# Traffic Light System

Ana Catarina Cardoso M00634184
Patrick Falcon M00668092
Rojhat Sipan M00643413
Smith Rajesh D'Britto M00689896

23rd November 2018

# Contents

# 1 Introduction

## 1.1 Software Used

Our Traffic Light System was made possible by using Windows as an OS, DrRacket as an IDE for our code and finally Git Bash so that we can push our code from our machines to GitLab.

## 1.2 Languages

Our system runs primarily on Racket but does reference JavaScript Object Notation (JSON) to run simulations. JSON is not a language however, it is a data-format which shares some of the syntax from JavaScript

## 1.3 Aim and Objectives

Our primary objective was to design and implement a Arduino system that runs a traffic light system. Our goal was to ultimately to have a fully automated system that could run independently without human interaction*. * - *Human interaction on the code, we have an input but that process is also automated by the code and will still function even if the button were not in place*

## 1.4 Equipment Used

To make the project a success we used multiple pieces of equipment which served a variety of purposes.

**1 - Arduino Board**: The Arduino Board is a vital part of the entire project. Arduino is a microcontroller that allows us to power the *breadboard*. The Arduino gets its power from a connected power source, this could be anything ranging from a portable mobile phone charger to a laptop

**2 - Breadboard**: The breadboard is used as a testing platform and is widely used for prototyping in the real world. Once the breadboard has been used people will use their own boards, we could not do that however so we continued to using the breadboard. Using the breadboard was a great asset as we are able to change wiring quickly and effectively. If we would have used a board with digital wires then we would not be able to change anything once

it had been produced

**3 - Laptop**: We used a laptop as our power source as well as a deployment tool. We deployed our code directly from the *DrRacket IDE* using the Asip library. This allowed us to debug our code in a timely manner.

**4 - 3D Printer**: A 3D printer was necessary to our project as we needed to produce our designs for the traffic light. This was a cheap and effective way of producing our designs.

**5 - Wooden Board**: Our wooden board acted as a base for the project. All of the components were glued onto the wooden board. Wires were fed underneath the board, this allowed us to keep a clean surface for our traffic light system

**6 - LEDs**: LEDs were used as a way to output what state our traffic light was currently in. We had soldered *electronical wires* to the light in order to make them longer and able to fit into the traffic light design

# 2 Physical Design

Our design consists of multiple elements, consideration went into both the design of the physical board as well as the code that was used to run the system effectively.
We wanted to create a simple system that was effective and reliable, throughout the design process we faced many issues. These issues range from not having enough spare GPIO* pins on the Arduino board to how many breadboards we use. The final design is simple and elegant. We linked four breadboards together which allowed us to have a neater cable layout, this made our project both presentable and effective.
* - *General-purpose input output*

# 3 Racket Design

Racket was the primary language that was used in the creation of our traffic system therefore it consisted of multiple classes that were written and depend on both our own libraries and third-party libraries.

## 3.1 Arduino Manager

Arduino Manager* is the backbone of our project, it was written by us to work with the *Asip Library*. This was a crucial part of the project because AMAN managed the code that could be seperated from our execute file so that we could keep a nice, clean and readable class. Arduino Manager was designed to handle the important fnctions such as JSON and Gregor so that we did not have to import those libraries into the execute file.
* - *Usually referenced as AMAN or AMAN.rkt*

## 3.2 Execute

Execute is our main class that is executed once we run the system. Execute also handles all of our transitions between the lights.

## 3.3 Theory of design

Our general design follows a simple pattern, our system ran on a central traffic light which would determine the rest of the states. We can call this light $S_1$.

$S_1$ would be placed into a state, this state could be RED/REDAMBER/GREEN/AMBER. Once this has occured we can then start to change the other lights depending on the state of $S_1$.

Using this method of overall design allowed us to use the least amount of code possible as we really only needed to change one light rather than the 6 lights that we actually had on the board. This allows us to use only one Arduino rather than multiple Arduinos to output the right amount of lights.

## 3.4 Documentation: AMAN.rkt

In this section we will be going over the documentation for the AMAN.rkt class

```
(require "AsipMain.rkt")
(require gregor)
(require gregor/time)
(require json)
```

By using these lines of code we are importing all of the required libraries that are needed in order for our project to work. Gregor manages timings, gregor is able to get the current time. JSON is the library that is used to parse JSON files, this is necessary later on once we actually load the files

```
(define set-ports (lambda (v1 v2) ... ))
```

Set-ports is a dynamic method of registering the pins that are needed throughout the project. *V1* is parsed as a list, a for loop will then go through each of the values and register them to *V2*. *V2* will be either 'in' or 'out'. This is a shortened version for OUTPUTMODE and INPUTPULLUPMODE.

```
(define in INPUT_PULLUP_MODE)
(define out OUTPUT_MODE)
```

These are the shortened methods that saves time whilst typing

**JSON Functions**

```
(define simulation 0)
(define simFile "null.json")
(define load-file null)
(define set-simulation (lambda (v1)
         (set! simulation 1)
         (set! simFile v1)
         (set! load-file (call-with-input-file simFile read-json)
)))
```

The above definitions are used as a way to set up our 'simulations'. Simulations are just JSON files with a specific value which will allow us to simulate certain conditions. As mentioned in the **Removed Features** section, weather was originally supposed to be located in the JSON file but was removed due to constraints with the API.

```
(define get-value (lambda (v1 v2)
         (cond
                  ((equal? load-file null)
(raise "no simulation loaded")
                  )
         (hash-ref (hash-ref load-file v1) v2)
))
```

get-value is an example of a nested function. This is because we call a function within a function, this is used with *hash-ref*. get-value uses the JSON parser to extract information from the JSON file, we will be gathering time information using this function

**Vital functions**

```
(define c (lambda (v1 v2)
         (digital-write v1 v2)
)
```

The function c is used as a shortened method for *digital-write*

```
(define turnOn (lambda ()
         (for ([i ports])
                  (c i 1)
         )
)
```

turnOn uses *c* to change all of the registered ports (which are saved in a list) to turn on all of them at once. There is also a function for turning them off that uses the exact same method just using a 0 rather than a 1.

```
(define connect (lambda ()
         (display ... )
         (display ... )
         (display ... )
         (setTimes)
         (open-asip)
)
```

connect is to be called at the start of the execute class. This is because without connecting to the board we cannot actually set the ports as we will get a contract violation back.

## 3.5 Documentation: execute.rkt

In this section we will be going through the code for the execute.rkt class which is used to run the code.

```
(connect)
```

Connect is a functin which was defined and provided in the AMAN.rkt class, by calling it here we are able to connect to the Arduino board.

```
(set-ports '(13) in)
(set-ports '(12 11 10 9 8 7 6 5 4 3 2) out)
```

These two lines are code are important as they are setting all of the pins up that are being used throughout the project. We created a nice little shortcut by using 'in' and 'out' instead of the methods that were provided within the Asip library.

```
(define pushed 0)
```

This variable is used to represent a binary value, 0 being OFF and 1 being ON. The reason why this is used is because we need to be able to save a state that the system is currently in which can be easily changed. By using $(set!pushedx)$ we are able to change the value which will allow us to have a dynamic state.

```
(define i 0)
(define checkb (lambda () ... ))
```

checkb is a crucial method in the execute file. This is because it is needed so that we can check for a physical input *(button)*. checkb is a recursive function, this means that it is called within itself. However with the introduction of $i$ we are able to stop the recursion after $i$ reaches twenty.

```
... (cond
((equal? (digital-read 13) 0)
...
```

This conditional is checking whether our GPIO pin has a value of 0. For whatever reason Asip returns a value of 0 whenever the button is pressed, therefore we need to use the value 0.

```
... (cond
((< i 21)
(sleep 1)
(set! i (+ i 1))
(checkb)
)
)
...
```

The above conditional is where our recursion is called, we are checking for 20 seconds, the reason why we have put 21 is because we used the '¡' symbol. This means it will check all values up to 20 rather than 21. If we were to put 20 then it would only be up to 19.

## 3.6 Removed Features

Throughout the project we had discussed many potential features which were ultimately scrapped due to lack of time, resources or just adding complexity to the project that were not as important as our primary features.

**Proximity Sensor**: A proximity sensor was originally going toe be heavily involved within our project to allow a smarter system to be developed. The original concept was to install a proximity sensor at each light. If, for example, sensor $A$ detected a car whereas sensor $B$ did not then we would be able to change the light that was related to sensor $A$. However we had a huge constraint relating to time, therefore we could not actually implement this feature as we would not have had enough time to perfect our core features.
**SQL Backup**: An SQLite database* was originally going to be implemented so that we could backup the state that the traffic light was in. This feature was considered as we wanted to created a reliable system, this reliable system

would need to remember what state it was in so that if in the event that it crashed then we could pickup from exactly where we left of. This would have been a simple but effective feature if this were a real world traffic light system that could fix practical issues.* - *An SQLite database is a local database that would not require third-party hosting like how an SQL3 server would*

**Weather Detection**: Similarly to how we had a time detection to change the rate at which the lights change, we also had an idea to change the rate depending on the weather. This feature was in the final stages of being implemented, however we could not get a live reading as the API which we were using was very limited without purchasing a subscription. This lack of resource caused us to not be able to use weather detection.

# 4 URLs

*For a more in-depth look at the code there is a GitLab repository that has been setup with all of our code.*
*GitLab Repository: https://gitlab.com/mdx-project/traffic-light*
*Gregor Library: https://github.com/97jaz/gregor*
*Asip Library: https://github.com/fraimondi/racket-asip/*