor and a simple pushbutton.  
 As there was no possibility of simulating a camera detecting the approaching entities and their direction of movement, the assumption that the PIR sensor is only triggered by entities that are going to cross the intersection straight ahead, was made. The network connection between the nodes and the Traffic Manager was implemented by an I2C bus with the Traffic Manager acting as a master and the end nodes acting as slaves.

  
Figure n – Setup of the Simulation in TinkerCAD.

The actual implementation of the system required the implementation of two sub-systems, Traffic Manager and End Node, as those individual systems are running separate hardware platforms, and the functionalities of the overall system are distributed between them.

For the implementation of the Traffic Manager, the needed parts of the software were determined by identifying the needed functionalities from the design.

Ein Bild, das Text, Diagramm, Screenshot, Rechteck enthält.

Automatisch generierte Beschreibung

Figure n – Architecture of the chosen use cases.

Three main parts were identified as being necessary for the implementation of the Traffic Manager in the desired use case:

* Communication subsystem
* Logic controller for the traffic lights
* Logic controller for the streetlights

Because of the independently working functionalities for the traffic lights and the streetlights, the decision to implement them as independently working tasks was made. As the communication system is used by both tasks, it was considered as a shared resource and implemented as a third task.

To simulate the communication of the Traffic Manager with the Smart City Server and to add the possibility of observing the systems’ status, a fourth task was added.

The tasks’ periods were chosen by taking their functionality and the conception of the whole system into consideration. As the communication subsystem is needed to distribute the commands of the logic controller subsystems as fast as possible and poll the values of the sensors below the system’s timestep of 1s, a period of 100ms was chosen. Because the logic controllers only need to decide every second if there should be a switching command sent, the periods of these two tasks were set to 1s. As seen later, their computation time is negligible so having a slightly larger period than 1s has no negative impact on the system.

The period of the fourth task was set to 2s as it was found out to be sufficient to being informed about errors every two seconds.

## Traffic Manager

The proper execution of tasks needs a scheduler which the Arduino is lacking by default, so a simple RMS was implemented as a first step. Due to the low utilization of the CPU by the tasks and the limited technical possibilities of the Arduino language, the scheduler was implemented without the feature of preemption.  
It consists out of three main components: a struct called “Task” for defining tasks with their attributes, a task set made from an array of tasks and a method used for calling the scheduler.

### Struct “Task” and the task set

The struct “Task” contains five fields:

* id [byte]: used to give the task a unique ID.
* period [unsigned int]: used to define the tasks’ period in ms.
* executionTime [unsigned int]: used to store the measured duration of the tasks’ execution in us.
* lastExecutionTime [unsigned long]: used to store a timestamp of the tasks’ last execution since the start of the system in ms.
* taskFunction [function pointer]: function pointer to the function that is associated to the task.

As mentioned, the task set consists of an array from the type “Task”. It is used to store the information of each task and associates a function to be called when the task should be executed.

### Functions “runRMS” and “executeTask” of the scheduler

To work properly, the scheduler needs a function to be called frequently (ideally continuously) that manages and ensures the proper execution of the tasks. For this purpose, the function “runRMS” was implemented. It has a very simplistic working procedure that realizes the functionality of a Rate Monotonic Scheduler. When the function is called, it follows these steps:

1. Setting up an empty pointer of the type “Task”.
2. Getting the current timestamp.
3. Iterating over the task set:
   1. Calculating if the task needs to be executed.  
      If not: continue with the next iteration.
   2. Assigning the pointer to the task if the pointer is empty or this task has a shorter period than the task that is already assigned to the pointer.
4. Checking if there is a task assigned to the pointer, refreshing the tasks’ last execution timestamp, and executing it by calling the “executeTask” function.

The called function “executeTask” stores a timestamp before execution, calls the function that is referenced by the tasks’ function pointer, and gets a second timestamp to update information of the last duration of the execution.

As the function “runRMS” needs to be called as continuously as possible, it is the only function that gets called in the main loop. Its operation is only interrupted by the duration of the execution of the functions of the tasks. Because of the lack of preemption, the tasks needed to be designed as non-blocking and with the shortest possible execution time. This fact also removes the need of blocking mechanisms because there is no concurrent access of variables possible.

### Subsystems of the Traffic Manager

After the implementation and testing of the scheduler, the four tasks that provide the functionalities of the subsystems were implemented.

As all subsystems of the Traffic Manager deploy controlling commands to the connected End Nodes and therefore an overview of all End Nodes is needed, a struct called “EndNode” and a matching array were created prior to the implementation of the tasks. It consists of four fields:

* i2c\_address [byte]: contains the I2C address of the receiver.
* location [char]: char to specify the location of the End Node (e.g. ‘n’ for north).
* emergencyApproaching [bool]: flag that is set, when an approaching emergency vehicle was detected.
* citizenDetected [bool]: flag that is set, when a citizen that wants to cross the intersection straight ahead was detected.

**Task 1: communication subsystem**

Because of the periodic operation due to the chosen scheduling algorithm, the decision to implement a buffer for the messages to be transmitted was made. To encapsulate the data of the message, a struct called “Message” was created. It consists of three fields:

* receiver [byte]: contains the I2C of the receiver.
* message [String]: contains the actual content of the message.
* tx\_attempts [byte]: counter to keep track of the transmission attempts of the message.

The TX buffer consists of an array of the type “Message”. Its length was later set to 16 as there exist four nodes, the maximum number of TX attempts was set to two, and a maximum of two messages were queued by the traffic light controller at a time. The task itself is iterating over the TX buffer first while performing the following actions:

1. Checking if the current block is empty. If yes, continue with the next block.
2. Getting the I2C of the destination and beginning the transmission
3. Transmitting the message string
4. Ending the transmission and storing the result
5. Checking the result:

If the transmission was not successful and the number of the transmission attempts is below the threshold, the number of attempts would be incremented, and the iteration would continue with the next block (step 1).

If the transmission was not successful and the number of attempts is already above the threshold, an error message will be added to the error message buffer.

1. Deleting current block and continuing iteration (step 1).

After the iteration over the TX buffer, the task is iterating over the array of End nodes while performing these actions:

1. Requesting a message of the size of two bytes from the End Node and storing the size of the received data.
2. Checking the result and the size of the data waiting in the buffer of the I2C hardware module. If the size is unequal to two bytes, a message will be added to the error message buffer and the iteration will continue with the next End Node (step 7).
3. The two received bytes containing the value of ‘1’ if an emergency vehicle was detected and if a citizen was detected are read into two separate variables of type “char”.
4. The value of the variables is checked for being equal to ‘0’ or ‘1’, converted to a bool, and stored in the fields “emergencyApproaching” or “citizenDetected” of the current End Node. If the value diverges from ‘0’ or ‘1’, an error is added to the error message buffer. Continuing with the next iteration (step 7).

**Task 2: logic controller for the traffic lights**

The logic controller of the traffic lights was implemented by a simple Finite State Machine with guarded transitions that rely on a simple counter to allow timed transitions. The timer is made by incrementing a variable every time the task is executed. As the tasks’ period is 1s, the counter is incremented every second and therefore provides the possibility of executing timed actions at a resolution of 1s. Since the accuracy strictly depends on the execution of the task on time, it could be considered a rather crude way of realizing a timer. In this simulation, the measured deviation was way below 1ms, caused by the lightweight implementation of the scheduler and the low utilization of the CPU, because of this it was considered as being sufficient.

The state machine itself consists of 11 states to realize the switching sequence of the intersection. The state is encoded numerically and stored in a variable of the type “byte”. The switching logic is implemented by using a switch-case statement. After changing a state, the counter for implementing the timer gets resetted to 0 to prevent an overflow. The states make use of the function “addPacketTX” to send the lamps of the traffic light that should be switched on to the right End Node.

To support the acceleration of emergency vehicles, two variables were introduced in the relevant guarded transitions of the state machine. The variables are of the type “char” and contain the location of the End Node that received a signal with an emergency vehicle approaching. The status of the flag of all End Nodes is checked when the task is called and when a flag is found to be set, the location is written to the variable. In the state machine, the variable is checked if it contains a character to allow a direct switching to the next state or prevent switching to the next state.

The variable is reset when the flag of End Node that is located in the opposite arm of the intersection is set (e.g. direction north can reset the flag of direction south and vice versa).

**Task 3: Printing the systems’ status**

The third task checks if errors were reported by the other subsystems and informs the user if any errors occurred by printing them via the serial interface.

Its working principle is similar to the communication subsystems’ task. A buffer of the type “String” contains error messages that are added by the other subsystems using the function “addErrorMsg”. During the execution of the task, an iteration over all blocks of the buffer is done and if a block contains an error message, it will get printed out via the serial interface and erased afterward to free the block.

**Task 4: logic controller of the streetlights**

To allow the logic controller for the streetlights to turn on the streetlights of an arm for 15s, four timers were used. These timers are triggered when the arm of the opposite direction recognizes a citizen that wants to cross the intersection in a straight way. Just as in task 2, the 1s periodicity of the task was utilized to create a timer using a variable that gets decremented at every execution of the task.

The task itself is iterating over all timer variables performing these actions:

1. Checking if the variable has a value greater than or equal to 1. If not, continuing with the next timer (step 1).
2. Decrementing the variable.
3. Checking if the variable has a value that is equal to 0.

If yes, adding a packet to the message queue via the function “addPacketTX” to turn off the corresponding streetlights. Continuing with the iteration (step 1).

After the iteration over all timer variables, an iteration over all End Nodes is performed while executing the following actions:

1. Checking if the flag that a citizen was detected is set.

If not, continue with the next End Node (step 4).

1. Checking the location of the End Node with the set flag.

Adding a packet to the message queue via the function “addPacketTX” with the destination of the opposite End Node to turn on the streetlights of this location.

1. Setting the corresponding timer to 15s and continuing with the iteration (step 4)

This rather simplistic approach ensures that the streetlights in the direction that a detected citizen will continue its journey are turned on for enough time and turned off afterward to save energy.

## WCET Analysis

To calculate the WCET of the implementation, a flow analysis was carried out. Because of the limitations of the simulation environment, TinkerCAD, analysis of the binary executable was not possible. Therefore, a mixture of static and dynamic analysis of the implementation flow was considered as the basis for the WCET. In the flow analysis, naively the worst scenario in each task was considered, i.e. the inputs were the worst (lengthiest) ones possible that would take more time to execute. Many loops are never iterated as much, and inputs would not include such values in a real scenario. Furthermore, the execution time for simple assignments like if-statements and variable declarations and assignments was ignored due to their negligible execution time. For this analysis, the execution time of I/O operations was considered, as they caused the longest execution times.

### Task 1: Communication

This task contains the most I/O operations and was the best candidate for the WCET analysis. This task contains 2 for-loops that will decide the execution times. The first one iterates over the *rx\_buff* array where *Message* objects are sent over the *i2c* port to their receiver. The array has a size equal to the constant *TX\_BUFF\_SIZE* which was 20 in the version that was considered for this analysis. Normally, the buffer contains a total of 4 messages to be sent. For the analysis, it was assumed that the array was full and a total of 20 messages needed to be sent; this equals 20 loop iterations. Furthermore, the size of the message itself needed to be considered. For this, the lengthiest message in the system (the 4-char “loff” message) was considered.



Figure 12 - Analysis of the first loop in task one.

The i2c bus can transmit 100 kbit/s in standard mode. But since for the implementation the *Wire* library was used and this library adds some overheads of its own, for the transfer time, the execution time was dynamically managed; Arduino’s *micros()* method was used to determine the time between the sending of each message. This amounted to a total of 255 microseconds for each byte. Finally, the worst execution time for the first loop can be calculated using the following formula:

Since each *char* in the C programming language equals one byte, our lengthiest message would be 4 bytes. Taking into this, the 20 iterations for the loop, and the dynamically calculated time for transmitting each byte using the Wire library, we will get the following execution time:

For the second loop, the end nodes, i.e. traffic lights, are iterated. It can easily be assumed that in the worst case, the loop will have 4 iterations. Unlike the last loop, in this one, bytes are read from the i2c ports. The reading time is done at the same rate as writing for the i2c bus, but we used the same method as the above to calculate the time needed to read each byte, which equaled 331 microseconds per byte. Furthermore, a total of 2 bytes are read in the loop. The worst execution time can then be calculated:

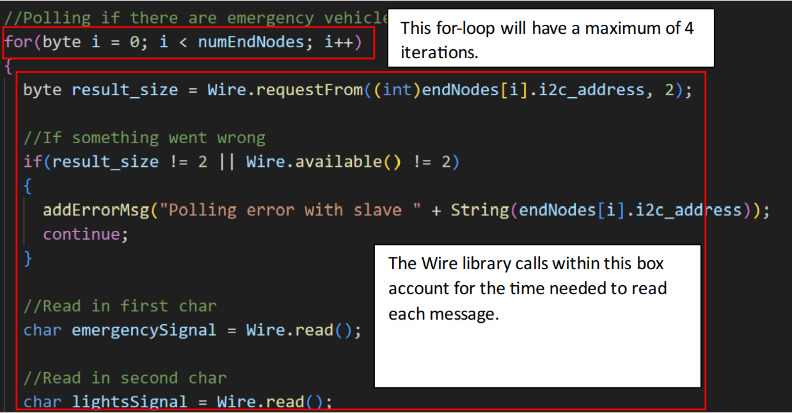


Figure 13 - The second loop of task one.

Finally, the worst execution time of this task can be calculated by summing up the numbers above:

### Task 2: Logic

For this task, no I/O like the last task is done. Also, the task only contains one direct loop and more loops within the function calls. But most of the statements of this task are simple assignment or comparison statements that can safely be ignored. The only costly statement in this task is the *Serial.println*()statement. The execution of this statement was also dynamically calculated similar to the statements in the previous task using the *macros()* method. The result for a 100-character string was 500 microseconds. In this task, the serial method will only be called once, resulting in the following:

### Task 3: Logic

This task contains a for-loop which iterates over the *error\_msg\_buff* with the maximum size of *ERROR\_MSG\_BUFFER\_SIZE* which equals 20 at the time of writing this analysis. Therefore, a worst-case of 20 iterations can be assumed. Within the loop, the *Serial.println*() is called. The execution time for the loop can then be calculated as such:

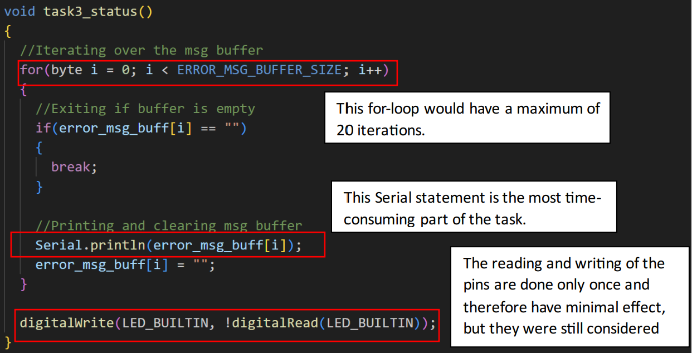


Figure 14 - Analysis of task three.

The task also includes one digital write and one digital read method which take 4.84 microseconds and 9.13 microseconds respectively. In total, this task will have the following execution time in the worst case:

### Task 4: Lights

No outstanding statements are included in this method and therefore there is no basis for calculating its WCET. Nevertheless, the worst execution time of the task given by the scheduler in a normal run, 212 microseconds, can be considered as its WCET:

1. Schedulability

With the WCET of the tasks available, the schedulability of the system can easily be tested. As mentioned, each task has a suitable period where it needs to run at to fulfill its responsibilities. By calculating the last upper bound of our rate monotonic scheduler, we can check the feasibility of our defined schedule. Considering task one with a computation time of 23.048 milliseconds and a period of 100 milliseconds, task two with a computation time of 0.5 milliseconds and a period of 1000 milliseconds, task three with a computation time of 10.00937 milliseconds and a period of 2000 milliseconds, and task four with a computation time of 0.212 milliseconds and a period of 1000 milliseconds, the following least upper bound can be calculated:

The least upper bound for our system equals to about %34 CPU utilization which is easily below the %69 threshold. As a result, our set of tasks are schedulable.

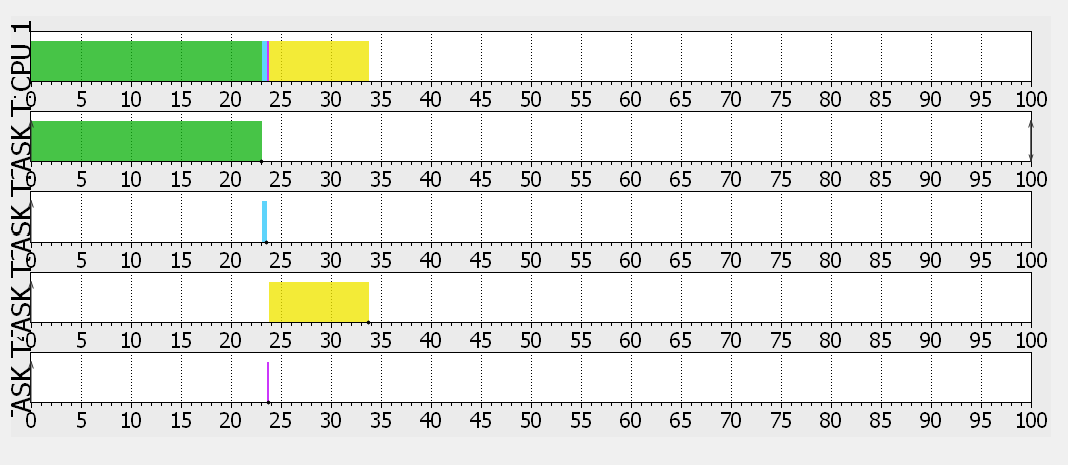


Figure 15 - Graphical representation of the execution of the tasks (generated using SimSo).

## Traffic Node

The implementation of the traffic node or end node is made from the following parts (which are shown in Figure 14):

1. The LED Lights represent the traffic lights signal.
2. A digital button used to send the “Emergency Vehicle Is Approaching” Signal
3. A PIR sensor for checking if their cars are waiting on road of the node.
4. LED lights represent streetlights.
5. The Arduino which connects to all mentioned parts and the Traffic Manager

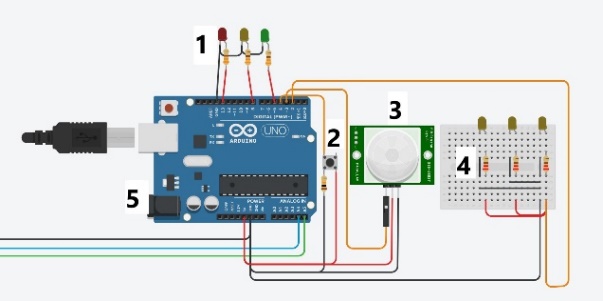
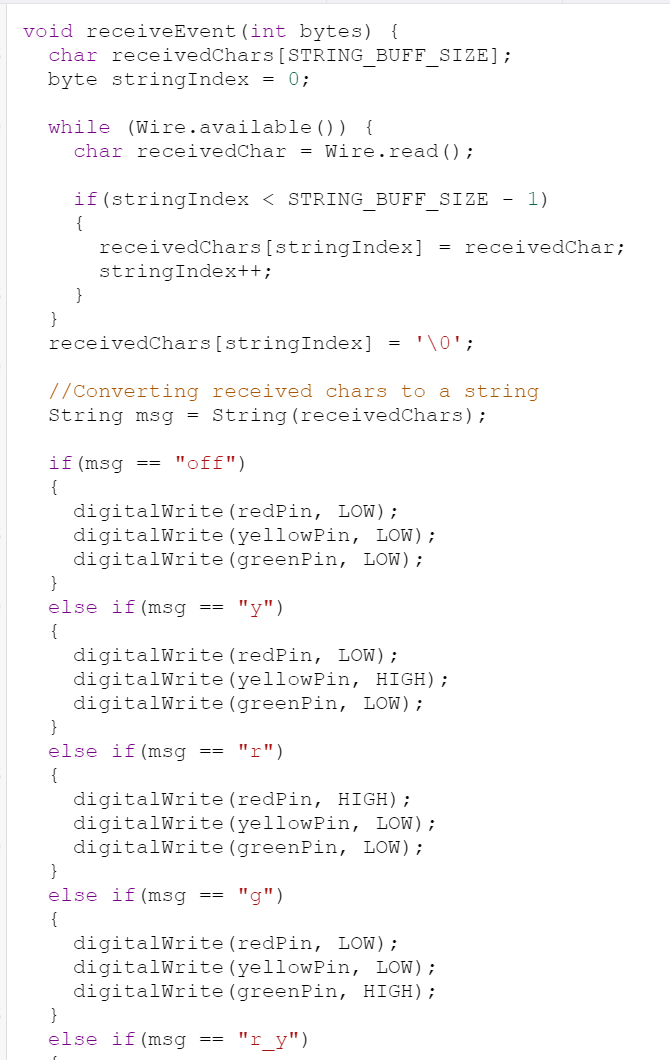


Figure 14 – Circuit Diagram for Traffic Node.

The tasks of the traffic node can be summarized into the following:

### Receiving Traffic Light Signal From Manager

The traffic node receives the signal from the traffic manager to turn on the intended traffic light.

Figure 15 – The “receiveEvent” function part 1.

### Receiving the Street Lights Signal From Manager

The traffic node receives the signal from the traffic manager to turn ON or OFF the streetlights.

A screenshot of a computer program

Description automatically generated

Figure 16 – The “receiveEvent” function part 2.

### Send “Motion Detected” Signal To Manager

When the PIR sensor detects there is movement, the node sends the “Motion Detected” signal to the manager to handle.

A white screen with black text

Description automatically generated

Figure 17 – The “sendRequestedSignals” function

### Send “Emergency Detected” Signal To Manager

When the digital button is pressed, the node sends the “Emergency Vehicle Approaching” signal to the manager to handle it (check Figure 17)

## Testing

For the testing the implementation we had 2 test environments

### Unite test using C++ code

To test specific functions in the code, we created an automated testing environment using the C++ with the “assert.h” library. The testing environment was setup as follows:

1. We defined the function that is to be tested in a header file “function.h”

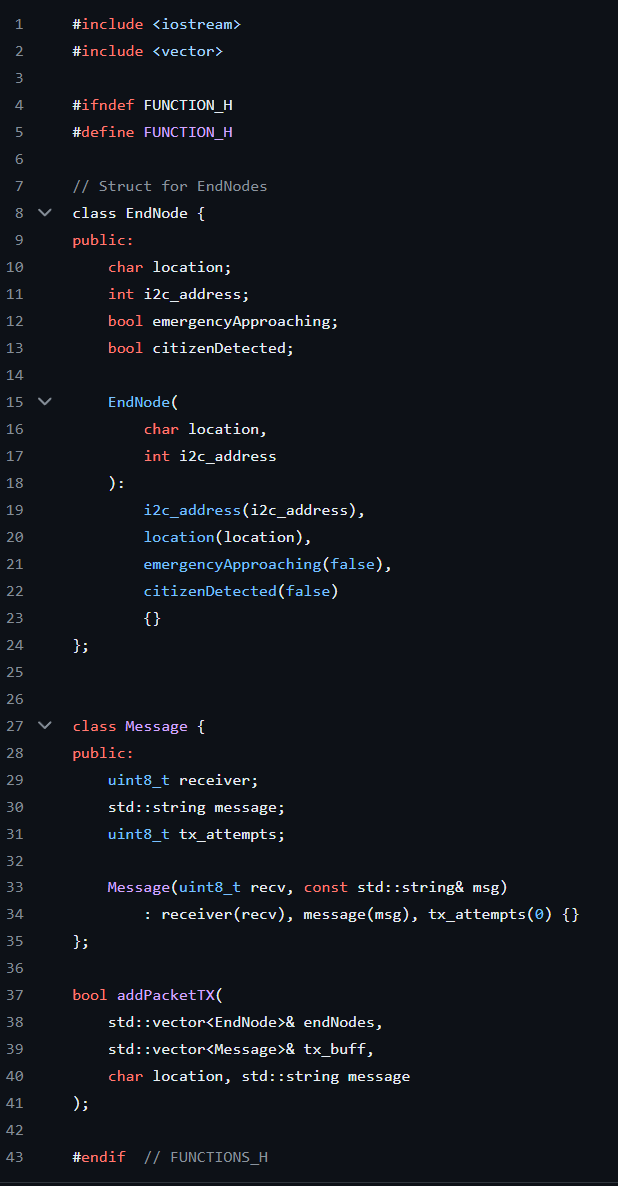


Figure 18 – defining a “function.h” file for unit testing environment.

1. We created the function inside a “function.cpp” file. The functions were created using the regular C++ data types and libraries before being transformed into Arduino functions.



Figure 19 – defining a “function.cpp” file for unit testing environment.

1. Then we created a “test.cpp” file we tested the expected output for both valid and invalid inputs

A screenshot of a computer program

Description automatically generated

Figure 20 – defining a “test.cpp” file for unit testing environment.

1. We than created a “run.cmd” for automating the needed script for running the tests

A screen shot of a computer program

Description automatically generated

Figure 21 - defining a “run.cmd” file for unit testing environment

The process above was done for multiple functions inside Traffic Manager and the Traffic Node

A screenshot of a computer

Description automatically generated

Figure 22 – The unit test hierarchy

The testing functions were:

* The “addPacketTX” function inside the traffic manager
* The “Task1” function inside the manager, which was split into 2-unit tests. One for testing the function of reading the signal, and for testing the functionality for handling the signal.
* Test the “receiveEvent” function inside the end node.

### Integration Testing using Tinkercard

Using tinkercard, we create a manual testing environment that covers.

* The code
* The I2C connection between the end nodes and the manager
* The connection between the end node and its parts

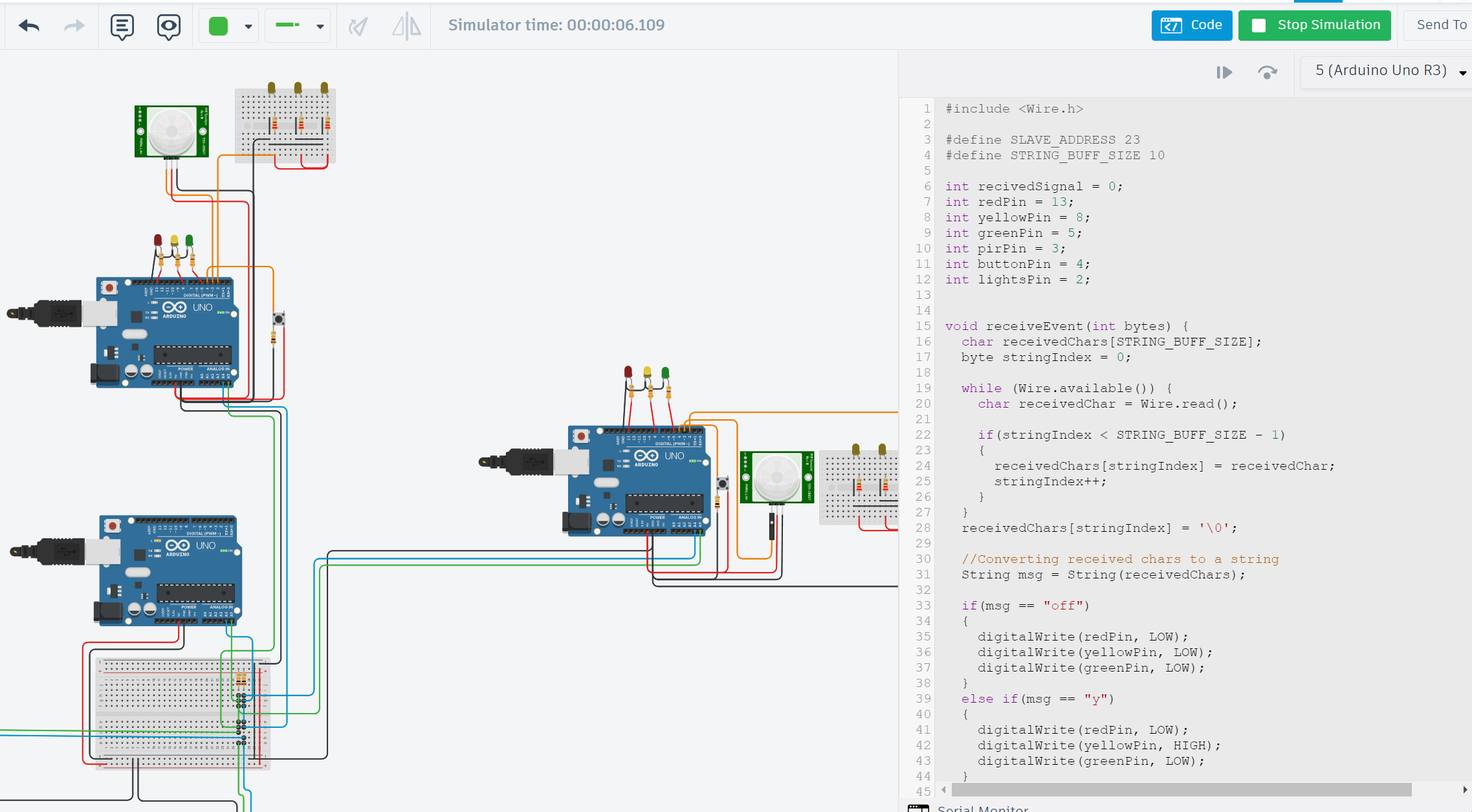


Figure 23 – Tinkercard testing environment.

# Conclusion

# Reference

# Appendix

You can find the implementation of this approach to a smart city on [GitHub](https://github.com/allerter/ESE-Smart-City). The GitHub repository consists of 4 main parts.

* 01\_Documents
  + 01\_Presentations
    - 01\_PowerPoint
    - 02\_Trash
  + 02\_Paper\_Final
  + 03\_ProjectSpecifications
* 02\_Implementation
  + 01\_EndNode
  + 02\_TrafficManager
* 03\_Requirements
  + 01\_Diagrams
    - 01\_SmartCity
    - 02\_Trash
  + 02\_Trash
* 04\_Tests

In this format you will be able to find an organized structure for all documentation, implementation and design of the system utilizing enterprise architect. In the corresponding files there can also be found a “Trash” folder. This folder is utilized to discard items, as the project was in constant development. Simple deletion of files was not the appropriate way of cleaning.

Commits:

A screenshot of a computer

Description automatically generated

Figure 16, Summary workspace.

A screenshot of a computer

Description automatically generated

Figure 17, Total amount of code lines.

A screenshot of a computer

Description automatically generated

Figure 18, Code lines per folder.

Workload distribution.

Prototype Implementation (**50%**):

* Leander Hackmann (**25%**)
* Mohammed Al Salihi (**25%**)

Analysis and Design (**50%**):

* Luis Fernando Rodriguez Gutierrez (**25%**)
* Hazhir Amiri (**25%**)

# Affidavit

We Luis Fernando Rodriguez Gutierrez, Hazhir Amiri, Mohammed Al Salihi, Leander Hackmann herewith declare that we have composed the present paper and work our self's and without use of any other than the cited sources and aids. Sentences or parts of sentences quoted literally are marked as such; other references with regard to the statement and scope are indicated by full details of the publications concerned. The paper and work in the same or similar form has not been submitted to any examination body and has not been published. This paper was not yet, even in part, used in another examination or as a course performance.