# Cross-Traffic Management System

**Anaedu Ukamaka Akumili**
*Electronics Engineering Department*
ukamaka-akumili.anaedu@stud.hshl.de)
Lippstadt. Germany
ukamaka-akumili.anaedu@stud.hshl.de

**Doula Shihab Ud**
*Electronics Engineering Department*
Hochschule Hamm-Lippstadt
Lippstadt. Germany
shihab-ud.doula@stud.hshl.de

**Unanma Justice Chisom**
*Electronics Engineering Department*
Hochschule Hamm-Lippstadt
Lippstadt. Germany
justice-chisom.unanma@stud.hshl.de

**Sali Emirkan**
*Electronics Engineering Department*
Hochschule Hamm-Lippstadt
Lippstadt, Germany
emirkan.sali@stud.hshl.de

***Abstract*—A cross-traffic management system without traffic lights for autonomous vehicles aims at improving the efficiency and safety of vehicles at road intersections. The system depends on wireless communication and coordination among connected and autonomous vehicles to eliminate conflicts and cross the intersection without stopping. Conflict zone resolution and Route Planning are the main focus of this proposed system. Conflict zone resolution handles the optimal passing order of Connected Vehicles based on their arrival time, speed, acceleration, and Type of Car(Government and Emergency Services). Route planning ensures each autonomous vehicle can move without crashing into each other and it also decides the best path for the cars to move based on their direction. The system can handle the movement of cars at the road intersections, such as four-way, three-way, or direct. The system can also adapt to different traffic situations in case of emergency services or police vans.**

***Index Terms*—connected and autonomous vehicles (CAVs),Vehicle-to-vehicle (V2V), vehicle-to-infrastructure (V2I) communication**

## I. Motivation

The idea behind this research work is to find a solution between autonomous vehicles and the cross-traffic management system. In this paperwork, we will focus on how to effectively manage the traffic to avoid collusion or interference among the vehicles. To achieve this we will need to implement the queuing method as well as the first in first out(FIFO) scheduling algorithm. Several researches have been done on this topic but we also came up with our idea to make our system functional and allow all the vehicles to cross the road without damage or hindrance to each other. Using the Hardware/Software design approach, we will implement our design in VHDL to show how we intend to move the vehicles at the cross traffic.

## II. Introduction

As the years go by, more vehicles have been manufactured which has dramatically increased the number of vehicles on the roads. This has led to some serious problems such as traffic jams, accidents, and many other dangerous problems. Advancement in technology has seen cities turn into smart cities. Moreover, In recent years, an increased number of vehicles caused traffic jams which have become one of the main challenges facing engineers and road designers because of the need to create an intelligent cross-traffic management system that can effectively control and minimize the traffic on the road intersections. It is evident how huge Information and communication technologies have contributed in the development of cities as well as the automotive sectors. Traditionally the traffic lights controlled the traffic flow but we have seen many research on how connected vehicles can communicate with each other at crossroads and with the control of an infrastructure, all vehicles can smoothly cross without any collision. This is enabled by wireless sensor networks, IoT, data sharing, and cloud computing.

The organization of this paper is as follows

- Problem Statement
- Background and Definition
- Solution Approach
- Implementation and Simulation

## III. Problem Statement

In this section, we will discuss the problems encountered at Crossroads and how our intended system will solve them. One of the problems is making sure that the cars move in their directions without colliding with each other, ensuring the safety of all road users. Additionally, we hope to minimize the delay of the cars in the queue and reduce congestion.

## IV. Literature Review

This section is about the existing literature on the cross-road traffic management system. The aim of this is to provide in-depth knowledge of the topic as well as serve as a basis for future analysis of our work.

### A. First come first server (FCFS)

Because it takes a fairness-based approach, FCFS is a recommended option for reservation-based control. As the name implies, it is simply based on a first-come, first-served basis. The centralized approach used in this work provides entire vehicle information to the junction manager. It is aware of the specified entry times and destinations for all cars desiring to enter the junction, as well as the next available tile slot for each vehicle. It is important to keep in mind that cars do not always adhere to FCFS at genuine intersections with non-conflicting streams.

### B. Queuing theory

This method of estimating queuing behaviors is based on a set of constraints and presumptions. For a number of servers, it is presumable that there are inputs for the arrival distribution and service time distribution. Queuing theory directly leads to the evaluation of efficiency or operational characteristic that measures the performance of a queuing system. You may easily use the formulas from queuing theory to predict the long-term behavior (also known as the steady state condition) of the queuing system in terms of its performance if your queuing system distribution complies with the theoretical presumptions. In consideration, if you could figure out how many servers you'll need or how long it would take to meet your demand. The cost of queuing can also be estimated and experimentation with certain ideas is a due cost to enhance the queuing system.

## V. Solution Approach

Autonomous Vehicle Communication establishes a communication network between autonomous vehicles and the traffic management system. This enables vehicles to share their real-time information, such as location, speed, and intentions, with the system. The system can use this information to optimize traffic flow and make informed decisions. The M/M/1 strategy, with one server in charge of the queuing system, is the one we chose to employ for our system. These are the details of this technique: When a car approaches an intersection, it sends the direction it wants to travel in, which is determined by where the car is coming from and where it is headed (for example, "NE" if the car is coming from the North and heading east). The area where all the cars arriving from all directions are intended to cross has now been formed as a table slot. Arriving vehicles must wait until a slot becomes available if they want one that is not open. After the slot has been freed, the car can enter the grid, pass through until its final destination, and exit the system. This action is repeated in parallel with the movement of another car that is also requesting a slot inside the system. Once a vehicle has finished crossing the system, it returns a message meaning it successfully reached its target.

## VI. System Diagrams

this are the diagrams views of the system. use case, block diagram, state machine and sequence diagrams.

### A. Use case diagram

The essential players, roles, and classes of our system's design are shown in this diagram along with more visual information using a unified modeling language. The vehicle is able to provide movement data and input to the cross traffic system in the use cases that follow. A cloud server also stores the directions that the car receives from infrastructure.

### B. State machine

For the state machine, there are have four initial states which we identified as our first bit (direction). These First bits are the entrances into the cross intersection. The Final state of the bit shows the exit state of our system. As seen, if the slot is free, and the second bit is true, the vehicle will be permitted to drive, otherwise it waits and checks again.

### C. Block diagram

The cross-traffic management system's components and their relationships are graphically represented in a block diagram. It aids stakeholders in comprehending the system's overall structure and organization, including the many sub-systems, modules, and linkages between them. Stakeholders can recognize and define the various elements involved in cross-traffic control using the block diagram. This contains all necessary components, including as sensors, controllers, communication interfaces, and decision-making algorithms.

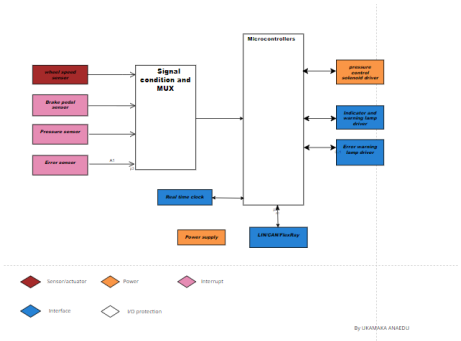


Fig. 1. Block Diagram

### D. Sequence diagram

Our main objectives for this project were the creation of a cross-section for traffic system congestion and a communication channel between two autonomous vehicles. Stable network communication and an infrastructure that aids in connection maintenance were required for both our connection and environment. The sequence diagram that follows shows us exactly how our series was created from the beginning. We have considered a scenario in which there will be automobiles in a lane and a cross section that communicate with one another using sensors and cutting-edge communicators. For instance, vehicle 1 needs a reliable connection to the infrastructure server to verify a connection with vehicle 2 is possible. Additionally, the communication server times out if the connection breaks down in some way. Now, the sensors

assist in making turns when a vehicle needs to turn left or right in a cross section by making requests to the main infrastructure server. Additionally, if the sensors malfunction, the vehicles in the lane stop until the connection is stored, at which point the FIFS Queue system in that line causes them to restart.

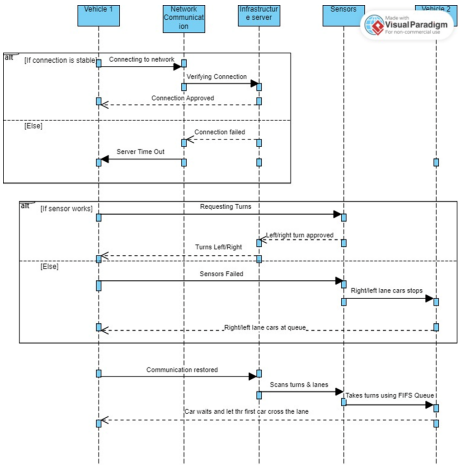


Fig. 2. Sequence Diagram

## VII. System concept and problem concept

For the implementation of the UPPAAL model and the implementation in C++, we first modeled the problem and the system in an abstracted way. In our model, we assumed the traffic cross section to consist of one lane per direction, which would give us a total of 4 lanes, considering the cars can approach and leave in 4 directions. We gave each lane a designated letter as a label, starting from lane 'A' to D. The most important aspect is, of course, the cross-section itself; therefore, we divided the cross-section into four blocks that are named according to the lanes that are crossing in that specific block. This approach was chosen because cars in the real world that cross a cross-section always occupy at least one of these blocks. Depending on the action of the car, one or more blocks can be left free and therefore used by another car. With this approach, we can feed the system the necessary information about the current state of the cross-section. The cross-section model is visualized in Figure 3.

Furthermore, for our system model, we made the decision to design the queue with First Come, First Serve (FCFS) in mind, which would describe the behavior of the real-world equivalent of cross-sections without lights in the best way possible. However, as it is important to include scenarios for emergency situations or just cars with higher priorities, we also introduced a priority system with three levels:

1) This priority is assigned to every normal car in the system that does not have an emergency situation or a higher priority of some sort. Every car with this priority will simply be queued according to FCFS.
2) The second highest priority is assigned to cars that have a higher base priority than normal cars, e.g., users paying for higher priority or important businessmen, etc.; they will always be queued in front of cars with the first priority setting. If more cars follow this priority approach, they will be handled according to FCFS again.
3) The highest priority is assigned to cars in emergency situations, such as police cars or ambulances. They will always be queued first, and every other car needs to make space for them to pass the cross-section to ensure their timely arrival.

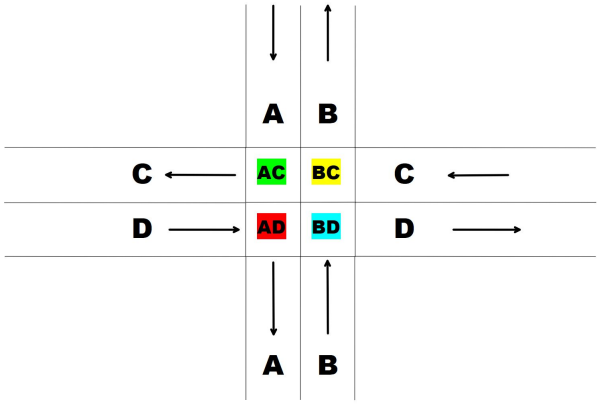


Fig. 3. Model of the cross-section with information about location and blocks

## VIII. UPPAAL Model

Based on our System concept and considering the problem concept, I created an UPPAAL model. The model consists of two main templates:

1) The car model, which includes all the actions and behavior of the car in a clocked manner. The number of cars is based on the integer variable 'N' in the declarations of the system. For our showcase of the model, it is kept to 3 cars.
2) The system, which communicates with the cars, gives commands to the cars, and handles the queuing to ensure the functionality of the whole system.

### A. Initial Info about the car model

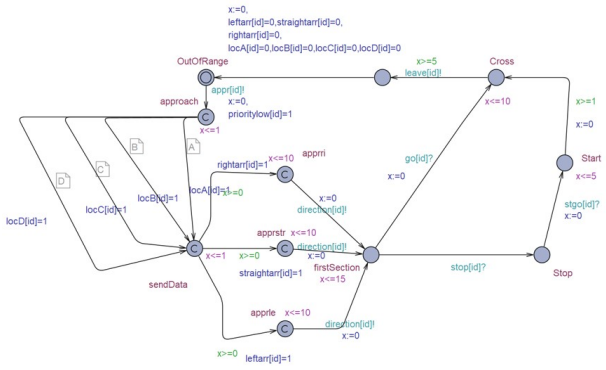


Fig. 4. UPPAAL Model of the cars in the system

The car model, as visualized in figure 4, starts in the 'OutOfRange' state, which, as the name indicates, means that

the car is still out of range but will eventually move on to its next state, the 'approach' state. In the transition, the specific car communicates with the system, giving it a signal that it is approaching with an identifier for the car and the priority of the car. In the UPPAAL model of the project, the priority is set to 1 by default. The next states of the car model in UPPAAl until before the 'first section' state do not consume any time in a real-world application since they exist here to assign the car an initial location A-D and the action it wants to take at the cross-section. This information is saved in Arrays that are created for each car. The Nth point in the array corresponds to the Nth car in the system and is true when the value is 1 and false when it is 0. The arrays are visualized in Figure 5.

```
const int N=3;

int blocks[4];

int leftarr[N];
int rightarr[N];
int straightarr[N];

int prioritylow[N];
int prioritymid[N];
int priorityhigh[N];

int locA[N];
int locB[N];
int locC[N];
int locD[N];
```

Fig. 5. Code for storing the information about the cars and cross-section in arrays

With 'direction[id]!, the corresponding car's location and action information will be transmitted to the centralized system. From there on, the car will await a response from the system, so it can either cross the section or stop.

### B. Initial Info about the System Model

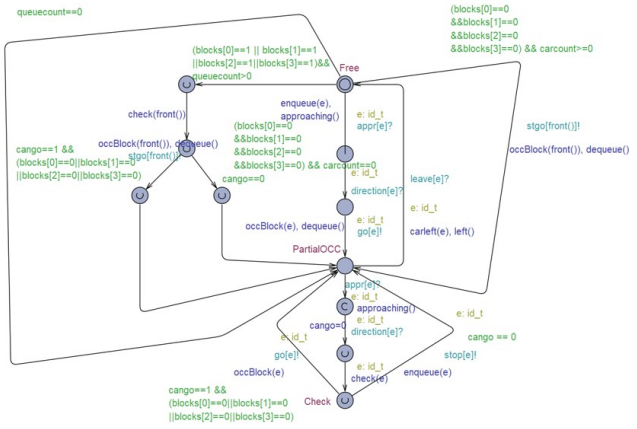


Fig. 6. UPPAAL Model of the cross-section System

The UPPAAL model for the system also has information about the current state of the cross-section saved in the array 'blocks' as seen in figure 5. Therefore, it starts in the 'Free' state and moves on to the next states directly vertical to it when a car is approaching. The other two possible transitions are only possible when cars are already inside the section before. Since the first car that is approaching has nothing to worry about the cross-section being occupied, it can immediately receive the signal to cross the section without further check. As long as the car is then crossing the section, the state of the system will remain at 'OCC', meaning that the section is occupied. The possible scenarios here are another car approaching or a different car leaving the cross section.

### C. System model and Car model interactions

When a car is approaching the section while another car is crossing or has received the signal to cross the section, the next approaching car will trigger the system to check for a collision between the two cars if it also crosses the system. Depending on the actions and locations of the cars, the approaching car will either receive the command to cross the section or to stop when a conflict is detected. This is done in the transition to the 'Check' state of the System model, in which the identifier of the approaching car is fed to a function that will determine if the car can cross the section based on the 'Blocks' array. If the car receives the signal to stop, it will be enqueued in the FCFS queue and will wait until its path through the section is clear.

Whenever a car leaves the section, the system will check the queue and choose the next car in the queue to perform a check to see if it can cross the section. If yes, it will receive the command to start again and cross the section.

### D. UPPAAL Model Conclusion

Finally, this UPPAAl model serves as an abstracted, time-constrained implementation of our system. Verifying the model, no deadlocks could be found in the system with a car count of three, as shown in figure 7. A check for deadlocks with more cars inside the system resulted in excessive RAM usage before finishing and could therefore not be performed. This is due to the huge number of possible states of the UPPAAL model that have to be saved in fast memory during verification.


Fig. 7. UPPAAL model verified of having no deadlocks

## IX. Implementation in C++

As a first abstract implementation, the C programming language seemed the best fit because it was already used and ready in the UPPAAL model, but because of the possibility of creating a class for the cars in the system in C++, the implementation was done in that programming language at last.

The goal of this implementation was to simulate the behavior of the system in the UPPAAL model and of our

general concept by using the output terminal. Furthermore, it was important to create a basis for the Hardware/Software Codesign part with the functions in C++.

### A. Set-Up Code section

As a set-up for the implementation in the main loop, the necessary variables (Arrays, bools, etc.) for the program are declared. The car class is initialised with different paramaters as seen in figure 8.

Each entity of the car class receives:

- an identifier 'id'
- a 'priority'
- a 'timing' variable, representing the passed time for the car in the system.
- a string for its 'location', 'action' and 'status'

All parameters aside from the identifier, status and timing variable are assigned randomly among a few specified options.

```
class Car {
private:
    int id;
    int priority;
    int timing; // timing for cars
    std::string location;
    std::string action;
    std::string status;
```

Fig. 8. Section of the C++ Code in which the private attributes of the car class are initialised

Finally, several functions for printing out car info, setting car info, queueing, dequeueing and more are declared before the main loop.

### B. Behavior of the Code

The main loop functions with a switch case statement with a total of 2 different states:

1) Free state
2) Occupied state

The 2 states represent the states of our concept and the UPPAAL model and function similarly. In the free state, a car can either approach or a new car can spawn into the system. This is determined by a random event in the code. After a car approaches, it will receive the signal to cross the section, because it is not occupied. After that, we are in an 'occupied' state. There another car can either approach or time simply pass, which eventually leads to the first car leaving the cross-section again. Timing advances are being checked whenever the one-loop cycle completes. Approaching cars will be enqueued according to their priority. They will receive a command to either stop or cross after a check has been performed, just like in the UPPAAL model. Finally, every event, time advance, queue change etc. is displayed in textform in the output terminal. An example is shown in Figure 9

```
timing of car '1' = 4

Dequeuing

New Status: Out of Range
Blocks array =0
0
1
0
Car '1' has left the cross section
```

Fig. 9. Output of the C++ implementation showing a scenario where a car leaves the cross-section after crossing it

## X. Implementation and simulation

For the hardware simulation, we used the VHDL for the synthesis of our system. VHDL is used for system design, modeling, and verification. The latest version of ModelSim is 2021.4. the software we used is ModelSim which supports VHDL. The latest version of VHDL is VHDL 2019.

```
use IEEE.STD_LOGIC_1164.ALL;

entity Traffic is
        Port (
                Direction: in std_logic_vector (3 downto 0);
                slot: in std_logic
        );
end Traffic;
    -- North is "00"
    -- West is "01"
    -- South is "10"
    -- East is "11"
architecture behavior of Traffic is
        type t_state is (NW, NE, SE, SW);
        signal state: t_state;
        signal FirstBit:  std_logic_vector (1 downto 0);
        signal SecondBit: std_logic_vector (1 downto 0);
```

Fig. 10. Declaration of entity

In the code above, we created an entity called Traffic and also declared the port. The direction is an input for the position of the cars at the intersection, the slots represent the presence of cars and it takes a single value of 0 or 1. We also declared the behavior of our system such that we intend to exit a state if two directions are the same, for example, our system is expected to do nothing if the Direction is North-North or East-East. Also If the second bit of the direction is different from the second bit of the state and if the next slot is "0" then we can proceed to the next state. Otherwise, the state will remain in the same position.This is done to check for safety at the intersection to make sure that cars are safe on different points of location.

```
if (Direction /= "0000" and Direction /= "1111")
then  --no action is expected if the directions
are either Nort-North or East-East
case State is
when NW =>
if (SecondBit /= "01" and slot = '1') then
-- Check if the second bit of the direction is
different from "01" and
there is a car waiting
State <= SW; -- Go to the next state
else
State <= NW;
      -- Otherwise remain in the state
end if;
if (SecondBit = "01") then
```

```
--If the second bit of the direction is the same
as the second bit of the state.
report "Exit successful";
end if;
```

From the snippet code above, we can check for directions for the cars and change the state based on the location and presence of a car at the slot. we used the case statements to simplify the conditions on which the cars should move to avoid collision and to minimize congestion. There are cases for the South-East, North-West, South-West, and North-East. Below is the wave simulation of our system.

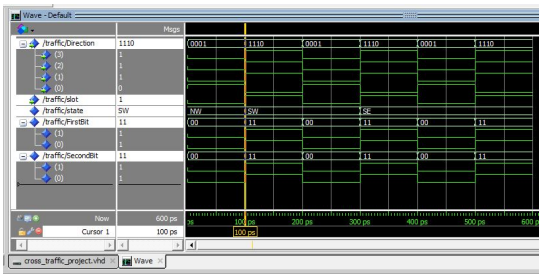

Fig. 11. Simulation wave of our traffic system

## XI. FreeRTOS Implementation

Multiple jobs frequently need to share constrained resources, such as I/O devices, memory, or communication channels, in embedded systems. If not appropriately managed, concurrent access to various resources may result in conflicts, data corruption, or inconsistencies. The system where Tasks can request and release semaphores, acquire and release mutexes, and send and receive messages through queues is made safe and well-coordinated by FreeRTOS. These techniques make it possible for jobs to coordinate their operations, prevent conflicts, and reliably exchange data.

Our project's goal was to develop a system for efficient cross-traffic management employing autonomous vehicles and queues. employing C code and FreeRTOS to lessen traffic congestion. We decided to include FreeRTOS in our project to do this. And because Arduino esp32 is a capable piece of hardware with FreeRTOS support, is well known for its multitasking and concurrency, and is still simple to develop for in the Arduino environment, we chose to use it.

### A. Inclusion of FreeRTOS headers

To implement FreeRTOS, we modified the code by including the necessary headers and initializing the FreeRTOS kernel. We then created tasks, which are individual units of work that can run concurrently.

### B. Creation of task

Here, task1 and task2 are the two tasks created using the xTaskCreatePinnedToCore function. These tasks will execute concurrently and independently. Task 1 handles printing car information, and Task 2 handles setting status and timing. The tasks are then pinned to separate cores for parallel execution.

```
1  #include <Arduino.h>
2  #include <vector>
3  #include <algorithm>
4  #include "freertos/FreeRTOS.h"
5  #include "freertos/task.h"
```

Fig. 12. Necessary headers and Freertos Kernel

```
void setup()
{
  Serial.begin(9600);

  // Initialize random seed
  randomSeed(analogRead(0));

  // Create 6 cars
  for (int i = 1; i <= N; i++)
  {
    cars.push_back(Car(i));
  }

  xTaskCreatePinnedToCore(task1, "Task1", 2048, NULL, 1, NULL, 0);
  xTaskCreatePinnedToCore(task2, "Task2", 2048, NULL, 1, NULL, 1);
}

void loop()
{
  // Empty loop as tasks handle the execution
}
```

Fig. 13. Task Creation in FreeRTOS

### C. Introduction of delay

This function suspends the current task for a specified amount of time. In the code, it is used to introduce a delay of 1000 milliseconds (1 second) between iterations of printing car information.

```
175          car.printCarInfo();
176          vTaskDelay(pdMS_TO_TICKS(1000));
177        }
178      }
179    }
```

Fig. 14. Delay in FreeRTOS

These modifications allow the code to use FreeRTOS tasks and introduce concurrency in the execution of different parts of the code.

The usage of semaphores, mutexes, or queues was not included for the sake of simplicity and to maintain similarity with the original code. However, in a more complex real-world scenario, we would typically utilize these synchronization mechanisms. So for now, each task runs independently and has its own execution context, stack, and scheduling priority.

### D. Output In serial monitor

When running the code and observing the output in the Serial Monitor, we will see the following information:

Car Information: The task1 prints the information of each car in the system. It displays the car ID, priority, location, and action. This information is repeatedly printed every second.

Current Status and Timing: The task2 updates and prints the current status and timing of each car in the system. It sets the current status and timing for each car, advances the timing, updates the status, and prints the updated values. This

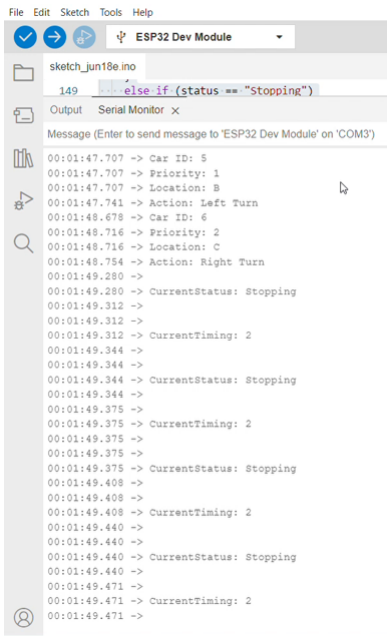


Fig. 15. Output in Serial Monitor

includes the current status, current timing, and timing of each car. The updated status is displayed as the car progresses through different stages, such as ”Approaching,” ”Stopping,” ”Crossing,” and so on. The timing of each car is incremented with each iteration.

Car Status: The task2 also prints the car status separately, showing only the status of each car. This allows you to see how many cars have the same status at a given time.

The output in the Serial Monitor provides a visual representation of the traffic system simulation. It shows information on each car, its current status, and the progression of timing. You can observe how the cars move through different stages and how their status and timing change over time.

## XII. Conclusion

The cross-traffic management we implemented in this paper aims at reducing traffic congestion and allows the non-collision of vehicles on the roads. As a first step, we had to model our system by defining all of its elements and implementing their interaction using the SysML methodology (Use case diagram, state diagram, sequence diagram, activity diagram). VHDL language was used to model the hardware part of our system, and then we generated a test bench to test the behavior of our system.

## XIII. Declaration of Authenticity

We, Anaedu Ukamaka Akumili, Doula Shihab Ud, Unanma Justice Chisom, and Sali Emirkan declare that all the information, ideas, and concepts presented in this project work are sourced from reliable and verifiable academic references and are properly attributed through accurate citations and references. We have made sincere efforts to avoid plagiarism or any form of intellectual dishonesty.

## XIV. Contributions to the Paper and Project

1) Shihab:
   - Project: Sequence Diagram, FreeRTOS Implementation in Arduino
   - Paper: Uml Diagram Section- Sequence Diagram, FreeRTOS Implementation.
2) Justice:
   - Project: VHDL code and testbench implementation as well as the wave simulation of our system.
   - Paper: Problem statement, Hardware implementation, and Simulation.
3) Ukamaka:
   - project: VHDL vehicle direction code and simulation.
   - paper: Documentation of our project work, Block Diagram and References.
4) Emirkan:
   - Project: Use-Case Diagram, Requirements Diagram, State Machine Diagram, UPPAAL Model, C++ Implementation
   - Paper: System Concept And Problem Concept section, UPPAAl Model section and Implementation in C++ section

## References

[1] C.-L. Fok et al., “A platform for evaluating autonomous intersection management policies,” in Proc. IEEE/ACM 3rd Int. Conf. Cyber-Phys. Syst., 2012, pp. 87–96.
[2] L. C. Bento, R. Parafita, S. Santos, and U. Nunes, “Intelligent traffic management at intersections: Legacy mode for vehicles not equipped with V2V and V2I communications,” in Proc. Int. IEEE ITSC, 2013, pp. 726–731.
[3] K. Dresner and P. Stone, “A multiagent approach to autonomous intersection management,” J. Artif. Intell. Res., vol. 31, no. 1, pp. 591–656, Jan. 2008.
[4] K. Dresner and P. Stone, “Multiagent traffic management: A reservationbased intersection control mechanism,” in Proc. 3rd Int. Joint Conf. Auton. Agents Multiagent Syst., 2004, pp. 530–537.
[5] K. Dresner and P. Stone, “Multiagent traffic management: An improved intersection control mechanism,” in Proc. 4th Int. Joint Conf. Auton. Agents Multiagent Syst., 2005, pp. 471–477
[6] M. Quinlan, J. Zhu, N. Stiurca, and P. Stone, “Bringing simulation to life: A mixed reality autonomous intersection,” in Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst., 2010, pp. 6083–6088.
[7] Cooperative Intersection Management: A Survey Lei Chen, and Cristofer Englund.
[8] Introduction to FreeRTOS - Tutorial 1, Hackster.io. https://www.hackster.io/Niket/introduction-to-freertos-tutorial-1-3b7111.
[9] All libraries - FreeRTOS, FreeRTOS. [Online]. Available: /all-library.html.
[10] FreeRTOS, “GitHub - FreeRTOS/FreeRTOS: ‘Classic’ FreeRTOS distribution. Started as Git clone of FreeRTOS SourceForge SVN repo. Submodules the kernel.,” GitHub, Jun. 16, 2023. https://github.com/FreeRTOS/FreeRTOS