PaderControl

Vu Nguyen, Shih Chen, and Thanh Long Phan

***Abstract*—(Phan) In today's world, traffic congestion can have tremendous effect not only on the economy but also on the livelihood of citizens. Therefore, it is essential to keep the traffic flow as smooth as possible. To achieve that, one solution arises from the world of real-time systems: a real-time traffic control application that can schedule vehicles coming in and out of an intersection, which happens to be the major bottleneck in any traffic system. The use of the real-time operating system FreeRTOS is explored in the application of Traffic Management at intersections.**

Proper steps were taken for the implementation of the application. After initial research were complete, it was designed that the system would utilize FreeRTOS on an ESP32 MCU in order to take advantage of the ability to execute two concurrent tasks. The design steps were taken subsequently to construct the the conceptualized algorithm and multiple diagrams including class diagram and activity diagram. UPPAAL then was used to construct a simulation of the application. Afterwards the implementation was developed in C++ using FreeRTOS library for the ESP32 MCU, with unit testing included. First functioning iteration was finally complete using LEDs for easy visualization. The first iteration was a success as the system successfully schedules the cars according to the algorithm, which opens the door for future improvements.

I. Introduction (Nguyen)

THERE is a saying "*Time is money*". Yet, in many parts of the world, commuting takes up a large portion of people's day. During this time, not only it is time-consuming, it is physically draining and there are no productive work that can be done. Even worse, in more populated part of the world, such as the states of California, traffic congestions during rush hours are common occurrence, making the commutation even longer and more frustrating.

Therefore, the aim of this project is to reduce commute time and chance of congestion. With the advancement in self-driving technology, traffic can be revolutionized to be much faster and more efficient. Therefore, this project will explore the development of a real-time traffic control system for vehicles.

If this project is successful, it is also believed that the system that the system will have a major positive impact on the economy. As according to Statista, in 2016, the road logistic transportation occupies 78 percent of the volume of logistic transportation in Europe [1]. An efficient traffic control system can greatly increase the flow of goods, boosting the economy.

II. Background Information

A. Real-Time System and Real-Time Operating Systems (Phan)

Real-time systems are systems that have the objective to complete given tasks within a given deadline. There are two types of real-time systems:

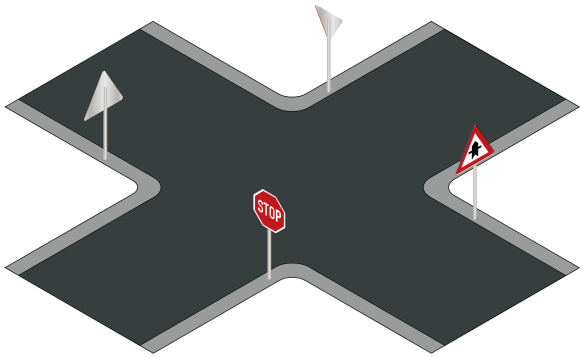

Fig. 1. Typical Four-way Intersection [2]

- Hard real-time systems: Must complete the task before the deadline, otherwise there would be complete failure of the system which leads to catastrophes. Application: healthcare, aircraft control…
- Soft real-time systems: Accept a delay in the completion of task before the deadline, when there is a delay, the consequences are not as much severe. Application: Online commerce, telecommunication...

Real-Time Operating Systems are the operating systems for real-time systems. These usually have several components, notably tasks, kernel and scheduler. Task is a function created to run continuously, while Kernel manages the tasks and distributes resources to them. Scheduler is a part of Kernel, responsible for scheduling the tasks.

B. FreeRTOS (Phan)

There are a lot of RTOS in the market, for example:

- embOS (SEGGER)
- FreeRTOS (Amazon)
- Keil RTX (ARM)
- LynxOS (Lynx Software Technologies)
- ThreadX (Microsoft Express Logic)
- µC/OS (Micrium)
- Zephyr (Linux Foundation)

Among those, FreeRTOS is the most popular. It is a free and open source Real-Time Operating System developed for microcontrollers and small microprocessors. Being an open source under the MIT license, FreeRTOS can be used and distributed freely. As a result, over the past 18 years of continuous development and support, it has been successfully ported to 40+ microcontroller and microprocessor architectures and 15+ toolchains, including the most recent RISC-V and ARM (Arm Cortex-M33) microcontrollers. [3]

To deploy FreeRTOS, it is necessary to use a supported development board and a supported toolchain for ease of

development. As stated above, FreeRTOS has been ported to over 40 microcontroller unit (MCU) architectures and over 15 toolchains, there are several options for implementing FreeRTOS. The most straightforward option is using one of the STM32 development boards along with STM32CubeIDE, which is an IDE with the toolchain for STM32 programming included. Another popular option is to use Keil C Embedded Development Tools that is compatible with most ARM Cortex-M MCUs, this obviously includes popular STM32 and ESP32 development boards. For availability reasons, in this project, an ESP32 board was used, and the toolchain of choice was the ArduinoIDE with self-built libraries.

C. ESP32 *(**Chen**)*

ESP32, developed by Espressif Systems, is a series of dual-core Socs (System on Chip) that are characterized by low cost and good power efficiency. It has a built-in Wifi module, Bluetooth and Bluetooth low energy for effortless use in wireless applications.The dual-core microprocessor supports operating frequencies of up to 240 MHz. Dense electronic integration can be found on the ESP32, such as antenna switches, baluns to control RF, power amplifiers, low-noise reception amplifiers, as well as filters and power management modules. These components help reduce energy usage, making the ESP32 the perfect tool for battery-powered applications. With 4 MB of flash memory and 520 kB of RAM, the ESP32 is capable of running multiple concurrent tasks like reading inputs, writing outputs, controlling hardware, etc. In addition, real-time operating systems usually require a large amount of memory for libraries and computations. Therefore, ESP32 is considered a good candidate for real-time applications on microprocessors and related operating systems, such as FreeRTOS. [4] [3]

III. Methodology (Chen)

The development of this traffic control system began with a set of requirements definitions. First, the developers co-ordinated with stakeholders to define the objectives of the project, and then researched background information. Further specifications and priority of all relevant requirements are later determined. Base on the scenario and requirements, the developers formulated the design using UML diagrams for the first iteration of system. The design was formally verified by a third-party model checking tool, UPPAAL. After formal verification, a hardware implementation was performed by using an ESP32 microprocessor and an on-board FreeRTOS.

IV. Requirements

A. Requirement Elicitation*(**Chen + Phan**)*

1) Objectives: The traffic control system is designed to maximize traffic throughput in an intersection while preventing traffic congestion and dangers imposed by human errors. Although safety is of utmost important, it has to make sense and has to be balanced with traffic throughput, for example, all vehicles standing still and not moving at all is inherently the safest option, but it will never be implemented as in that scenario there is zero traffic flow. Hence, it is important to balance traffic throughput and safety of the system. Automating traffic management comes in as a solution to achieve both of these objectives. 57% of road accidents were caused solely by human errors, and over 90% were at least contributed by them [5]. Therefore, automating the traffic flow has the possibility to mitigate the number of accidents greatly.

2) Background Research: To get familiar to the area of the subject, background research was conducted, mostly on existing traffic related systems. Initially, traditional techniques were being analyzed, in this domain there are traffic lights, roundabouts and traffic signs. These techniques serve as some basic principles for managing the traffic flow. In order to make improvements to the traditional traffic control, however, modern technologies have to be applied. In this domain, there are several options that can be looked at. Vehicular cloud computing is obviously a technology that can be helpful to analyze, as the technology to control vehicles through cloud services is relevant to this particular scenario. Self-driving technology from companies like Tesla and Waymo can also serve as a good source of inspiration. In addition to that, there are working Traffic Monitoring Services in the industry that can be studied, for example, Cross-Traffic Management developed by IMK Engineering. After thorough research about background technologies and knowledge, the following step was to organize the knowledge and collect requirements.

3) Organize Knowledge:

a) Stakeholders: The stakeholders of the project can be identified as follow:

- Developers (Team Paderborn)
- Road users
- Relevant authorities
- Car manufacturers
- Service providers

b) Goal Prioritisation: From the objectives specified for this project, goal prioritization was made. The main goal in the short term would be to develop the first iteration of a simple traffic management that can sort through traffic while providing proper safety for vehicles. In the long term, the traffic control system can be developed and improved to be more complex, but for the short term, a simple simulation and implementation is prioritized.

c) Collect requirements: The collection of requirements is the last step of Requirement Elicitation. The relevant requirements for the system are listed as follows:

- Traffic and driving regulations
- Hard deadline real-time system.
- Fast computing
- Security
- Environment awareness.
- Eco-friendly

B. Requirement Analysis

1) Priority of requirements: After analysing, there are three requirements that are deemed priority. The first requirement is safety. In any traffic control system, it is the responsibility of the system to provide safety for users. If not implemented

properly, the system can cause traffic accidents which have the possibility of leading to injuries or even loss of human lives. Because of that reason, the number one priority is unambiguously Safety. The second most important requirement is traffic regulations of the host country. In the real application, any deployment of traffic controller will be vested thoroughly by relevant authority of the country's administration, therefore abiding to the traffic regulations is necessary. The third prioritized requirement is that the system has to be hard real-time. Because the task missing deadline can have major consequences for the road users, it is very important that the system guarantees the completion of all tasks before deadline.

2) System boundaries: In the early stages of design, various practical factors are taken into account. Firstly, all operations of this traffic control system are compliant with the up-to-date traffic regulations and local driving rules. Secondly, the constraints on the system performance are respected, i.e. resources, computational power, communication latency, or sensibility of sensors, etc. In the first iteration of the implementation, the algorithm developed was only applicable to certain types of standardized vehicles.In addition, all the vehicles are moving at a monotonous speed.

3) Trouble spots: Not all constraints are properly defined, and some exceptions have been identified. For example, extreme weather condition may cause failure or serious damage. The system is designed to be highly dependent on feedback from sensors, so it can be subject to poor communications such as signal delays and interference. In the worst case scenario, such undesirable external events can cause important operations to be overdue and lead to catastrophic consequences. System can also suffer from power outage and unexpected deadlocks.

C. Requirement Specification

The requirement specifications are divided into five major categories. The first category is traffic regulations and driving rules. All vehicles should move along the corresponding lanes based on its driving direction. To form a restricted environment and a simple scenario, a regular size hatchback was chosen for the vehicle in the design.It is based on the Toyota Corolla 2020, which is 4.375 meters long, 1.790 meters wide and weighs 1340 kg. Furthermore, all the cars are driving at 50 kph for safety reason. The next category is about the real-time aspect of the system. There should be at least a hundred meters between each release to ensure that the system has enough time to respond. On this basis, the whole process from the generation of a new vehicle to the completion of scheduling should be done in less than one second. Sensor communication plays an important role in system design. A suitable communication protocol enables fast transmission while ensuring correct data. After research, a list of candidates was found, such as WiFi mesh networks, Dedicated Short Range Communication (DSRC) and GPS positioning technologies.The remaining two categories of requirements are ecological considerations and safety.To promote a sustainable future, this traffic control system complies with RoHs (Restriction of Hazardous Substances) and related regulations.Last but not least, due to the fact that live and property are involved in the implementation, the system is highly resistant to malicious attacks and is able to function properly in extreme weather condition(Level 4).

V. Design and Implementation

A. Algorithm overview ***(Nguyen)***

As the first implementation, the scheduling algorithm that was formulated to tackle the problem set out is a first-in-first-out (FIFO) algorithm with additional resource-based element.

This algorithm utilized is called Four-Block Algorithm. The main idea of the algorithm is that there are 4 blocks on the road, corresponding to 4 different sections of the intersection, as seen in figure 2, that are shared resources. The blocks can be reserved by tasks, which are vehicles that are queuing. Once the blocks are occupied, no other task is allowed to reserve the blocks, similar to a mutual exclusive system.

The number of blocks reserved are dependent on the direction of travel.

- Turning right: 1 block
- Straight: 2 blocks
- Turning left: 3 blocks

Since the blocks are scarce and is a shared commodity, the priority of which vehicle should be scheduled first should also reflect that. Turning right have the highest priority, followed by going straight then turning left.

Currently, the task that first arrives get to reserve the block. If multiple tasks arrive at the same time, then the task with the higher priority is allowed to reserve the block.

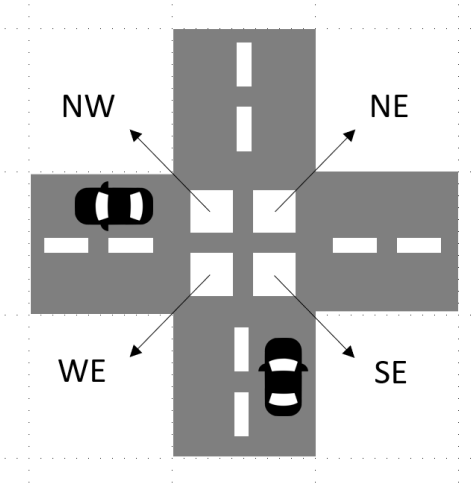

Fig. 2. Typical Four-way Intersection [2]

B. Design

*1) Conceptualization **(Nguyen)**:* The initial step to design the system is to plan out how the system should behave. This is done with the aid with UML activity diagram.

Initially, the algorithm was conceptualized as a completely priority-based approach where each task is assigned a priority point based on a set of condition. However, it was found that

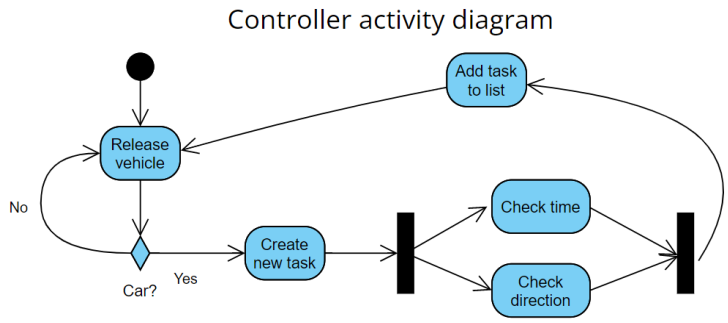

Fig. 3. Typical Four-way Intersection

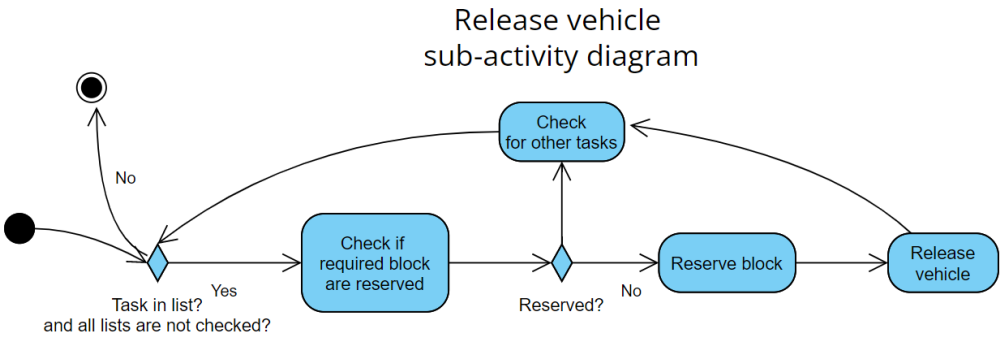

Fig. 4. Typical Four-way Intersection

this idea can lead to many situations that violate the safety requirements that were set out. Therefore, it was scrapped and replaced with a resource-based algorithm instead.

Figure 3 and figure 4 give behavioural models of the algorithm, split into two parts:

- *Main kernel*: The main kernel is the module that encompasses the whole system. It can call functionality of the scheduler and handle input/output interfaces.
  The initial and the default activities of the kernel is simply to try to release any ready tasks in the queue. If there is no ready task, the kernel simply skip to the next step.
  The system would then check to see if there is any car detected by the sensor, if there is, the system will call upon the scheduler to schedule the vehicle as a new task. Once the task is created, it is added into the task list waiting task list. At the same time, the tasks that can be released are added to the same task list. The cycle resets after this point.
- *Scheduler*: The scheduler is the module that decide which tasks should be released next based on the condition mentioned. Once a schedule is made, the scheduler would reserve the block for the task. After finishing its task, the scheduler go to idle and return to the kernel.

2) Abstraction: In the next level of abstraction, the classes can be constructed for the system. This can be seen in figure 5. Each class is a distinct object, with the kernel encompassing the whole system. The cars are external factors and therefore only associated with the sensors and tasks.

The system can then be modelled using Time Automata to check whether the system actually functions as intended

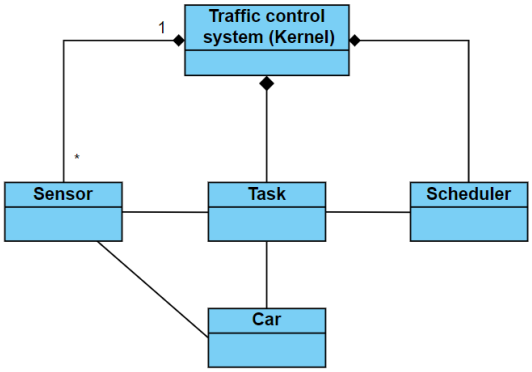

Fig. 5. Class diagram of the main kernel

without any deadlock. This was done on the program called UPPAAL. The model can be seen in figure 6 and figure 7. The system was verified using the simulation and can be confirmed with confidence that the system do acts as intended. However, deadlock was not able to be verified, as the model was simply too complicated for UPPAAL.

C. Implementation and unit testing ***(Phan + Nguyen)***

After the design was verified, the first implementation was created. For this implementation, the sensor class was not implemented to reduce the complexity. The kernel in this first implementation was also not created as a class, but instead as

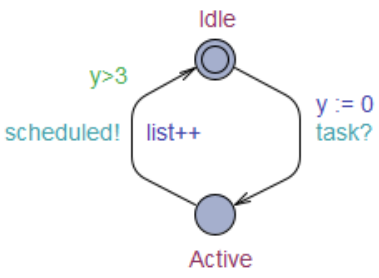

Fig. 6. Uppaal model of the scheduler

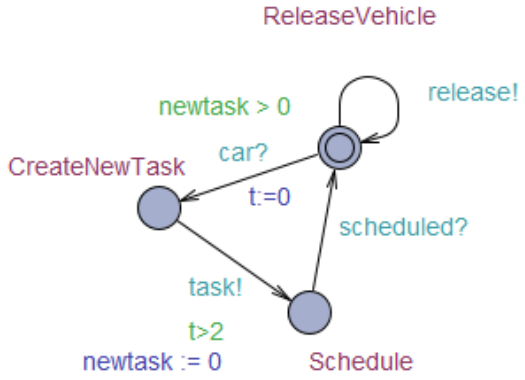

Fig. 7. Uppaal of the controller

a simple wrapper, which can be later be used to run as a task on the ESP32 as a real-time task.

The other two system related classes, namely Tasks and Scheduler, were implemented. The implementation was split into a header and source. The headers outline the attributes and methods of the classes, while the sources provide the method definitions for the classes.

With the *Task* class, there are no methods other than the constructor, where the object instance is created and values assigned, and the destructor, there the object instance get destroyed. However, the task attributes of the *Task* class store valuable information which aids in scheduling, such as which road the vehicle comes from, when was the task created, travel direction, and more.

The other class, the *TrafficScheduler*, is where the resources, logic, and methods for the systems are located. All methods are public for ease of access by the kernel. The status of each block and the lists for ready and waiting tasks are also stored here. To map the design of the Four-Block Algorithm to the implementation code, several methods are utilized in the *TrafficScheduler*. When a vehicle enters the phase and is added to the task list, there are certain blocks the vehicle would like to use for its travel. For example, a vehicle from South road wanting to turn right would like to take *BlockSE*, the Southeast Block; or a vehicle from East road wanting to go straight would like to take *BlockNE* and *BlockNW*, which are the Northeast and Northwest Block.

The method *fCheckIfBlocksAreReservable()* is therefore responsible for checking if the block wanted by the vehicle are occupied. For example, in the case, of the Vehicle coming from East road, there are 3 scenarios: vehicle going straight, vehicle turning right, and vehicle turning left. If the vehicle wanting to go straight, *BlockNE* and *BlockNW* are needed for its travel, and if one of these blocks are *occupied*, it can't be reserved, so the method will return a *false* which means the blocks are not reservable. Similarly, if the vehicle is wanting to turn right, *BlockNE* is checked, and if it is *occupied*, the method will also return a *false*. Last scenario is vehicle wanting to turn left. Here, the vehicle wants to take *BlockNE*, *BlockNW* and *BlockSW*, and if any of these blocks is *occupied*, a *false* is returned to signal that the blocks are not reservable. If none of these conditions are true however, it means the blocks are reservable therefore a *true* will be returned. This applies to other 3 cases for vehicle coming from West, North and South roads.

The two methods *fReserveBlocks()* and *fUnreserveBlock()* are two simple utilities to control the status of the blocks.

fReserveBlocks() is meant to reserve the blocks for the vehicle if they are reservable and the vehicle can go into the intersection. Here, there are 4 cases: vehicle coming from East road, vehicle coming from West road, vehicle coming from North road, and vehicle coming from South road. In the case of vehicle coming from East road, the fuction will reserve *BlockNE* and *BlockNW* if the vehicle goes straight, *BlockNE* if the vehicle turns right, *BlockNE*, *BlockNW* and *BlockSW* if the vehicle turns left. This applies to other cases of vehicle coming from other roads as well.

fUnreserveBlock() unreserves the block to make reservable again once the vehicle passes the block. It is done by marking the block in question *unoccupied*.

The culmination of all the methods above is the *fReleaseVehicle()* method. This is where each waiting list are checked and deciding the vehicles that will be released. With the current implementation, only one vehicle is released each loop.

The scheduler first collects all the first element of the waiting list for each lane, which is the vehicle that is waiting at the front of the queue, into a vector. Then each element within the new vector is compared with each other to find the task that has arrived first. When there are tasks that arrived at the same time, the direction of travelled is considered instead. If the direction is also the same, then the currently selected task is chosen. Once the task to release was decided, it is added to the ready list and removed from the waiting list.

The last method that was implemented is the fAddTaskToList() method, a simple method where tasks are assigned to their corresponding wait list based on the road that they came from.

The functionality of each method was tested using a unit test. Once the result is satisfactory, the final implementation on real-time system starts.

D. VHDL Design(**Phan + Nguyen**)

The code implementation is followed by hardware design in VHDL. The VHDL implementation is done on the basis of previous software code. It acts as the *Scheduler* class and has two concurrent processes running in parallel.

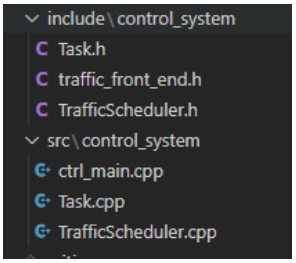

Fig. 8. Implementation structure

The first process is to schedule the vehicle by reading a 16 bits input and to decide which one to release. The inputs would be compared to each other and the one with the closest arrival time or highest priority will be released.

The second process will be to reserve the blocks when the vehicle is released and enter the intersection. The reservation process is done using the arrival road and the travel direction. If the vehicle turns right, 1 block is reserved. Whereas, 2 and 3 blocks will be reserved for going straight and turning left, respectively.

E. First release version ***(Chen)***

The first release was done on ESP32 using FreeRTOS and Arduino IDE. The code was adapted for the use on the Ardunio IDE, however, the main functionality of the code stays the same. For FreeRTOS, two tasks were created. The first task, called *SchedulerTask()*, act as the kernel, which call methods from the *TaskScheduler* class. Within this task, vehicles are also being randomly generated to act as stimuli, since there are currently no sensors connected to create new tasks. In order to monitor the operation, the scheduling results are printed on the serial monitor in Arduino IDE. It provides information regarding the vehicle on roads and the release status. The Vehicle ID and the driving directions (initial orientation and next direction) are displayed with a corresponding time stamp. The next task is *RoadTask*, which interacts with the road components and handle I/O. Besides serial monitor, a LEDs extension is used to visualize the status of reserve blocks. Four LEDs forms a square to represent different orientations, i.e.North-East, South-East, South-West, and North-West. Each LED is powered by ESP32 and is connected in series with a 330 Ω resistor.The on state of the LED indicates that the corresponding block is reserved for vehicle release. Once the vehicle has successfully reached its destination, the LED goes off, which means that the block is not reserved and is ready for the next task.

F. Open points and future improvements ***(Nguyen)***

While the algorithm works well, there are still some flaws. One of them is the fact that if there are multiple cars on the same lane that wants to go in the same direction, because of the mutual exclusion of blocks, these cars have to wait until the leading car finish before the next car can pass. This is very inefficient. Connected to this is the release of multiple cars. Next implementation should undoubtedly have this as a functionality. However, it is also crucial to be mindful of how many cars from the same lane are released at the same time so that other lanes can also be released.

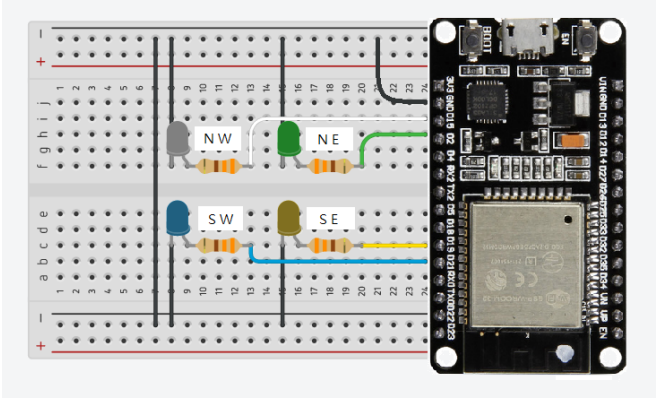

Fig. 9. Hardware implementation: Represent status of reserve blocks using LEDs

The other open point is that the full functionality of RTOS has not been utilized, up to this point, there are two tasks, each pinned to a core, meaning the operating system never had to truly schedule the tasks. Therefore, with the expansion in the scheduling algorithm, more task could be created to keep the complexity lower while fully utilize the functionality of RTOS.

Other less pressing open point is adding sensors instead of random generator for the tasks.

VI. Conclusion **(Chen)**

In summary, a series of in-depth studies were conducted on different aspects of this system. The development teams successfully implemented the first iteration if the traffic control system and delivered the requirement using ESP32 and FreeRTOS. FreeRTOS exploits the outstanding computational power of ESP32 to perform multiple concurrent tasks, which brought a satisfying result for the hardware implementation. On the other hand, the vehicles are successfully scheduled and dispatched without any collision in the simulation.Finally, future improvements will build on the first iteration to improve efficiency by minimizing the wait time between each release.

References

[1] Statista, "Distribution of transportation used for flow of goods within europe in 2016, by modes of transport." https://www.statista.com/statistics/639962/logistics-market-transportation-share-europe/.
[2] RoutetoGermany, "Right of way." https://routetogermany.com/driving-ingermany/right-of-way/.
[3] Amazon, "Freertos." https://www.freertos.org/.
[4] espressif, "Esp32 wi-fi bluetooth mcu." https://www.espressif.com/en/products/socs/esp32.
[5] J. R. Treat, N. S. Tumbas, S. T. McDonald, D. Shinar, R. D. Hume, R. Mayer, R. Stansifer, and N. J. Castellan, "Tri-level study of the causes of traffic accidents: final report. executive summary.," tech. rep., Indiana University, Bloomington, Institute for Research in Public Safety, 1979.

VII. Responsibilities

The distribution of work is as follows:

A. Chen Shih

ESP32 implementation, front end for simulation.

B. Vu Quang Tuong Nguyen

VHDL, Task class, 2/3 TrafficScheduler class, unit testing.

C. Thanh Long Phan

Algorithm, FreeRTOS research & structure, 1/3 TrafficScheduler.