Traffic Intersection Optimization

Final Report

Team Name: HIIIII-5

Group: 5

Callum Chan (20709934)
Rachel Wong (20702194)
Wesley Barton (20709983)

Table of Contents

(1) Project Overview

1.1 Preface

In today's ever evolving world, despite the enormous advancements in technology, the remarkably elevated standard of living, and the luxuries that come with living in the 21st century, we are faced with an unprecedented crisis on a global scale. The planet is warming and we aren't doing enough to stop it. While the factors contributing to this phenomenon are numerous and complex, there is one that is reasonably approachable: idling. Idling occurs when a car is not moving but the engine is still running. As one can imagine, idling is extremely wasteful and routinely occurs around intersections. If traffic lights could be optimized in such a way that would reduce unnecessary wait-times, not only would it have an impactful effect on planetary health, but will also decrease gas spending for drivers and cut down on commute times.

Optimizing an intersection is no small feat, everyone is different and as a result, there is no one-size-fits-all. However, even the most basic forms of wait-time reduction, will contribute to the prevention of greenhouse gas emissions filling the atmosphere.

1.2 The Idea

For this concept to be applicable on the streets, sensors would need to be installed on all four sides of the intersection to detect cars as they pass by. If the sensors detect that no cars have passed through one way of the intersection after a certain amount of time, the lights will automatically switch. Furthermore, should there be a steady stream of cars, there will be a default time at which the lights will swap. By changing the lights prematurely if no car has passed, time will be saved since the traffic lights do not have to wait for the default time interval before switching.

Additionally, statistics over the intersection are also very useful tools when trying to reduce unnecessary wait-times. For example, having the intersection calculate when one way of the intersection is at its busiest and extending the green light period accordingly. These statistics are not only valuable for sole-intersection optimization, but are also beneficial for urban planners who strive to enhance city systems as a whole. Finally, expanding our perspective to encompass entire blocks or streets, a network of such intersections linked together passing around information, for example, construction zones, vehicular collisions or heavy traffic, and adjusting each of their own timings individually would incorporate this concept into a much grander scale.

1.3 Subset

In our scaled down version of the aforementioned traffic optimization concept, red and green LEDs represent traffic lights and ultrasonic range sensors are used to detect the toy cars passing by. For the sake of redundancy and the limitations on the available GPIO pins, only half of the sides, the northward and westward lanes, of the intersection were constructed. A basic optimization process was also implemented into the project to decrease needless idling and time stopped at a red light. This process switches the traffic light from green to red, and red to green at the other light, if 10 seconds has passed and no cars have crossed the intersection.

Finally, statistics were computed both, over each interval of green light and over the entire simulation. Examples of statistics calculated include: time of interval, cars passed over the interval and total time saved. Despite the fact that the process of optimization used is rather basic, the underlying idea that traffic intersections can be modified to reduce greenhouses gases is still present and can be expanded upon to increase its effectiveness.

(2) System Design

The system design as a whole can be broken down into a few key components. There are two breadboards, each with a green LED, a red LED, and an ultrasonic range sensor wired to them. Each breadboard is used for the north and west directions respectively. The sensor detects whether or not a car is passing through the intersection and the LEDs represent the traffic lights. During each time interval, the software counts the cars that pass and also calculates the cars per second. At the end of each day (or simulation time), statistics over the whole day (or simulation) will be calculated. These statistics are mentioned in section 3.4 in detail. More detail on the hardware and software components can be found below.

2.1 Hardware Component

Materials

- Omega 2 Plus
- Power Dock
- Breadboard x2
- 1KΩ Resistor x6
- 500Ω Resistor x4
- Jumper Cables x14
- Ultrasonic Range Sensors x2
- 3mm Red LED x2
- 3mm Green LED x2

Note Circuit diagrams will be omitted from this report as we do not have sufficient experience with them and this is a software course.

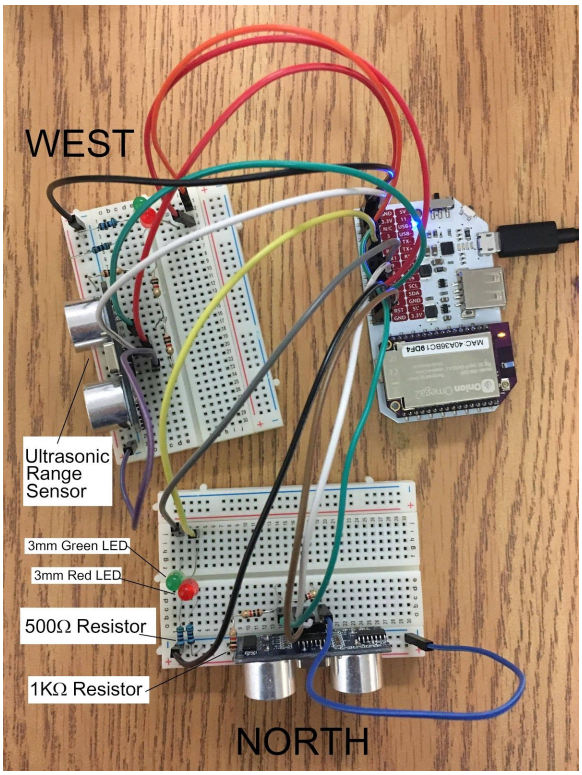

Diagram 2.1.0
Overhead View of System

Component Mapping

Component	GPIO Pin #
North Green LED	18
North Red LED	46 (RX1)
North Sensor Input (Echo)	2
North Sensor Output (Trig)	19
West Green LED	3
West Red LED	1
West Sensor Input (Echo)	0
West Sensor Output (Trig)	11

Functionality of Main Hardware Components

The Omega: The Omega board is the brains of the embedded system. It provides the bridge between the software and the hardware. All GPIO pins are located on the power dock. The GPIO pins provide a way to control the current through our hardware components.

The Sensor: The sensor that was used for the project was an Ultrasonic Range Sensor. The basic functionality of the the sensor is divided into three simple steps. First, the sensor must be triggered by the Omega board. This is done by sending a current through the "Trig" component of the sensor (see diagram 2.1.1). The current is provided by setting the GPIO pin that is attached to the "Trig" wire to HIGH. After the sensor has been triggered, the current is stopped and the sensor sends out a sound wave. As soon as the sensor has sent out a sound wave, it sends a current through the "Echo" (see diagram 2.1.1) which is connected to a GPIO pin (as an input) on the Omega. The sensor continues to send a current until the sound wave has returned to the sensor. Once the sound wave returns to the sensor, it stops sending a current to the Omega. The software then determines how long the sound wave took to reach back to the sensor and determine a (relative) distance of the object it detects. The sensor is used in this project to determine whether or not a car

has passed through the intersection. The sensor is quite inaccurate in this respect. Limitations will be discussed later in the report (4.2).

Diagram 2.1.1

Picture of Ultrasonic Range Sensor

The LEDs: The LEDs have a very simple mechanism. They are connected to an output GPIO pin on the Omega board. When it is necessary to turn on an LED, a current is passed through the LED's circuit by setting it's GPIO output to HIGH. When it is necessary to turn off an LED, the GPIO output is set to LOW and current stops passing through its circuit.

The Resistors: The resistors are used to prevent damage to the LEDs and sensor. Different resistances were used for the LEDs and for the sensor. 500 ohm resistors were used for the LEDs in order to limit the current that passes through the LED, preventing damage to them. 1000 ohm resistors were used for the sensor.

2.2 Software Component

Functionality of Main Software Components

Reading from the sensor: On the hardware end, all the sensor does is detect whether or not something is within its range (approx. 4 meters). On the software end, a relative distance must be determined based on the amount of time that the "Echo" is inputting a current to the Omega's GPIO pin. This is done by measuring the time in microseconds (using the usleep function) in small increments starting from when the "Echo" begins to input to the Omega. These microsecond increments are accumulated until the "Echo" stops inputting to the Omega (meaning the sound wave has reached back to the sensor). Once that happens, a relative distance is calculated by divided the time by a factor equivalent to twice the speed of sound in cm/s. This distance is not a true distance, but it provides us a relative distance which is enough to determine whether or not a car has passed by the sensor. At regular intervals of about 100 milliseconds, the sensor detects whether or not a car has passed (only if the light is green through that direction of the intersection). If a car has passed a counter gets incremented.

Turning on/off the LEDs with timers: Turning on and off the LEDs is a simple process, which was described briefly in the hardware component section. The LED gets turned on when the software sets the GPIO output pin to HIGH. This can be done as necessary at any point in the program's runtime. Similarly, to turn off an LED the software sets the GPIO output pin to LOW. Of course, at an intersection, the lights change at certain time intervals. The software handles this fairly easily. It starts off by getting the current time of day (in seconds) at the beginning of the intersection interval, this is the start time. At regular intervals, the current time of day is determined again and the difference between the start time and the current time is also determined. If the difference between them is sufficient enough to warrant a light change, then the state of the intersection changes (i.e. the lights change).

Optimization: The main goal of this project was to limit the wait time at an intersection. The optimization technique used is as follows; if a car has not passed through a green light in a time limit of 10 seconds then the light will switch to red before the normal default time interval (30 seconds). The timers and light switching is done through the software the same way as described above. The only thing that changed with the optimization timer is the fact that it needs to be updated each time a car passes. Everytime a car passes, the "start time" of the optimization timer gets updated to the current time of day (in seconds). If the timer has not been updated, it means that a car has not passed through a green light. Once the difference between the current time and the optimization timer is greater than 10 seconds, the state of our intersection changes (lights change). This overrides the default timer.

Computing Statistics and File Writing: After each time interval where the light is green, a few things are being done before the light switches to red. The first thing that is done is the number of cars passed and the time interval get stored in a temporary structure (detail on the structure in 3.2). Using that structure, the amount of cars per second over that time interval is calculated and stored in that structure. The structure is then put away in an array containing all the raw data from the intersection. Raw data is kept separate for each direction (north and west respectively). When the day is over (or the simulation is over), all the raw data for each direction is used to compute statistics over the day (or simulation). These statistics include the maximum cars per second, and the amount of time saved. The statistics are calculated separately for each direction. Once these statistics are calculated, four different files are written to the Omega. Two raw data files, which contain the raw data for each direction of the intersection (cars passed, time interval, cars per second). The other two files are the statistics over the simulation for each direction. More detail on the file writing process is in 3.3.

Logging: The software also provides a way to debug the embedded system, since a console is usually not present in an embedded system. The way logging works for this project is based on a file writing function. When something needs to be logged, the writeToLog function is called and it writes the specified log message to the log file. The function takes in multiple parameters, some of which are optional, depending on what needs to be logged. More information on the logging structure is in section 3.5.

(3) Software Design

3.1 Overall Functionality

The intersection that was created for this project runs based on a very simple state machine (diagram 3.1.0). If the light is green in the north direction, then it must be red in the west direction. This works the same the other way around. For each iteration of a green light, the program keeps track of how many cars have passed by in the respective direction. The program also checks if the timers have passed their certain time thresholds (as described in 2.2). This process of checking if a car has passed and checking if the timer has reached it's end is all done in a while loop in the respective state. Once the default timer (30 seconds) has ended or the optimization timer has ended, the while loop is exited. This signifies a change in state. Before a change of state occurs, the program records all raw data for that iteration of the light and stores it in an array for later use. After this is completed, the state changes (green turns to red in current direction, and red turns to green in other direction).

This back and forth state shifting continues until the simulation time is finished (this is currently passed in by the user). In a real life situation this process would run indefinitely of course. Once the simulation is over, various statistics get computed for each direction of the intersection using the raw data obtained throughout the simulation. Once that is completed, the raw data and the statistics are written to four different files as described in 2.2. Logging also takes place throughout our program at various locations that are necessary for debugging. The program has several functions (see 3.3 and 3.4 for detail) which all get called by our main() function which contains our state machine (see diagram 3.1.1). A basic flowchart for the overall functionality of our program can be seen in diagram 3.1.2.

System Dependent Components

The system dependent components of the project deal with the hardware components used in our specific intersection. System dependent components are those that rely on the system in order to work properly. These system dependent components cannot be tested on any regular system and requires the use of external hardware for testing. The system dependent functions in our project are those that interact with the hardware specifically. These include, turning LEDs on and off, reading data from the sensor, and checking if a car has passed by. These functions are reliant on the ability to obtain or provide information to the hardware and as such are system dependent components. In order to test the overall functionality of the project without the hardware, dummy functions are used in order to mimic a hardware output or input.

System Independent Components

The system independent components of the project deal with statistics, file writing, and timers. System independent components are those that do not rely on external hardware for testing and can be tested separately from the system. In this project system independent functions include all the statistic computing functions, the statistic/raw data file writing functions, the logging function, and the timer functions. All of these functions can be tested independently of the system as they do not rely on any hardware interaction. Therefore, all of these system independent functions can be written and tested prior to having any hardware. These function are also easily implementable in any other system, since they are system independent components.

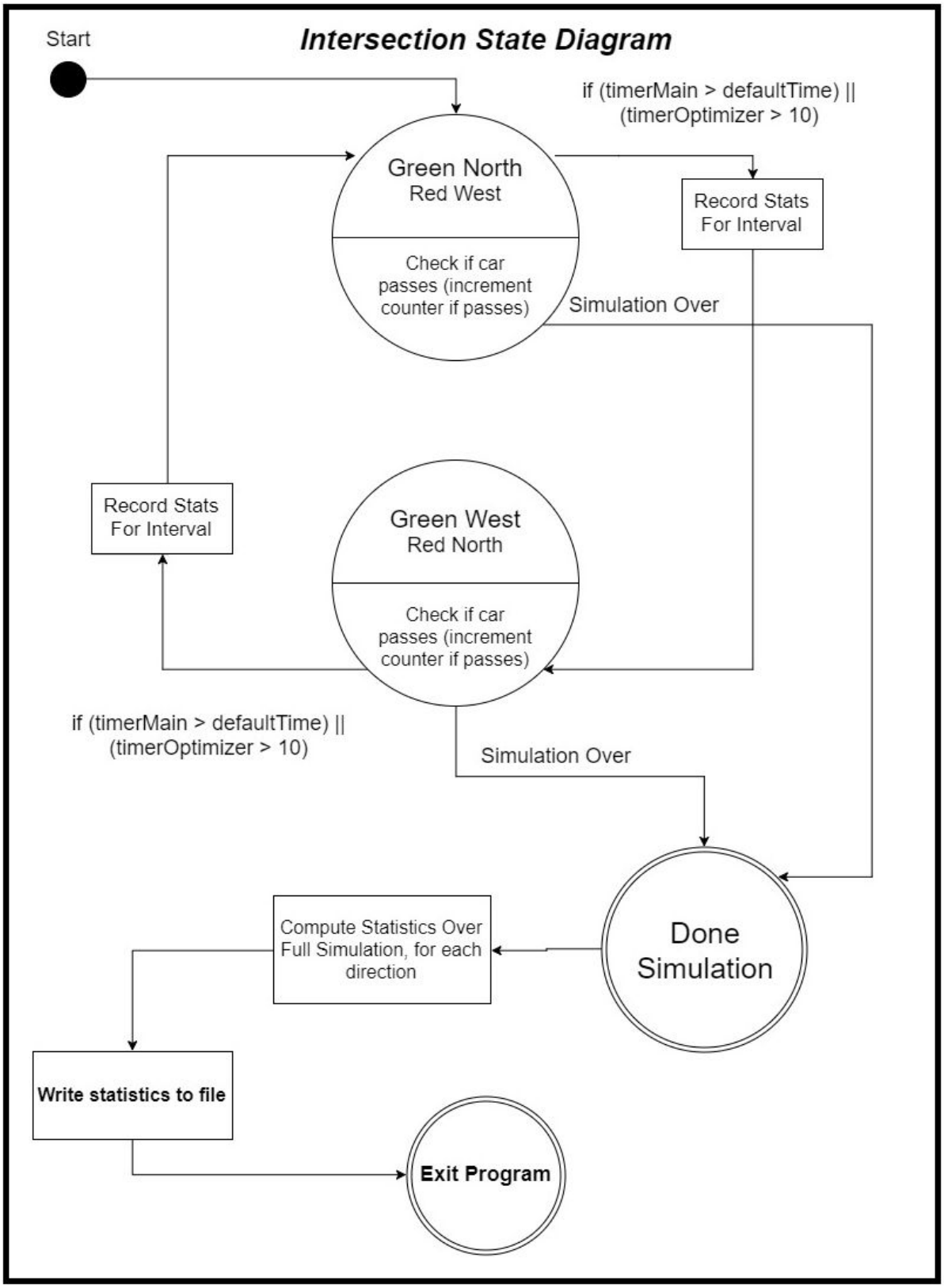

Diagram 3.1.0 State Machine Diagram

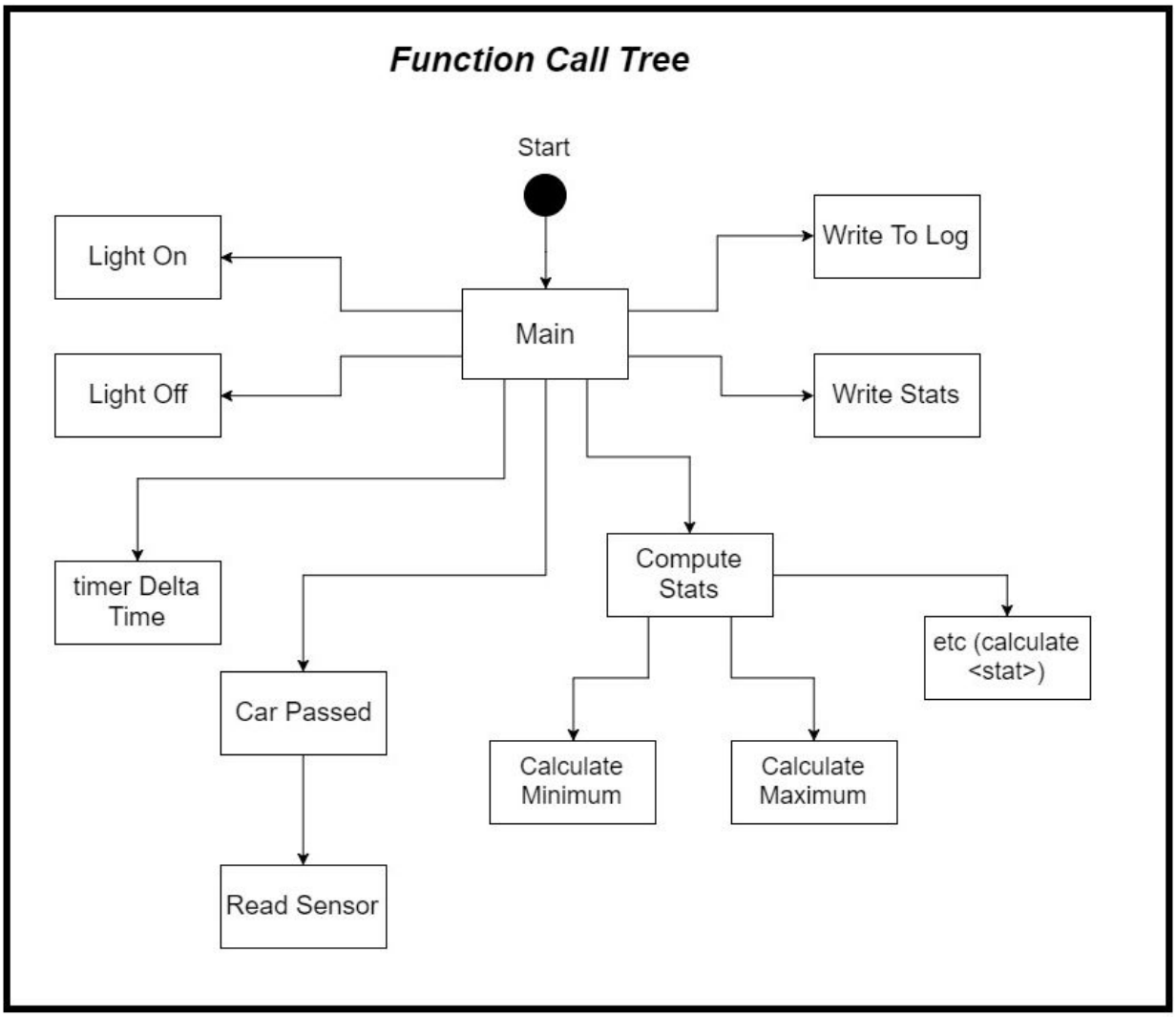

Diagram 3.1.1 Function Call Tree for Main()

General Program Usage

./traffic <simulationTime> <logDegree>

-This will set the simulation time of the program to <simulationTime> (in minutes) and set the degree of logging to <logDegree>

./traffic <simulationTime>

-This will set the simulation time of the program to <simulationTime> (in minutes)and the degree of logging to 0

./traffic

-This will set the simulation time of the program to 5 minutes and the degree of logging to 0

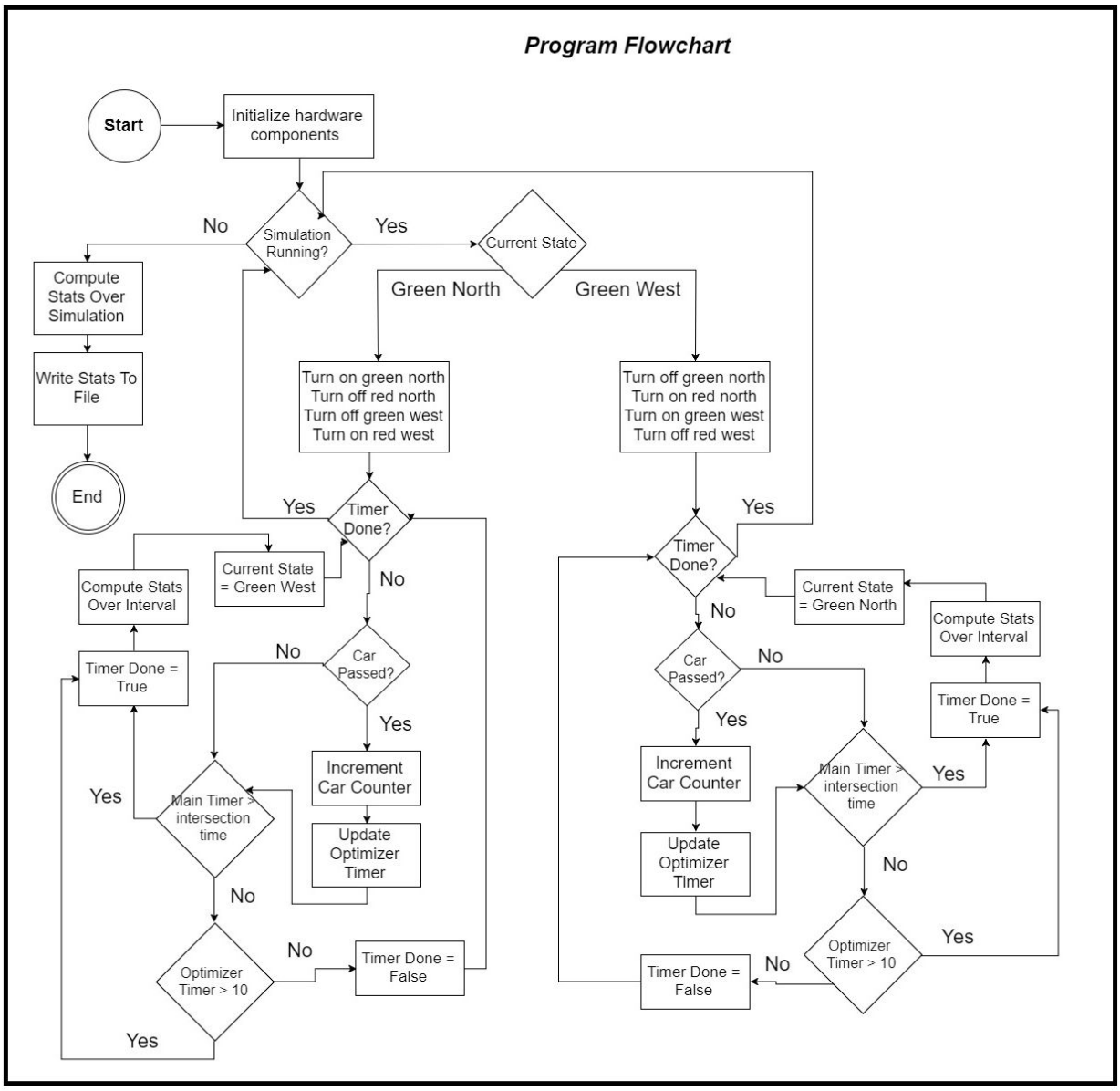

Diagram 3.1.2 Flowchart displaying overall functionality

3.2 Structures and Interface

(1) Stats Over The Interval

Definition:

```
struct StatsOverInterval
{
     int numCars;
     float timeInterval;
     float cps;
};
```

Description: Keeps track of the raw data for each iteration of a green light. *numCars* represents the number of cars that have passed by. *timeInterval* represents the time that the light was green. *cps* represents the cars per second that passed by.

(2) Stats Over The Simulation

Definition:

```
struct StatsOverSimulation
{
     int totalCars;
     float totalTime;
     int maxCars;
     int minCars;
     float averageCars;
     float averageTime;
     int modeCars[10000];
     int numModes;
     float maxCPS;
     float minCPS;
     float avgCPS;
     float medianCPS;
     float popStdDevCPS;
     float smplStdDevCPS;
     float timeSaved;
```

};

Description: Keeps track of the statistics over the simulation for each direction of the intersection. *totalCars* represents the total amount of cars that passed in that direction. *totalTime* represents the total time the light was green in that direction. *maxCars* represents the maximum number of cars that had passed by. *minCars* represents the minimum number of cars that had passed by. *averageCars* represents the average number of cars that had passed by. *averageTime* represents the average time the light was green in that direction. *modeCars[]* represents the modes of the number of cars that passed by. *numModes* represents the size of the modeCars[] array. *maxCPS* represents the maximum cars per second that had passed by. *minCPS* represents the minimum cars per second that had passed by. *avgCPS* represents the average cars per second that had passed by. *medianCPS* represents the median cars per second that had passed by. *popStdDevCPS* represents the population standard deviation of cars per second that had passed by. *smplStdDevCPS* represents the sample standard deviation of cars per second that had passed by. *timeSaved* represents the amount of time saved (in seconds) in that direction.

3.3 Core Operating Functions

The core operating functions are functions that are required for the functionality of our intersection. Each function will be presented with a function prototype and then a brief description.

System dependent functions are marked with a (🔵) and system independent functions are marked with a (🔴)

(1) Light On 🔵

Prototype:

bool light_on (const unsigned int port)

Description: Turns on an LED at GPIO port ‘port’. *port* represents the GPIO port where the LED is located.

(2) Light Off 🔵

Prototype:
`bool light_off (const unsigned int port)`

Description: Turns off an LED at GPIO port ‘port’. *port* represents the GPIO port where the LED is located.

(3) Calculate Delta Time 🔴

Prototype:
`int deltaTime (const int oldTime)`

Description: Returns how much time has past from a certain time in seconds. *oldTime* represents the time from which you want to find how much time has passed.

(4) Update Time 🔴

Prototype:
`int timeUpdate ()`

Description: Returns the current time in seconds.

(5) Convert Minutes to Seconds 🔴

Prototype:
`int minutesToSeconds (int minutes)`

Description: Converts minutes to seconds and returns the value in seconds. *minutes* represents the time in minutes.

(5) Read Sensor 🔵

Prototype:
float readSensor (const unsigned int gpioIn, const unsigned int gpioOut)

Description: Reads the sensor and returns a relative distance (helper function for carPassed function). *gpioIn* represents the port where the Echo of the sensor can be found. *gpioOut* represents the port where the Trigger of the sensor can be found.

(6) Did Car Pass? 🔵

Prototype:
bool carPassed (const unsigned int gpioIn, const unsigned int gpioOut, const float threshold)

Description: Calls the readSensor function and determines if a car has passed by. If the relative distance returned from the readSensor function is less than the passed threshold, it will return true (car passed). If it is greater than the threshold, it will return false (car didn't pass). *gpioIn* represents the port where the Echo of the sensor can be found. *gpioOut* represents the port where the Trigger of the sensor can be found. *threshold* represents the maximum detected distance that constitutes as a car passing by.

(7) Write Stats and Raw Data to File 🔴

Prototype:
bool writeStatsToFile (char filename[], struct StatsOverInterval statsIntervalNorth[], int sizeNorth, struct StatsOverInterval statsIntervalWest[], int sizeWest, struct StatsOverSimulation statsSimNorth, struct StatsOverSimulation statsSimWest)

Description: Writes the interval stats for north to a file called: filename + "_NORTH_RAW.rawstat". This will loop through the array of interval stats and write each component to the file

Similarly it writes the interval stats for west to a file called: filename + "_WEST_RAW.rawstat" in the same manner. Sample raw data file output for the north direction can be seen in diagram 3.3.0.

It will also write the stats over the simulation for north to a file called: filename + "_NORTH_SIM.stat" and writes the simulation stats for west to a file called: filename + "_WEST_SIM.stat". Sample simulation statistics file output for the north can be seen in diagram 3.3.1.

```
Raw Data for North Direction
x--------x--------x-------x--------x

Time Interval #1:
Number of Cars: 12
Time Interval Length: 30.000000 s
Cars Per Second: 0.400000 cps

Time Interval #2:
Number of Cars: 16
Time Interval Length: 25.000000 s
Cars Per Second: 0.640000 cps

Time Interval #3:
Number of Cars: 18
Time Interval Length: 30.000000 s
Cars Per Second: 0.600000 cps

Time Interval #4:
Number of Cars: 12
Time Interval Length: 10.000000 s
Cars Per Second: 1.200000 cps
```

Diagram 3.3.0

```
Simulation Statistics for North Direction
x--------x--------x-------x--------x

Total Cars: 58
Total Time: 95.000000 s
Max Cars: 18
Min Cars: 12
Average Cars: 14.500000
Average Time: 23.750000 s
Mode(s) # of Cars: 12,
Maximum Cars Per Second: 1.200000 cps
Minimim Cars Per Second: 0.400000 cps
Average Cars Per Second: 0.710000 cps
Median Cars Per Second: 0.620000 cps
Population Standard Deviation Cars Per Second: 0.297153 cps
Sample Standard Deviation Cars Per Second: 0.343123 cps
Time Saved: 25.000000 s
```

Diagram 3.3.1

(8) Write To Log File

Prototype:
bool writeToLog (char filename[], int degreeLogging, int logMessageNumber, char tag[], float value)

Description: Writes a different message to the log file depending on the logMessageNum. If the degree of logging for that logMessageNum is not met then it will not write to the log file. The log file format is: filename.log. More information on logging in 3.5.

3.4 Statistic Functions

All the statistic functions are self explanatory, so a description will be omitted for most of them. All the statistic functions are system independent functions.

(1) Calculate Cars Per Second
Prototype:
float calcCarsPerSecond (struct StatsOverInterval intervalStats)

(2) Calculate Total Cars Passed
Prototype:
int calcTotalCars (struct StatsOverInterval statsInterval[], int sizeStats)

(3) Calculate Total Time Light Was Green
Prototype:
float calcTotalTime (struct StatsOverInterval statsInterval[], int sizeStats)

(4) Calculate Max Cars Passed
Prototype:
int calcMaxCars (struct StatsOverInterval statsInterval[], int sizeStats)

(5) Calculate Min Cars Passed
Prototype:
int calcMinCars (struct StatsOverInterval statsInterval[], int sizeStats)

(6) Calculate Average Cars Passed

Prototype:

```
float calcAvgCars (struct StatsOverInterval statsInterval[], int sizeStats)
```

(7) Calculate Average Time Light Was Green

Prototype:

```
float calcAvgTime (struct StatsOverInterval statsInterval[], int sizeStats)
```

(8) Calculate Modes of Cars Passed

Prototype:

```
int calcModeCars (struct StatsOverInterval statsInterval[], int sizeStats, int modes[])
```

(9) Sort An Integer Array

Prototype:

```
bool sortInt(int dataset[], const int size)
```

(10) Helper Function for Sort Integer Array (selection sort)

Prototype:

```
bool selectionInt(int dataset[], const int size, int index)
```

(11) Sort A Floating Point Array

Prototype:

```
bool sortFloat(float dataset[], const int size)
```

(12) Helper Function for Sort Float Array (selection sort)

Prototype:

```
bool selectionFloat(float dataset[], const int size, int index)
```

(13) Calculate Max Cars Per Second

Prototype:

```
float calcMaxCPS (struct StatsOverInterval statsInterval[], int sizeStats)
```

(14) Calculate Min Cars Per Second

Prototype:
float calcMinCPS (struct StatsOverInterval statsInterval[], int sizeStats)

(15) Calculate Average Cars Per Second
Prototype:
float calcAverageCPS (struct StatsOverInterval statsInterval[], int sizeStats)

(16) Calculate Median Cars Per Second
Prototype:
float calcMedianCPS (struct StatsOverInterval statsInterval[], int sizeStats)

(17) Calculate Population Standard Deviation Cars Per Second
Prototype:
float calcPopStdDevCPS(struct StatsOverInterval statsInterval[], int sizeStats)

(18) Calculate Sample Standard DeviationCars Per Second
Prototype:
float calcSmplStdDevCPS(struct StatsOverInterval statsInterval[], int sizeStats)

(19) Calculate Time Saved at Intersection
Prototype:
float calcTimeSaved (struct StatsOverInterval statsInterval[], int sizeStats, const float defaultIntersectionTime)

(20) Compute All Stats Over Simulation
Prototype:
struct StatsOverSimulation computeStatsOverSimulation(struct StatsOverInterval intervalStats[], int sizeStats)

Description: Collective computes all the statistics over the simulation and stores it in a structure of type StatsOverSimulation (see 3.2) which is returned by the function for use in the statistic file writing.

3.5 Logging Infrastructure

Logging is implemented through the use of the writeToLog function.

Prototype:

```
writeToLog (char filename[], int degreeLogging, int
               logMessageNumber, char tag[], float value);
```

When called, the writeToLog function takes in the filename (a null-terminated array of chars), a logging degree, a log message number, a tag (another array of chars), and a value. Firstly, writeToLog creates or opens a file using the filename passed into it with the extension ".log". Each log number has a particular message associated with it. So, given that the log degree is high enough, a log number's message will be written to the log file once the function is called with that particular log number. A value and a tag are also passed into writeToLog in the case where a particular set of characters or a number are being written to the log file.

For example, let us say the writeToLog function is invoked as such:

```
float numCars = 67;
char tag[] = "numCars";
writeToLog(filename, 10, 13, tag, numCars);
```

The output to the log file would be:

"Value of numCars: 67."

Since the logging degree is greater than 0 and log message number 13 corresponds to the message: "Value of *[tag]*: *[value]*."

The tag and value come in very handy when a number needs to associated with a set of characters. This is useful when logging

a particular variable and its value or when differentiating between the west and north halves of the statistics.

A table with the log number, required degree and corresponding message is given below:

Log #	Degree	Message
0	> 0	Welcome to the log file!
1	> 0	Sensor at port: *[value]*.
2	> 0	Red light at port: *[value]*.
3	> 0	Green light at port: *[value]*.
4	> 5	Time saved: *[value]* seconds, Tag: *[tag]*.
5	> 5	Time for green light: *[value]* seconds, Tag: *[tag]*.
6	> 5	Cars per second during 1 green light: *[value]*, Tag: *[tag]*.
7	> 9	Sensor value: *[value]*, Tag: *[tag]*.
8	> 9	Car passed.
9	> 0	Currently in function *[tag]*.
10	> 0	Exiting function *[tag]*.
11	> 0	Successfully written statistics file *[tag]*.
12	> 0	Simulation terminated.
13	> 0	Value of *[tag]*: *[value]*.
14	> 5	Number of cars in interval: *[value*], Tag: *[tag].*

3.6 Additional Libraries

<stdio.h>

- Defines a plethora of functions that enable the creating, opening, closing and writing to a file.

<stdlib.h>

- Defines atoi() which takes in a pointer to a string and converts it into an int.

- Defines constant NULL which is a null-pointer constant.

<string.h>

- Defines strcpy() that takes 2 char arrays and copies a source array into a destination one.

- Defines strcat() that takes 2 char arrays and joins the end of the first to the beginning of another.

<unistd.h>

- Defines usleep() that takes in a value in milliseconds and pauses the execution until the specified amount of milliseconds has elapsed.

- Defines sleep() that takes in a value in seconds and pauses the execution until the specified amount of seconds has elapsed.

<stdbool.h>

- Defines boolean types and values.

<math.h>

- Defines several mathematical functions that facilitate the computation of statistics.

<limits.h>

- Defines Nan (not a number) which is used as a return value should any of the statistics computed by undefined.

<sys/time.h>

- Defines the timeval structure that contains members who keep track of time.

- Defines gettimeofday() that obtains the current time since January 1 1970 in seconds and stores it in a timeval structure.

<time.h>

- Defines localtime() which takes the seconds since January 1 1970 and converts it into a date and time expressed for the local timezone.

(4) The Results

4.1 Testing

The embedded systems project was tested using the "bottom-up" approach. This process included first writing different functionalities separately and performing individual testing and debugging of the code. We used the necessary valid and marginal test cases to test if our functions compile and work before putting them together for further testing. Using this approach is advantageous since it was ensured that each component of the code works fine. If there happens to be any bugs when combining, it will be clear that the individual components are not at fault.

After that, integration testing was performed. Combining all the software and hardware components, we began to test how the parts worked as a whole. Often when components are combined, problems will appear. Therefore, the code was debugged and tested again so that all of the sensors and lights worked as intended.

Finally, rigorous testing of the project was carried out to test for bugs and malfunctioning components. Many simulations were carried out to experiment with the timing of the sensors, the accuracy of the statistics and the implementation of the optimization. Once we had exhausted the testing of possible edge cases and ran a significant amount of trials, we then concluded our debugging.

4.2 Limitations

First and foremost, the vital sensors used in the project were unreliable and not ideal. The ultrasonic range sensors would often undercount or overcount as the cars passed by. This inconsistency leads to skewed representation of the statistics and inaccurate results. Furthermore, when scaled up into the

real world, many other issues arise using these types of sensors:

1) Each side of the intersection must be limited to 1 lane. For example, if the side of the intersection heading north had two or more lanes, the sensor would not be able detect the cars travelling in the middles lanes either because they are too far away or because they will be blocked by those in the inner lane closest to the curb.

2) Pedestrians, cyclists or other objects that happen to pass in front of the sensor will trigger it and will be incorrectly counted as a car passing through the intersection.

3) Larger vehicles such as flatbed trucks or construction vehicles will be counted as multiple cars due to the fact that the input from the sensor is indicating that an object is within the distance threshold for an unexpectedly longer period of time.

All of the above-mentioned points would undoubtedly falsify the data collected and would in turn produce misleading statistics.

Another aspect of the limitations on the project were the amount and availability of the GPIO pins on the Omega. Only half of an intersection could be constructed. This was in part due to the fact that there aren't enough GPIO pins to support a four-way intersection and because a mirrored setup of the current project to simulate a four-way intersection is redundant.

The project was also limited by its relatively basic optimization technique. For example, if there were no cars waiting at a red light, the light would still change after a default time interval. Adding a functionality that would keep the light green so long as no cars are waiting on the red light would prevent the intersection from wasting time on switching

the lights unnecessarily. Additionally, more complex and thorough optimization techniques could've been used to increase the efficiency of the intersection.

In reality, traffic optimization is done by using a partial outer convexification approach according to paper published February, 2017 in the SIAM Journal on Scientific Computing. However, the complicated calculations behind this approach is well beyond our level of understanding and beyond the computing power of the Omega 2.

Finally, the statistics collected from the intersection were also quite rudimentary. Collecting data regarding the cars per second or the time of an interval is necessary for collecting information such as the busiest or least busy times of day. However, more complex kinds of statistics are required to provide adequate reference for urban planners.

4.3 Lessons Learned

One of the most prominent issues that arose in the creation of this project was the inconsistency of our ultrasonic range sensor. If the project were to start over, the first thing that would be done would the the experimentation of different kinds of sensors. Having more reliable sensors would have made our project much more viable when scaled up to the real world. Furthermore, more accurate statistics would have been produced and which is crucial for a more effective optimization process.

Another aspect that would have been changed is the optimization process itself. More time would be spent creating and designing a more thorough and extensive optimization process. The current optimization technique is quite basic and can be expanded and developed to a much greater extent.

Finally, much more research would have been put into the collection of statistics relevant to urban planners and their occupation. By understanding urban planners and the types of

factors that affect traffic flow, data that is more relevant to this area of study could could be gathered.

Overall, our knowledge and experience of traffic optimization and urban planning is quite limited. Much more time would have been allocated into research and development of the optimization process and the collection of statistics.

However, despite all the changes that we would like to do, there are many things that have been gained from this project. With no prior experience dealing with embedded systems the entire journey from start to finish was a learning experience. Firstly, the entire notion of cross-compiling was completely new to us. The process of creating an executable for a platform other than the one on which the compiler is running was a completely unfamiliar process. Moreover, wiring the hardware also proved to be a challenge. With almost no hardware experience, wiring with the breadboard, determining the placement of the resistors and figuring out how the sensors worked were just some of the things we need to learn in order to complete our project. Although challenging and frustrating throughout, in the end, this project was immensely rewarding.

Even though we could not implement as much functionality as we would like, the end product is still a noteworthy addition to our list of technical experiences. Despite this project pushing each one of our capabilities to the limit, what resulted from it was undeniably worth the effort.

(5) Appendix

5.1 Peer Contribution

Wesley Barton (33.33%)
-Wrote the sensor functionality
-Wrote the statistic file writing code
-Wrote the main state machine functionality
-Wired one of the breadboards
-Wrote aspects of the report

Callum Chan (33.33%)
-Wrote the logging functionality
-Wrote the timer and time delaying functions
-Wrote the turn on/off LED functions
-Wired one of the breadboards
-Wrote aspects of the report

Rachel Wong (33.33%)
-Wrote the all the statistic functions
-Debugged and tested the system
-Major contributor to the overall design of the project
-Drafted and outlined initial and final report
-Wrote aspects of the report

5.2 Source Code

Please see attached source code file traffic.c and makefile.