Datateknik
Computer Engineering

Realtime Embedded Car Safety System on Arduino

Erick `Xiao` Guan / Nico Ferrari

Mittuniversitetet
MID SWEDEN UNIVERSITY

Campus Härnösand Universitetsbacken 1, SE-871 88. **Campus Sundsvall** Holmgatan 10, SE-851 70 Sundsvall.
Campus Östersund Kunskapens väg 8, SE-831 25 Östersund.
Phone: +46 (0)771 97 50 00, Fax: +46 (0)771 97 50 01.

MID SWEDEN UNIVERSITY
Department of Information Systems and Technology (IST)

Examiner: Mikael Gidlund, *mikael.gidlund@miun.se*
Supervisor: Ulf Jennehag*, ulf.jennehag@miun.se; Mikael Hasselmalm, mikael.hasselmalm@miun.se*
Author: Erick `Xiao` Guan, xigu1500@student.miun.se; Nico Ferrari, nife1600@student.miun.se
Main field of study: Computer Engineering
Semester, year: Autumn, 2017

Table of Contents

1 Introduction

The project is built for Realtime Embedded System in MIUN. The topic is about realtime embedded system. We present a working project that built with some embedded system with realtime theories in mind.

1.1 Background and Problem Motivation

Realtime embedded system is the system for embedded devices which can execute tasks meeting timeliness of their criteria. The system may be a component of a larger system. It's useful for many controlling systems such as rockets, manufacturing system and even a car. They can be cheap and physically small, hence it can find places in the critical part of our society, such as controlling the traffic lights and factories. As the Internet of Things arisen, the realtime embedded system is becoming more and more important topic to invest since IoT needs cheap, abundant and responsive devices to built upon. Cars have revolutionized our life and they are becoming more intelligent and safe over years. This requires more computational resources, stable system with time constraints and communication with other services, even physical infrastructure.

1.2 Overall Aim

We are interested in building a project with realtime embedded system with communications to other services, such as a distributed system. We come up with a typical scenario in our daily life to improve the driving safety. The trend for the future driving is utterly automatic without human interactions. As the time limit of our project and the difficult technical challenges ahead, we start with a simplification of this vision. We construct the realtime embedded system for cars as well as an intelligent intersection node to comprise a distributed system together.

1.3 Concrete and Verifiable Goal

More concrete goals will be:

1. Find and read papers and the book to better understand the problem.
2. Find proper hardware platforms to simulate this scenario.
3. Design the structure of two system nodes, communication strategy and simulation strategy.
4. Build and connect two devices with realtime operating system installed.
5. Implement the software system, distributed system and simulation.
6. Execute our program to see a preliminary pr the tasks execution time, delay and compare with non-realtime scheduling.

1.4 Time Plan

The project starts in the beginning of October, 2017 and ends in the month. We allocated our time into 4 weeks. For the first week, we will write the proposal, general aspects of problem and devising our methodology. For the second week, we will look into materials and hardware for Goal 1 and Goal 2 as well as Goal 3. For the third week, we will connect the hardware and build our simulation for Goal 4 and Goal 5. In the last week, we will measure our data and finalize our report for Goal 6 as well as an oral presentation.

2 Theory

In this chapter, we present the theory we used in the study.

2.1 Distributed System

A distributed system comprises of different computers system in which they communicate across the network by messages to collaborate on certain tasks. Its main purpose is to share system resources. In such system, there are challenges of manifold:

- *heterogeneity* of network, operating system, programming languages;
- *openness* with a published standards;
- *security* facing threats from internet;
- *scalability* to sustain running of the service with various of workloads;
- *failure handling* to recover from inevitable disasters;
- *concurrency* to process with optimized performance;
- *transparency* to mask underlying system implementation.

To facilitate with all these challenges, one has to design system carefully [1].

2.2 Real Time embedded systems

A Realtime embedded system is an electronic system that can provide guaranteed worst-case response times to critical events, regardless of how efficiently the hardware operates. Realtime systems are distinguished from the other applications by the need of adhering to rigorous requirements. The correctness of the system depends not only on the results of computations, but also on the time at which the results are produced. The most important and complex characteristic of real-time application systems is that they must receive and respond to a set of external stimuli within rigid and critical time constraints. Realtime systems are distinguished in:

- **Hard realtime system:** if producing results after its deadline could generate catastrophic consequences.
- **Soft realtime system:** if producing results after its deadline has still some utilities for the system but it causes a performance degradation.
- **Firm realtime system:** if producing results after its deadline is useless for the system but it doesn't cause any damage.

2.2.1 Task Definition

Usually, several jobs are handled by a single embedded system and to enhance re-usability and maintainability, those jobs are modularized and called tasks. Time bounds on different tasks may be different and , as already explained, the consequences of a task missing its time bounds could vary from task to task (importance of a task). There are different categories of real-time tasks:

- **Periodic Tasks:** A task that is repeated after a certain time interval.
- **Aperiodic Tasks:** A task that can arise at random instants and aperiodic tasks usually are soft real-time task because they can recur in quick succession so it could be difficult to meet deadlines.
- **Sporadic Tasks:** These are aperiodic tasks where the difference of arrival time between two instances of the task cannot be zero. Sporadic tasks usually are hard real-time tasks.

2.2.2 Task Scheduling

The task scheduling is the basic mechanism adopted by a real-time operating system to meet the time constraints of tasks. Real-time scheduling refers to determining the order in which the various tasks are to be taken up for execution by the operating system. Each task scheduler is characterized by his scheduling algorithms.

3 Methodology

We defines our higher-level problem and detailed problem; read the material; propose a scenario; devise a strategy; build hardware and software; and measure the data with interpretation.

3.1 Problem Definition and Understanding

We take a mix approach towards the research problem to understand the theory qualitatively. With the information we gathered from Goal 1 and Goal 2, we are comfortable to propose our imaginary scenario relating to the real world case.

3.2 Project Design and Architecture

We further extract the related components from the real world scenario and simplify it towards our concrete problem. Later, we compile our hardware schema with electronic components to demonstate our scenario for the audience. And we divide them with software components as Goal 3 claims.

3.3 Hardware Connection

We connect our hardware according to our hardware schema. Then we install the operating system we needed as Goal 4 claims. We test our connection and components before we commence the next step.

3.4 Implementing Software

We start to write code for our scenario as well as simulation parts with random variations. We adjust the parameters to understand system behaviors as Goal 5 claims. We design our software from top down approach but implement from bottom up.

3.5 Data Measurement

At last, we measure, compare and interpret our data. We reason its behaviour based on our previous view and theory to draw a conclusion. At last, we put up our report and presentation.

4 Implementation

In real world, we hit on the road and junctions inevitably like Figure 4.1. As a driver, we are often confused by the lights. As a pedestrian, we are obliged to wait for the signals. Moreover, when people is in terrible rush, they ignore the traffic rules. Often accidents happen in such junctions.

Figure 4.1: A busy intersection with many traffic lights [2]

The future vision for traffic is fully automatic. Since we are not here yet, we can take a gradual approach towards improving this scenario. Hopefully, it could be useful for the future technology.

4.1 Problem Definition

As of the time limit and research nature of the project, we simplified our problem as Figure 4.2 shows.

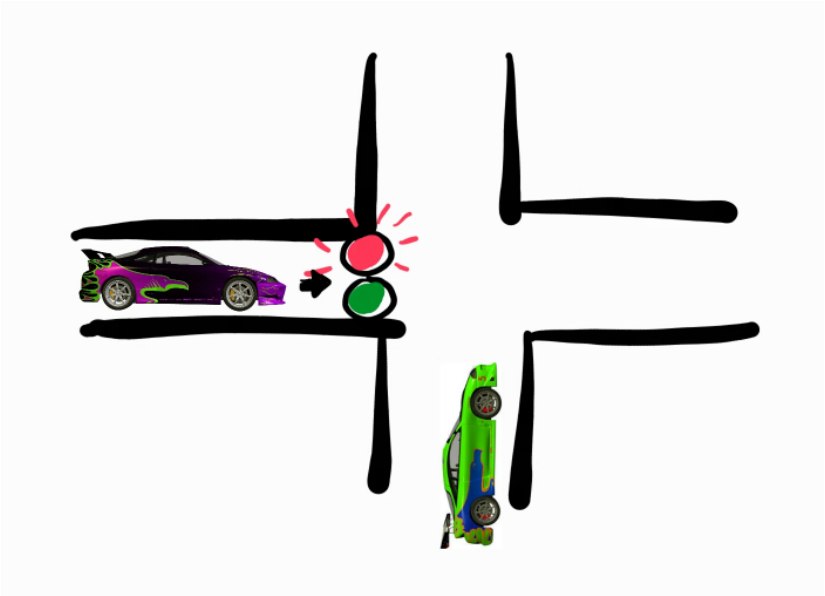

Figure 4.2: Our traffic intersection scenario

We propose two schemas. The first schema is that a car is entering the intersection when the other car is driving at $20km/h$ and about $20m$ towards the junction. Without doing anything from the driver of the other car, they will crash. So the intersection instructs the other car to stop. The message is handled by the controller of the car.

The second schema is shown as Figure 4.3.

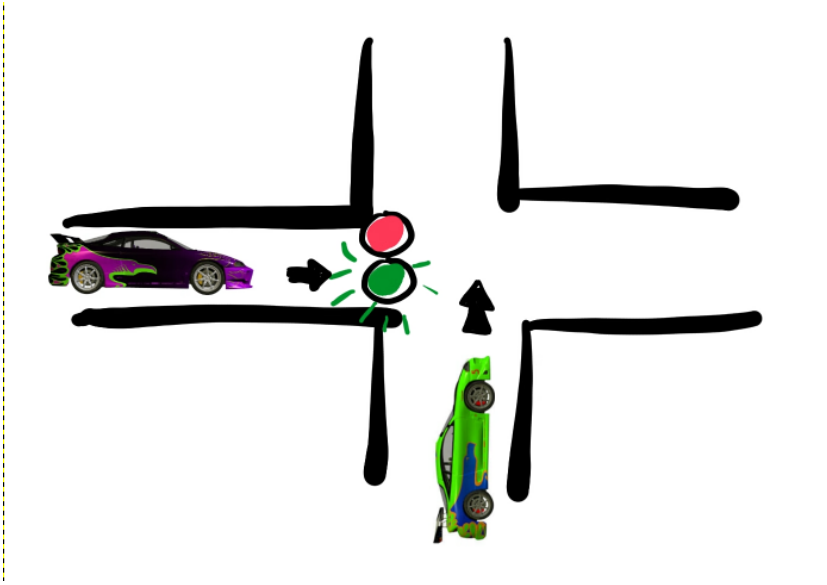

Figure 4.3: Our traffic intersection scenario 2

4.2 Hardware and Software Design

We will build a distributed system for the intersection management system (called *foreman*) as well as a realtime embedded system for controlling the car (called *proletarian*). The manager monitors the intersection and instructs the incoming cars to slow down or stop to prevent potential accident. The controller is responsible for the drivers' reaction as well as manager's instruction which can override the possible poor decision the driver can make.

We can represent our system with a simple graph that describes the actors of it as shown in Figure 4.4.

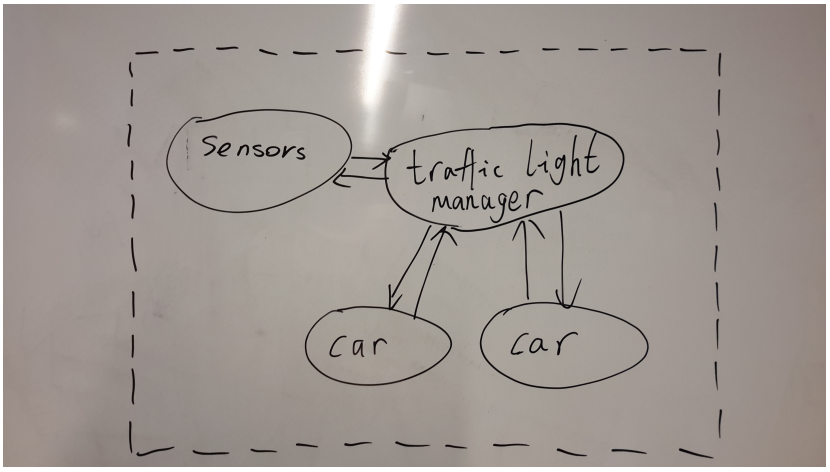

Figure 4.4: Actors of our system

Due to hardware limitations and project time constraints, we decided to simulate the external sensors and other incoming cars, as shown in Figure 4.4 using two embedded systems:

- Arduino: to simulate the car.
- Raspberry: to simulate the traffic manager and external events.

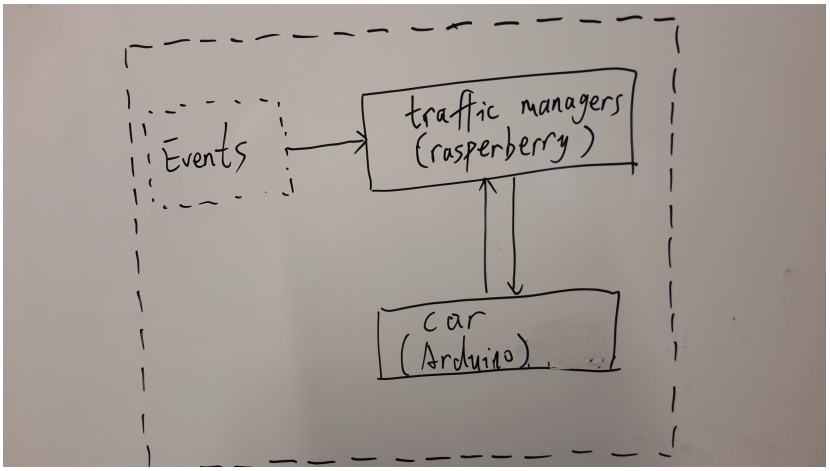

Figure 4.5: Simplified view of actors of our system

They together comprises a distributed system. Their hardware connections and gadgets are connected as Figure 4.5 shows.

At this stage, the draft of our platform are comprised as Figure 4.6 shown.

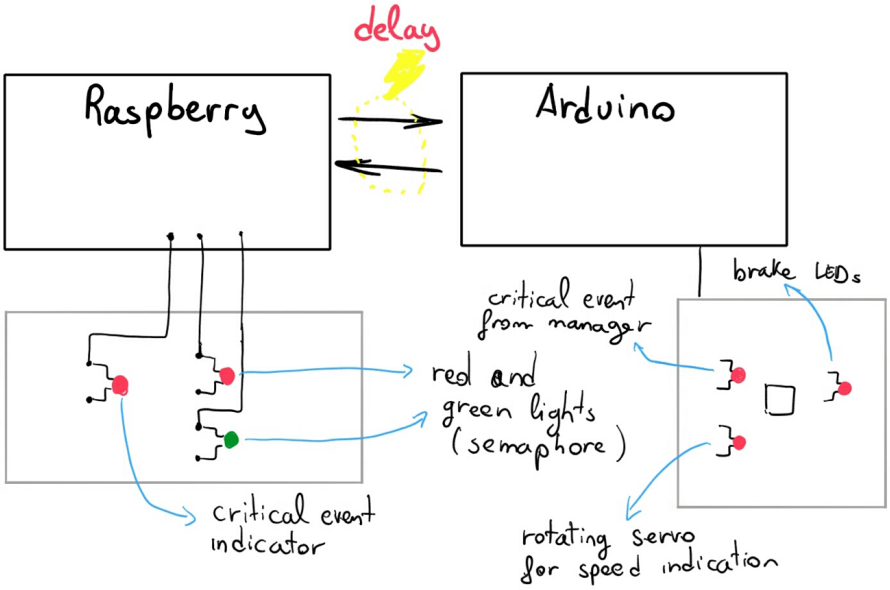

Figure 4.6: Simplified composition draft of all components of our system

4.3 System Development

We used PlatformIO for setting up our development environment as well as managing dependency such as FreeRTOS. PlatformIO is written in Python. It provides package manager, package repositories, environment management and serial tools for the hardware board. We also use git for version control.

Due to limited constraints, we found that we should carefully choose data type for lower memory print.

Table 4.1: Data type in Arduino

Data type	Size (byte)
boolean	1
char	1
unsigned char	1
int	2
unsigned int	2
word	2
long	4
unsigned long	4
float	4
double	4
string	1 + x
array	1 + x

The Arduino language is actually based on C. The compiler concatenate our program with Arduino framework. Then the compiled objects are linked against AVR Libc to enable standard C library functions. In particular, we prefer to use the type defined in `stdint` as they present the data size directly rather than some data type defined in Arduino as Table 4.1 shown. By doing so, we can have more confidence of data range and memory size. For example, `int8_t` is one of such type.

4.3.1 Parameter Tweaking

In finding our parameters to implement algorithms for the car model. We chose to use the real parameters in such simulation. We choose Volvo C30 TS as our model. The braking distance from $100km/h$ is $39.05\ meters$ [3]. As of acceleration number, we refer to [4] - the maximum braking acceleration should be $8.5m/s^2$. We also know C30 get to $100km/h = 27.778m/s$ in $6.7s$ from [5]. So that the acceleration should be $4.145937m/s^2$ which we consider it's distributed uniformly without any gear ratio selection.

We plan to design our response boundary from this real world data. We denote that $d_{response}$ is the distance traveled in the driver's response time when the driver don't do anything. We also denote that d_0 is the distance traveled when the car is stopping at a maximum rate from $v_{car} = 27.78m/s$. We concluded that $t_{response} = d_{response}/v_{var}$. Some examples are shown as Table 4.2.

Table 4.2: Response range

Total distance (m)	Response distance (m)	Response time (ms)
40m	0.95m	34.19
45m	5.95m	241.18

From Table 4.2, we can design our realtime embedded system with a reasonable timing constraints in mind.

4.4 *Proletarian* Software

In *proletarian*, we divided the whole tasks of controlling a car into a task set. We also use FreeRTOS facilities to implement the task set.

4.4.1 Real world time in FreeRTOS

Given the velocity in a specific point and the acceleration, we need the variation of time to calculate the velocity in another point [6]. To do this we used the function `xTaskGetTickCountFromISR` that returns the count of ticks. To convert this in milliseconds we used the constant variable `portTICK_PERIOD_MS`.

4.4.2 *Proletarian* Task

We modeled the car with several functions.

There are sporadic tasks, aperiodic task and periodic tasks. They are:

$$\mathcal{J} = \{J_1, J_2, J_4, J_5\}$$

and

$$\Gamma = \{\tau_1, \tau_2, \tau_3\}$$

Table 4.3: Task List

Task	Type	Priority	Description
J_1	Aperiodic	3	Receive message from *foreman*
J_2	Sporadic	2	Speed control
J_4	Aperiodic	1	Slow down motor speed
J_5	Aperiodic	1	Control the brake light
τ_1	Periodic	4	Monitor speed from the motor
τ_2	Periodic	5	Send the information back to the *foreman*
τ_3	Periodic	4	Receive and process driver response from the potentiometer

As the Table 4.3 shown, in J_1, the messages from *foreman* were parsed and fed into J_2; in J_2, the speed is calculated based on current reading speed which should simulated by a servo; J_5 detects the speed and slows down the car by turning down the voltage for motor; J_5 will light up LEDs when it's slowing down; τ_1 reads the speed from the car which is simulated by a motor; τ_2 sends feedback to the *foreman* so that the junction can act accordingly; τ_3 reads the driver response from the potentiometer and fed the information to J_2 for processing. They have different priorities so that we can make sure we take on the urgent tasks without a degrading experience.

As the resource is limited, all our resources are shared by means of global variables. Action applied to certain resources and task dependency to certain resources are shown in UML figure as Figure 4.7 shown.

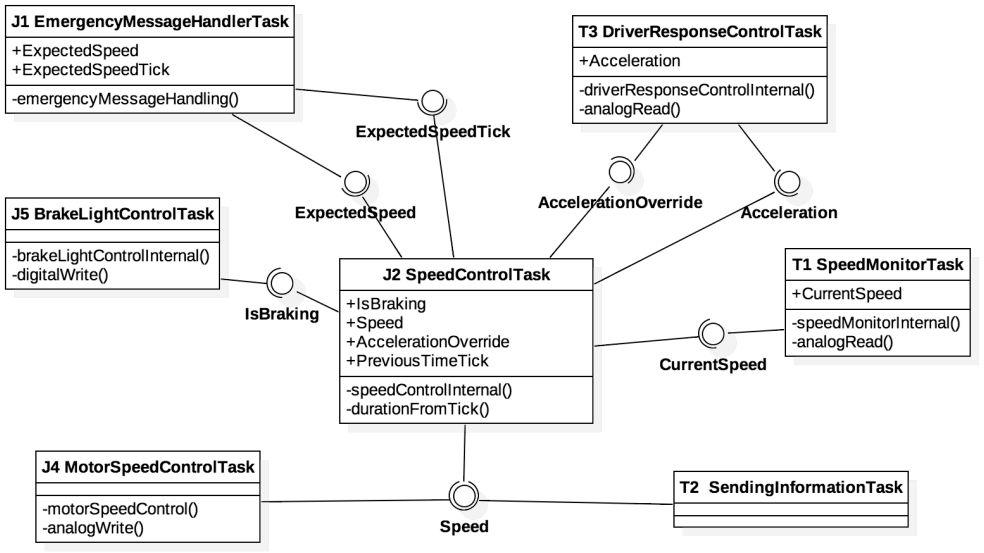

Figure 4.7: System Design on UML graph

We have a detailed explanation for each resources as Table 4.4 shown.

Table 4.4: Resource description

Resource	$\in$	$\rightarrow$	Description
`ExpectedSpeed`	J_1	J_2	*foreman* instructed speed at future moment of `ExpectedSpeedTick`
`ExpectedSpeedTick`	J_1	J_2	*foreman* instructed tick in the future
`IsBraking`	J_2	J_5	controls for LEDs
`Speed`	J_2	J_4, τ_2	speeds to be set for motor
`AccelerationOverride`	J_2	τ_3	whether *foreman* overrides driver's control
`PreviousTimeTick`	J_2	-	internal time tick to account for next speed value
`CurrentSpeed`	τ_1	J_2	reads speed from motor analog value
`Acceleration`	τ_3	J_2	takes into account of driver's response

We even have a math function to let system have a smooth feedback from the driver input as Figure 4.8 shown.

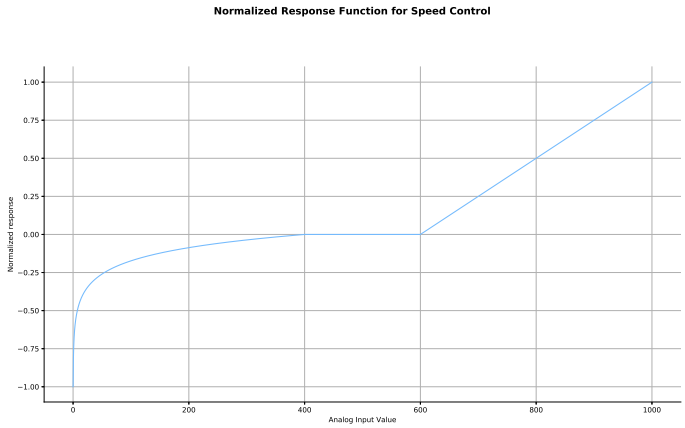

Figure 4.8: A smooth response function for driver

4.4.3 Hardware Design

Once we have decided what we want to do, we take on the hardware and breadboard. It's more likely to get a proper working circuit if we draw that first. We use Fritzing for drawing the board. They are shown as Figure 4.9 and Figure 4.10.

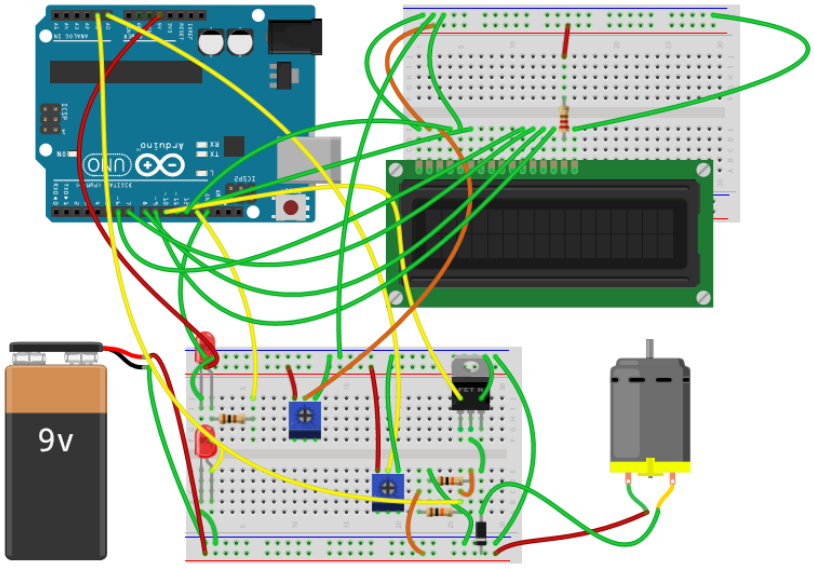

Figure 4.9: *Proletarian* Hardware Connection

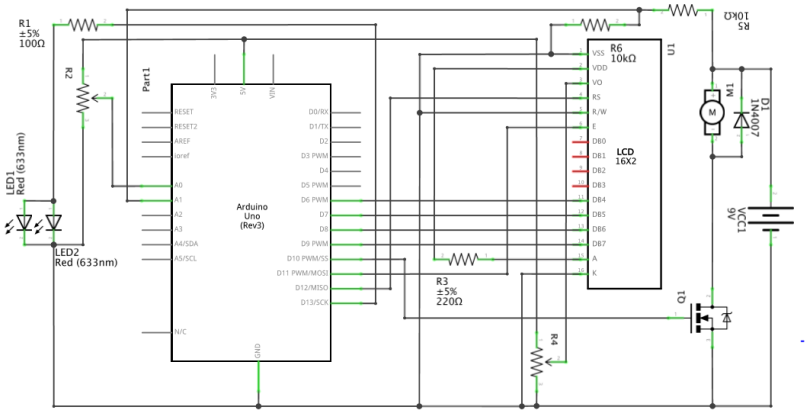

Figure 4.10: *Proletarian* Circuit Connection

Then we continued on connecting our hardware as Figure 4.11 shown.

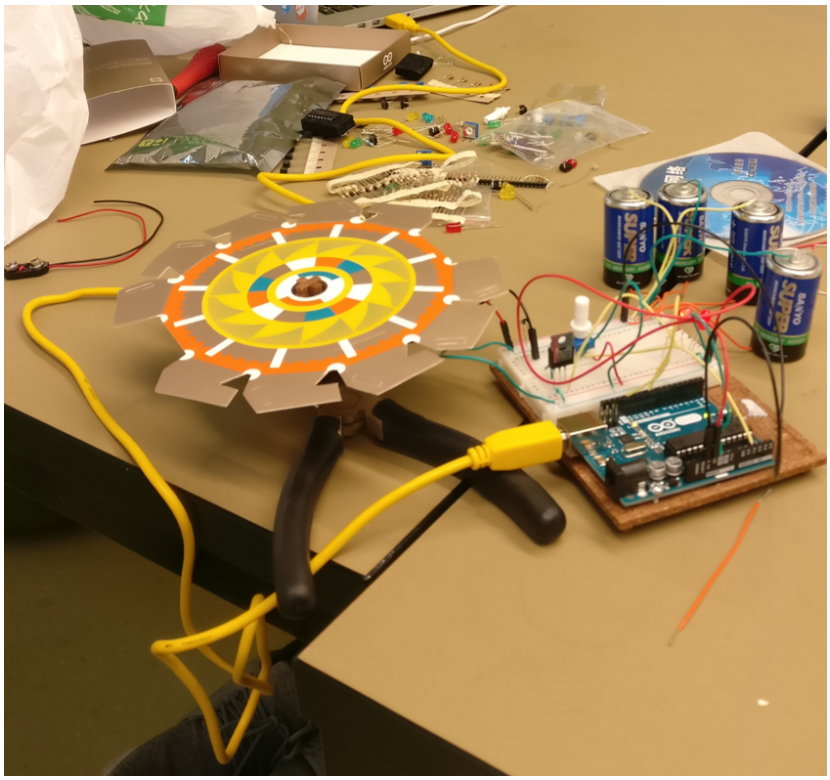

Figure 4.11: *Proletarian* hardware is composed and connected to the board

4.5 Architecture Work and Collaboration

Since we have a task set, we separate them into different tasks which resides in different files. We use C++ because it gives us better control of compiler behavior. Firstly, each task has a function which deal with acquiring locks or other mutual exclusion implementation (for example, semaphore or messages). Secondly, we implemented their core functionality by putting them into another inline function so that they only con-

tains the logic to control hardware or reading or communication. Thirdly, we documented their responsibility. Lastly, we build test suite to ensure their correctness before even run the tasks.

For collaboration, we use git flow and do code review to make sure mutual understanding of work by other. Moreover, we use `clang-format` to have the same coding style.

The git flow that we use is described as follows:

```
# 1. update from github
git fetch origin
git checkout -f master # This step clears out non-staged
↪  content!
git merge origin/master # so you sync with others.

# 2. starts to work locally
git checkout -b <dev-whatever> # use `dev-` as a prefix helps
↪  your branch list clean
# do you work and commit
# commit often and concise. every commit should be atomic
↪  which only affects one
# part of system
# the commit message should make sense.

# 3. prepare for pull request
# 3.a rebase to track master
git fetch origin
git rebase origin/master
# 3.b (optioanl) revise previous commits
git rebase -i HEAD~<NUM> # if you need to change the order or
↪  edit or squash previous commits
# 3.c push the right branch
git checkout -b <whatever> # this time without `dev-` prefix,
↪  we push this branch to github
git push origin <whatever> # push to the repo as <whatever>
↪  branch, with `-f` you can overwrite remote branch
# 4. go to
↪  https://github.com/fantasticfears/realtime-embedded-course
# and start a pull request.
```

4.6 Task Scheduling

After we had implemented all the tasks, we decided to try two task strategy. One uses a constraint . Then we draw one particular DAG to execute. More specifically, we don't use preemption. We don't use time slice. We

don't use periodic tasks. This is called *Approach A*; The other uses a priority scheduling with preemption. This is called *Approach B*.

Constraints $C = \{\tau_3 \succ J_2, \tau_1 \rightarrow J_2, J_1 \rightarrow J_2, J_2 \rightarrow J_5, J_2 \rightarrow \tau_2, J_2 \rightarrow J_4\}$ are used to derive DAG. And the DAG is presented as Figure 4.12

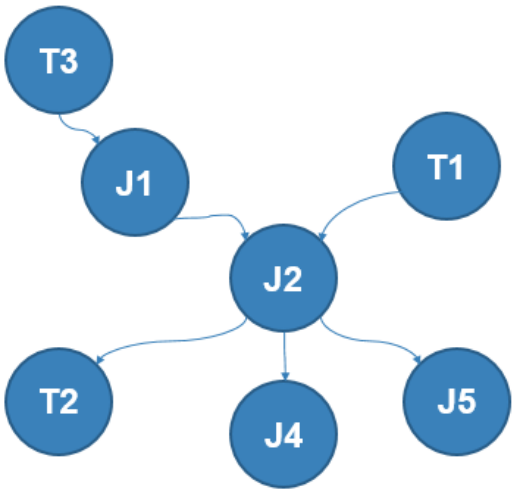

Figure 4.12: DAG for Approach A

4.6.1 System Interrupt

J_1 receives the serial communication. While the FreeRTOS handles the interrupt, J_1 will read from system buffer. That is higher than any priority just like task scheduler which we don't have to know.

4.7 macOS Toolchain

We use Arduino IDE in the beginning which turns out to be problematic. As an IDE, it doesn't reveal how it compile the individual files, how they are linked together. Nor does the complication flags or linker flags. Once we encountered sketch file oversized, we checked the binary generated by `objdump-avr`. We have found out that there are a lot of functions not stripped away by the linker for FreeRTOS static library. It must be a toolchain problem.

We switched to PlatformIO soon after that. To use PlatformIO, we have to setup local toolchain for the specific architecture - AVR. These steps are done by installing compiler and C standard libraries from homebrew.

4.8 Logging as data measurement

We have the logger set up and logs in every tasks. Moreover, we intended to use log to collect data from other end, possibility *foreman*. But it takes on buffer space as other serial communication library does. It's worth mentioning that serial communication takes a large portion of memory

space in our implementation. Though, we have a macro that can disable logging and can remove it from even being compiled.

However, it's impossible to understand or analyze without the logging functionality. It's so crucial to whole development.

4.9 Failure Attempt

However, we hit on a stack overflow. We will discuss this in the Chapter 5. To solve this problem we tried to change configuration file of FreeRTOS's scheduler, permitting us to edit the heap size. There are different heap implementations provided for this systems and we decided to use `heap_4.c` as suggested in exploreembedded.com. By default, the Arduino's version of FreeRTOS uses `heap_3.c`, making the `malloc()` and `free()` functions thread safe and probably considerably increasing the RTOS kernel code size. Furthermore, the `configTOTAL_HEAP_SIZE` setting in `FreeRTOSConfig.h` has no effect when `heap_3.c` is used. Contrariwise, with `heap_4.c` the total amount of available heap space is set by `configTOTAL_HEAP_SIZE` (which is defined in `FreeRTOSConfig.h`). We also tried to remove Log messages to reduce the memory used by these functions.

In numbers, we know that our Arduino board has 2048 bytes. And we tried to use FreeRTOS with only 2 priority levels and a extreme small stack space. From the accounting code, we also found out each task needs more or less 100 bytes and some on heap. The scheduler needs 260 bytes. With more resource variables, it's simply not possible to handle all tasks.

Changing the heap size and removing the Log functions didn't provide any success, so we started to build something much simpler than we previously design. We still manage to design a system with the similar tasks and reuse a large portion of code from previous codebase. The new tasks are shown as Table 4.5.

Table 4.5: Task List

Task	Description
J_1	Simulate message from *foreman*
J_2	Speed control
τ	Receive and process driver response from the potentiometer

We still starts from *Approach A*, but with a much more simpler DAG as Figure 4.13 shown.

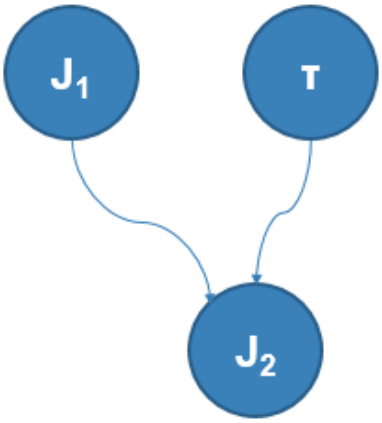

Figure 4.13: DAG for new design

5 Results

We have conceived an idea, have designed our schema, have implemented our project and have found out problems along the way. The effort vested in the project is shown as Figure 5.1.

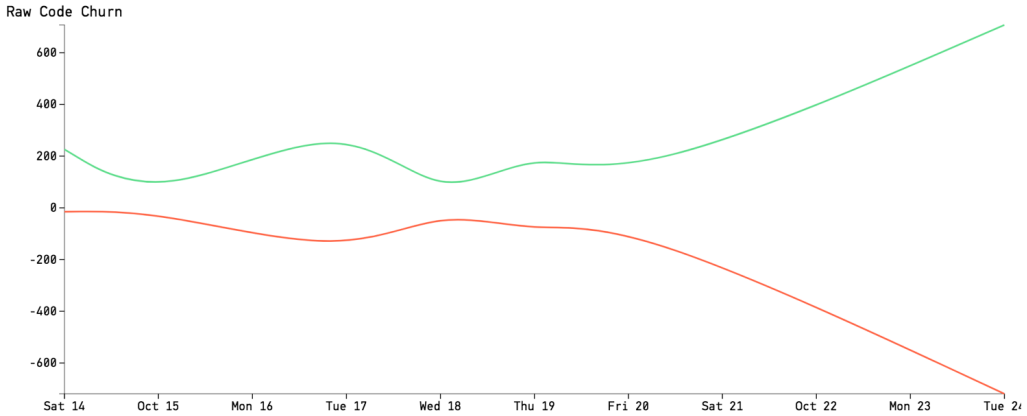

Figure 5.1: LOC change over time for *proletarian*

We have executed the program and extracted the log. The program can stop the car at pre-defined time tick.

Along the way, we find out that hardware resources are too limited to run FreeRTOS on Arduino for even simple program. There is nothing can be done to construct a simple working example. But we have known that realtime system should be fast enough that is more about conformity to predefined time constraint. While the embedding system gives us the resource constraint. We will discuss resource and money problem in the following section.

We also found out this FreeRTOS port is not feature-rich. It doesn't protect the memory as other operating system provides. It doesn't provide any programing support to fulfill time constraint either. So it will be hard to predict the system behavior which should not be considered as a production ready system. Towards embedded system, we know that it's hard if you don't know how much resources you need. When we start an embedded system project, we should always taken OS resources needed and our tasks resources needed into account.

5.1 Stack Smashing

The problem we hit on are stack smashing which still happens in Linux for example. The stack smashing is also called stack buffer overflow in which one stack size of function is growing to others which change others state. This can be used to exploit a security hole if being constructed carefully. In general, a modern kernel builds something to prevent it from happening.

But for FreeRTOS we used, it doesn't protect the memory since Arduino have small memory space.

One may argue that sometimes embedding system is that we should construct a system in a small memory space. But there are two counterparts to this proposition. The first is that the engineering time are much expensive than hardware price. Such a trend is declared by Moore and it has been one reason why the industry grows that fast. When considering the developing and production costs, we should always consider the business cost behind it. The other reason is that we can use better machine for debugging than production. Built on modern compiler and profiler, the memory footprint can be captured and measured. Once we claimed the data, we will be able to use a machine which is enough for certain tasks. Besides, premature optimization is the root of all evil by Donald Knuth.

6 Conclusions

In this section we explain the reached goals described in Chapter 1.

For Goal 1 we fount the most of the material in the FreeRTOS website, reading its documentation. We had problems to get part of hardware (we didn't have Raspberry Pi) for Goal 2 and, also due to time constraints, we had to adapt our project to this situation changing the hardware design (Goal 3). The main problems were in Goal 5 where we decided, after many tests and manual tuning, to start again from Goal 3, with a different design and reaching Goal 4 using the disposed hardware. We were not able to make tests and comparisons between different approaches and scheduling algorithms due to our limitations(Goal 6).

Considerably, according to Moore's Law which holds for decades, it grantees our developers' time are more expensive than such chips. If a development board or production system requires manual tuning, it just won't be economical.

6.1 Future Work

There are many open doors for future works:

- **Replace Arduino with a better board:** As already mentioned we had problems due to Arduino's memory limitations, so we want to replace Arduino Uno with a different board, maybe another version of Arduino with more memory.
- **Redo our implementation with Approach A:** With a better board we could be able to implement the initially designed tasks and implement Approach A and build preemptive time slicing scheduling (Approach B) to compare with.
- **Build foreman project:** We want to use Raspberry Pi to implement the foreman project and simulate the smart trafic light.
- **Log timing to analyze schedulers:** With the hardware just described we can have Logs messages to analyze and debug schedulers.
- **Combine two scenarios with two approaches:** It could be also possible to add more scenarios (cars that don't break at trafic light and other danger cases) and compare them with the different approaches to find the one which fits best with our project.

References

[1] G. C. Jean Dollimore Tim Kindberg, *Distributed Systems: Concepts and Design (4th Edition): Jean Dollimore, Tim Kindberg, George Coulouris: 9780321263544: Amazon.com: Books*.

[2] Y. C. Chang. (). ⊙⊙⊙⊙ nagasaki city. licensed as CC-BY-NC-ND 2.0, [Online]. Available: https://www.flickr.com/photos/37534750@N06/35235282661/in/photolist-VFC2Xt-pzWiQs-XXoE4W-ixo8Ty-BW4AyC-nHuW3V-jruSDu-XQN4i7-o4atQ6-HcNvcB-tKmVw-g3Y36y-aseUFq-dczyaJ-bxjbfo-5vjy4r-6pN9ZK-okAx61-NMQPu-dJ4riK-6jq5yr-f7iDcM-NNuE5-6jUeYr-9HAXoN-niuvH6-e1mxrT-pC82dY-pCZF1k-5tquPh-BU6mgh-5vw6bm-XBxn4p-nid4hE-7u1QRg-rjR1jX-rMcxC3-caSMcq-R8imT-aRChWV-4SyAa3-obcpdB-9wnJTP-YGA1zU-atkJE5-nVbDd7-fTSebM-u9Qx-32MRC3-e9rGHs (visited on 10/04/2017).

[3] (). Stopping distance, [Online]. Available: http://www.sdt.com.au/safedrive-directory-STOPPINGDISTANCE.htm (visited on 10/12/2017).

[4] (). Overall stopping distance for vehicles travelling at high speed, [Online]. Available: https://www.volvoclub.org.uk/pdf/SpeedStoppingDistances.pdf (visited on 10/12/2017).

[5] (). Volve c30, [Online]. Available: http://fastestlaps.com/models/volvo-c30-t5 (visited on 10/12/2017).

[6] (). The freertos tick – unravel the mystery to master the tick, [Online]. Available: http://www.learnitmakeit.com/freertos-tick/ (visited on 10/12/2017).

A Code and Environment

The code is not shown here since minted tends to be broken with long lines.

The system being used is:

- Hardware Model: 15 inch, Mid 2015, 2.5 GHz Intel Core i7
- Operating System: macOS 10.12.6
- Operating System: Windows 10

FreeRTOS by Richard Berry 9.0.0-1

Writing Notes