EEET2162 Design Project

Final Report

Group 7 -
Dale Giancono, Hanuman Crawford, Alexander Knapik
School of Engineering
RMIT

November 3, 2018

Contents

1 Introduction

Traffic, public transport, and intersections are ever so ubiquitous in modern society, and yet are inherently incredibly dangerous. This year, there have been 874 on road fatalities[1] so far. Despite that systems such as traffic lights may seem superficially simple, it is actually quite complex in developing the software for these systems to ensure that they are as safe as possible.

This report delves into the details behind designing a traffic control system for road to road intersections, road to railway intersections and a central controller interacting between the systems, utilising the QNX real-time operating system. The project was intended to develop an understanding and appreciation for the complexity and intricacies behind modern designed intersections with an emphasis on safety.

To understand the requirements of such a system, a problem statement modelled after a real world traffic system was provided by Dr. Samuel Ippolito, and the requirements to implement such a system most safely and effectively were analysed. The features of the QNX operating system in regards to a traffic system utilisied were discussed, as well how such system was implemented. Furthermore, the safety and suitability of such design was evaluated in regards to an ideal system.

2 Problem Statement

2.1 Problem Statement

A system is required to manage two busy traffic intersections modelled after a real world scenario that consists of three roads. There will be a train crossing including a boom gate with flashing lights to stop traffic from passing simultaneously with a train. Each traffic light state change shall communicate accordingly with a central controller to allow for constant monitoring.

The roads consist of two-way traffic with two lanes in each direction. At each intersection, traffic should be able to cross and turn from each position, and there shall be support for pedestrian crossings. Furthermore, two of roads have a greater chance of congestion out of the three due to being major roads; the specifics of which are detailed in figure 1.

The train crossing's boom gate is to communicate with the adjacent traffic light controller nodes about oncoming trains to alleviate traffic blockage, however the traffic light controller nodes are to be operated separately. If there is a fault with the boom gate not correctly opening or closing, the train crossing's controller node shall communicate with with the central controller about the fault.

A central controller will be available for monitoring the status all nodes within the network, as well as controlling the nodes when human interaction is required for traffic congestion, safety, or legal reasons. However, none of the intersection nodes in the network should depend on the central controller for normal operation, ensuring there is not a single point of failure within the system.

All controller nodes, and the central controller are to be programmed utilising the C programming language and for the QNX real-time operating system.

2.2 Assumptions

The decisions made which will be discussed in the following section are based on some assumptions on top of the initial problem statement. Firstly regarding the traffic lights, with the two lanes in each direction, there will not be any turn signals at the traffic intersections. Rather there will just standard green lights, and cars can turn freely whenever there is no opposing traffic.

The train railroad will have trains coming in from both directions, and the boom gates will only be activated by sensors on the railroad, not a timetable due to the unpredictable nature of public transport. Furthermore, there will not be any pedestrian crossings through the intersection between the railway and the road.

The local traffic and train nodes will be running on an ARMv7 instruction set, and therefore won't be binary compatible with an x86 architecture.

2.3 Primitives Discussion

In regards to both between intra-process and inter-nodal communication, the following primitives were chosen for utmost safety. Managing a traffic light system, especially with a train line crossing the road requires deep thinking on how to maintain safety even with partial system failure, as a fault within the system not handled properly, can lead to the death of either a commuter, pedestrian or train passenger.

It was decided that within the same process that data should be bundled and shared via using C structures, passed by reference, and only if access to the structure is required by the **entire** process will it be declared globally.

Shared data required to be accessed by more than one thread simultaneously will be required to be locked with a *timed* read-write lock. This allows for error checking and a fail safe, ensuring that even if a thread has crashed or is locked-up, that the process can handle its failure, without locking out the data completely for all other threads. Furthermore, if a thread failed while writing a data-set under a read-write lock, then the threads reading aren't locked out from reading. Mutual exclusions are faster within QNX as they make up the read-write lock, however the extra redundancy is used to ensure a hard real time system is met.

Communication between processes within and outside of the same node will be handled via QNX's native message passing system. These messages can be sent via the QNET network to other nodes, allowing for communication between the central controller and all the other different local intersection nodes. Furthermore, the QNX native message passing requires a reply after a message is sent, therefore error checking can be applied to determine whether the network is correctly operating, or if there is a fault with a specific node's connection.

2.4 Fault and Failure Tolerance Discussion

Road and traffic accidents are one of the leading causes of death within the world, therefore it is imperative that the systems designed for the public road network is as safe as possible.

Faults regarding state changes within the traffic light and train controller nodes are not allowed whatsoever, as a fatal accident could occur if for example the traffic lights were to skip a step. Therefore, all state changes are to be grey encoded as an extra precaution.

The nodes themselves should not be dependant on any other nodes to function correctly and should be completely self sufficient, this way there will not be any single point of failure within the system. This also includes the central controller which should only be available to monitor and set modes of operation, but never to actively control any other nodes to ensure complete operation of all nodes in the case of a central controller failure.

Sending messages from one node to another node however is the action most likely to fail on occasion within the system, so some precautions should be taken in order to ensure that the action fails the least amount of times as possible. A one-shot client/server system is to be used. At boot up, servers are started on relevant node. As messages need to be sent, the client connect to the server, send a message, receive a reply, and close the connection. This ensures that connectivity problems do not permanently effect the functionality of the entire system.

Messages sent to the traffic light controller from the central controller regarding scheduling changes are not essential the safety, however these messages dropped can be seen as an annoyance to those operating the central computer.

Messages regarding signalling and boom gate faults with the train crossing node are essential to be sent to the central controller. However, there is still the possibility that there may also be a networking fault, and therefore if the central controller is unable to receive any of the messages from the train controller, there must be a fail safe measure for the train conductors, signalling that they are not to cross the intersection until the faults are cleared.

3 Design Discussion

3.1 Intersection Overview

The intersection that the project will be built around involves two sets of traffic lights with a dual train line nested between the traffic lights. The traffic lights are situated on four way intersections.

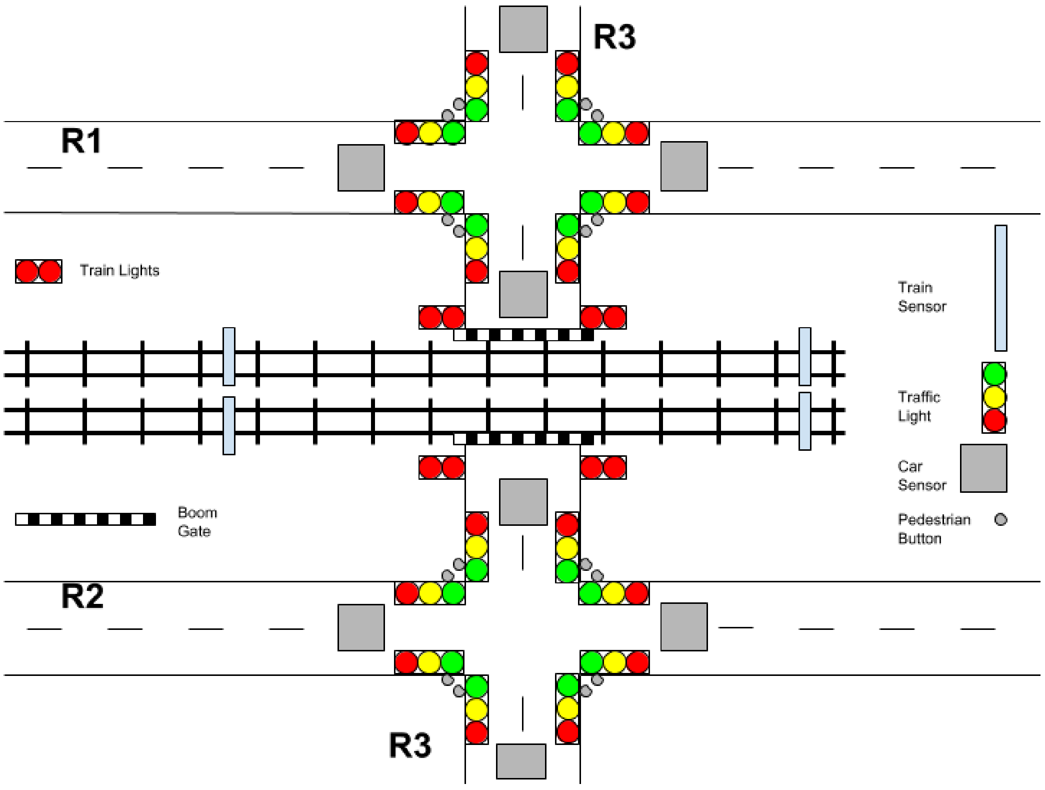


Figure 1: Intersection diagram

The road R3 represents the North-South direction, While the two R1 and R2 roads represent an East-West direction. A dual carriage railway crossing (X1) cuts through the middle of the two traffic lights. Each road can carry two lanes of bidirectional traffic. Pedestrians are able to cross roads R1, R2, and R3 at each traffic light. Cars are able to make turns at each traffic intersection, however dedicated turn lights are not at any intersection. Heavy traffic is expected at road R1 and R3.

The railway crossing has lights and boom gates at each section. Heavy train traffic is to be expected during the day, while light train traffic exists after 2200.

The traffic lights receive information from the railway crossing in order to make informed decisions in calculating the current traffic light state. In the case of a boom gate error, the central control point receives an indication.

Each intersection and crossing operates as an individual node that runs continuously, regardless of network connectivity status. If the nodes are connected, they communicate with a central control point. The central control point is able to override the various error states of the nodes, as well as send scheduling information.

The light sequence patterns for both the traffic lights and railway crossing are dictated both scheduling and sensing. Time scheduling is set by the central controller (or by a default value), while sensing is used to ensure the traffic/railway lights are only changing if absolutely needed. Central control is also able to override the state of scheduling of any node at a moment's notice.

3.2 Context Overview

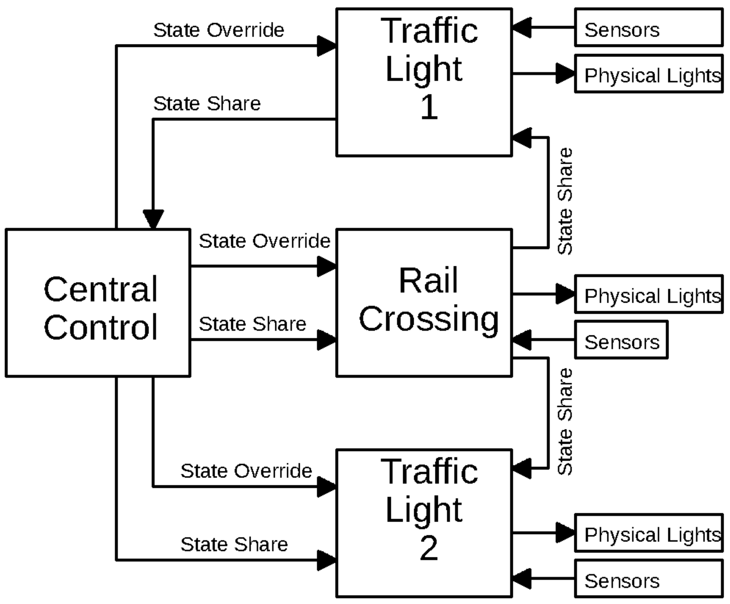


Figure 2: System Context Diagram

The system's design may be split up to 3 main sections, the two traffic light controller nodes, the train intersection and boom gate controller node, and the central controller node. These can then be further separated into their own QNX processes running on separate hardware.

The traffic light controller will be responsible for handling the traffic lights themselves, lights for the pedestrians to cross, and the status updates for the central controller sent via native QNX message passing.

Various inputs can alter the operation of the traffic light controller, these include: Override changes sent from the central controller to change from a sensor or time based schedule. Car sensors will change state for when the node is in sensor scheduling mode. Train sensors will change traffic light state, allowing one side of traffic to flow while the other is blocked from the oncoming train.

The rail crossing controller is responsible for handling the boom gate closing and opening, flashing the boom gate lights for commuters while a train is oncoming, stopping any trains from passing if there is a major fault via traffic lights specifically for the trains conductors, and sending messages to the traffic light nodes when a train is oncoming.

The inputs to the node will be the oncoming and outgoing train sensors, and messages from the central controller such as for overriding.

The central controller's main role is to monitor the status of the traffic lights and train intersection. It does this by requesting the current state from the traffic light and rail crossing nodes when instructed to. From this, the central controller will provide a terminal like user experience to the people in contact with it.

The central controller does not directly control how the nodes within the network operate, but with the traffic light nodes, it can send a message to operate on a sensor, or a timing based schedule. Furthermore, a person operating the central controller is required to override any error states of nodes within the network.

3.3 Use Case Overview

The use case of the traffic light controller is to safely control the flow of traffic, pedestrians and trains through a set of intersections by allowing traffic to pass only when it is safe to do so.

Listed below are a few expected use case scenarios that will be used to describe the usual functionality of the system, as well as some scenarios that may occur in exceptional circumstances, such as a failure in any part of the system or in the event of an accident.

3.4 Use Case Scenarios

3.4.1 Commuter Use Case

Car passes straight through the intersection in any direction. As the car approaches an intersection, the sensor is triggered. If the light is already green the car can pass without any action required. If the light is red the traffic light state machine will react accordingly.

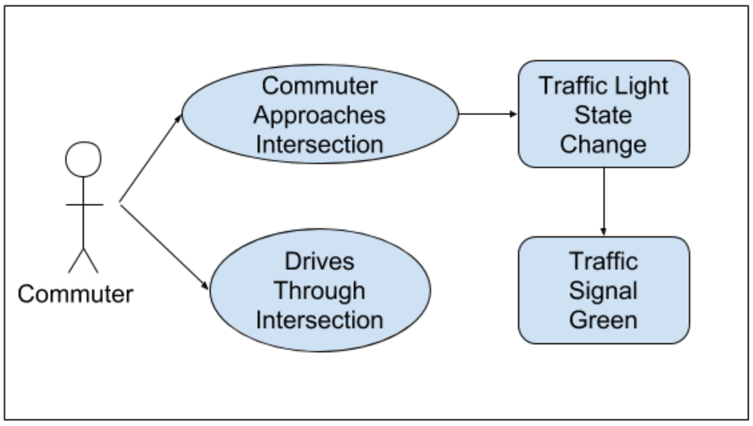

Commuter
Commuter Approaches Intersection
Drives Through Intersection
Traffic Light State Change
Traffic Signal Green

Figure 3: Traffic light commuter use case

3.4.2 Pedestrian Use Case

Pedestrian presses button to cross road, waits for walk light to turn green, then crosses the road. Triggered when the intersection controller receives an input from a pedestrian button. If East-West (EW) pedestrian button is pressed and NS traffic light is green, after the NS traffic light turns yellow, then red, then the EW walk signal will turn green. If NS pedestrian button is pressed and EW traffic light is green, after the EW traffic light turns yellow, then red, then the NS walk signal will turn green.

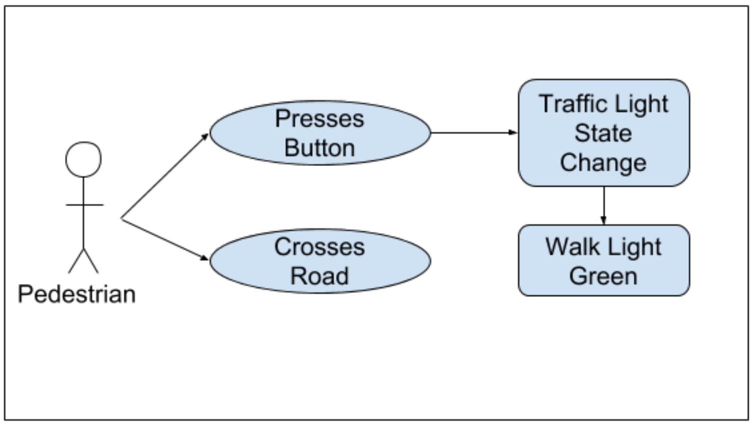


Figure 4: Traffic light pedestrian crossing use case

3.4.3 Train Use Case

Train passes, warning lights turn on, and boom gate activates to prevent traffic from passing in the NS direction. Event is triggered when the train sensor switch is activated and a message is passed to the main controller and the intersection nodes. If the NS traffic light is red, it is prevented from turning green while the train is passing, if the NS traffic light is green when the train sensor switch is triggered it will immediately turn yellow, then red.

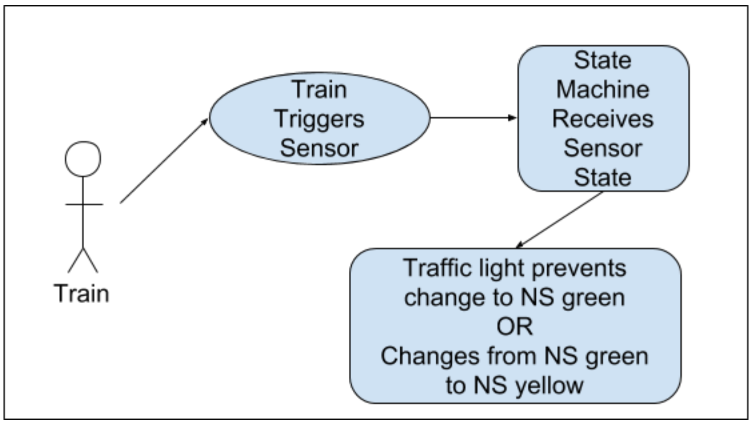

Train
Train Triggers Sensor
State Machine Receives Sensor State
Traffic light prevents change to NS green OR Changes from NS green to NS yellow

Figure 5: Train crossing use case

3.4.4 Train Accident Use Case

A train accident occurs when a train breaks down or crashes in the vicinity of the railway crossing in a way that impacts the ability for cars to cross the railway tracks. A train accident may also mean the failure of the railway controller or message passing, although not necessarily.

If a train accident occurs in a way that does not affect the railway sensing or message passing ability, the system functions as per usual. This would usually mean that the railway sensors would pass a high level to the traffic lights due to the presence of a train. The central control would also be able to send overrides to traffic lights if required.

If the sensor or message passing is impacted, it is up to central control to identify this, and react accordingly by giving the traffic lights the appropriate states to ensure that cars do not approach the railway.

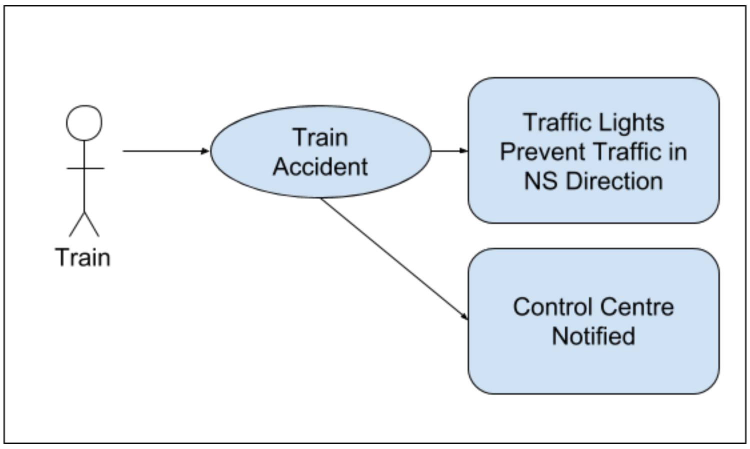


Figure 6: Train accident use case

3.4.5 Traffic Accident Use Case

A traffic accident occurs when a car breaks down or crashes in the vicinity of the traffic light in a way that impacts the ability for cars to cross the traffic lights. A car accident may also mean the failure of the traffic light controller and message passing although not necessarily.

If a traffic accident occurs in a way that does not affect the traffic sensing or message passing ability, the system functions as per usual. This would usually mean that the traffic sensors would pass a high level to the traffic lights due to the presence of traffic. The central control would also be able to send overrides to traffic lights if required.

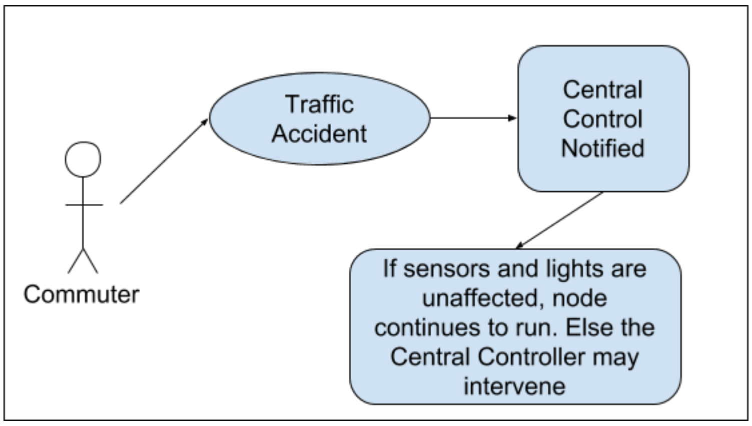

Commuter
Traffic Accident
Central Control Notified
If sensors and lights are unaffected, node continues to run. Else the Central Controller may intervene

Figure 7: Traffic accident use case

3.4.6 Central Control Failure

A central control failure means that the central control no longer can receive or send commands, and information to nodes on the network such as to the traffic lights and railway controllers.

Traffic lights and railway lights/boom gate continue to operate autonomously. Message passing between traffic lights and railway lights/boom gate mean that the system is able to continue to function properly and safely.

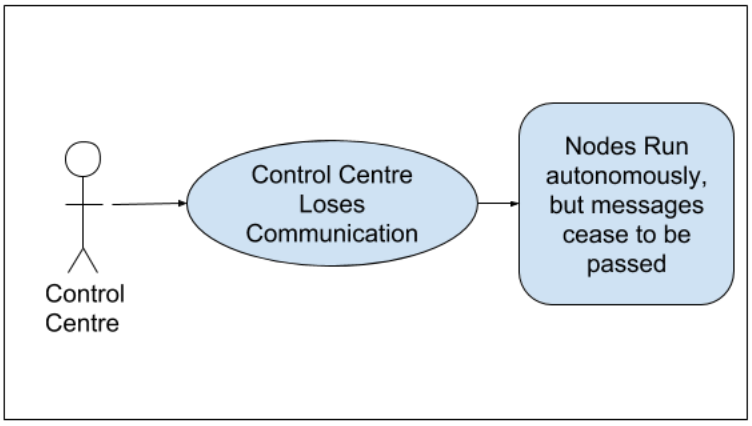

Control Centre
Control Centre Loses Communication
Nodes Run autonomously, but messages cease to be passed

Figure 8: Central control failure use case

3.4.7 Message Passing Failure

In the event that message passing in general fails, and the receiver does not receive a reply to from the recipient, the message is simply dropped. By doing this network congestion is reduced, and systems that utilise message passing are able to continue functioning with no adverse effect.

3.4.8 Train Controller Node Failure

If there is a train controller failure, the central controller is to use message passing to automatically set adjacent traffic lights to red in order to stop traffic from passing the railway and road intersection. The central controller may then override the error state of the traffic lights manually once everything has been confirmed to be okay.

3.4.9 Traffic Controller Node Failure

A failure with the traffic controller node sets the state to an error state where each light at the intersection flashes yellow to signal to commuters about the error. If power to the intersection is lost, then the central controller will not be able to communicate with the traffic controller. Therefore, it will be up to the central controller to identify a loss of power in the node for it to be resolved manually.

4 Design Solution

4.1 Solution Statement

The solution consists of the use of local nodes for the traffic light controllers, railway crossing controller, and central control controller. The two traffic light controllers and the railway crossing controller ulitlise DE10 nano boards. The central controller will use a beagle bone board.

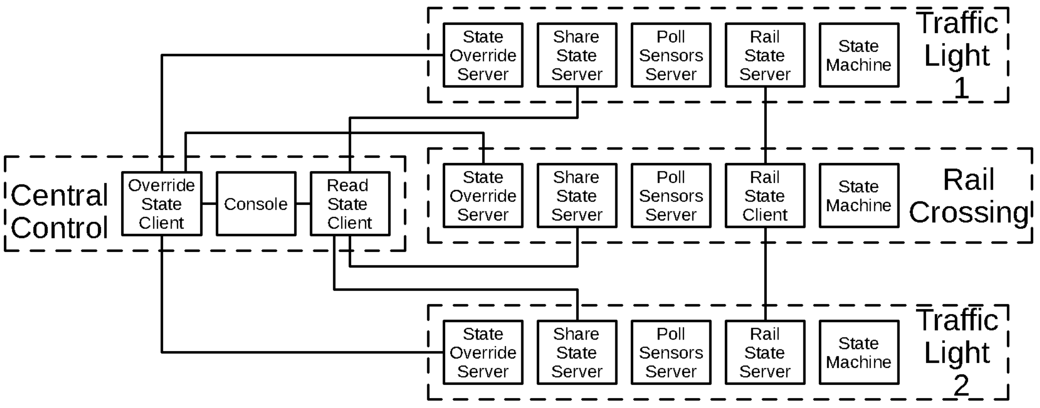


Figure 9: Task architecture diagram. It provides a broad overview of each task in the system, and which tasks communicate with each other.

4.1.1 Traffic Lights

Four conductive paper strips working as proximity sensors simulate car sensors on each traffic light node. Switches on the DE10 nano board are also used to simulate car sensors, however they are only used as a redundancy in case of the failure of the proximity sensors. The software for the proximity sensors is developed using Verilog. The FPGA software outputs of the proximity sensors are polled by the microcontroller component of the DE10 nano. These sensor values are then used by the state machine within the traffic light controller software to make decisions about future states.

Hardware featuring indicative LEDs was also built. This hardware interfaces with the FPGA software. The manner in which the different LEDs are be lit up on the hardware are determined by the current state of the traffic light in question. Each time the traffic light changes state, the FPGA portion of the DE10 nano sendes an update of the correct output values for each LED.

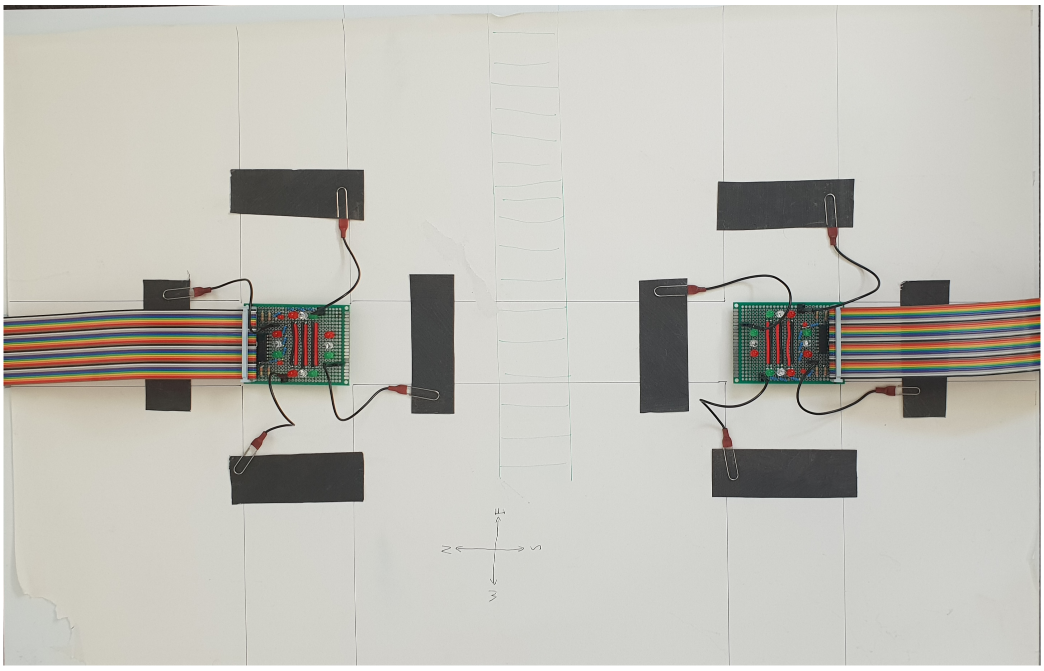


Figure 10: Photo of built hardware.

Each traffic light node runs three servers concurrently. The first server is for communication between the traffic light node and the railway crossing. The server listens for railway crossing state updates. If the railway crossing tries to send a message containing the current state of the railway crossing, this state is updated locally on the traffic light node, and factors in to the current state decision making process of the traffic light node as per the predefined traffic light state table. If this rail state message is received, a reply message of "ok" is sent.

The second server is for override state communication between central control and the traffic light. When a central control client sends a override message to the traffic light override server, this override state is immediately implemented as the current state on the traffic light node. A reply packet of "ok" is also sent.

The third server is for the sharing of the current state of the traffic light node to central control. When the central control data share client sends a data share packet to the traffic light data share server, the traffic light data share server sends a message containing the current state and sensors values of the node.

4.1.2 Rail Crossing

The four rail crossing sensors are emulated using four switches that are available on the DE10 nano board. The output state is also emulated using the on board LEDs. The software for the proximity sensors is developed using Verilog. The FPGA software outputs of the switches are polled by the microcontroller component of the DE10 nano. These switch values are then used by the state machine within the rail crossing controller software to make decisions about future states. Each time the rail crossing changes state, the FPGA portion of the DE10 nano will be sent an update of the correct output values for the on-board LEDs.

The rail crossing node will run two servers and one client concurrently. The client is for communication between the traffic light nodes and the railway crossing. The client sends a message containing the current state of the railway crossing to the traffic light nodes each time there is a state change. This message passing is "one shot". A connection is made between the client and the server and subsequently disconnected every time a message needs to be sent. If the railway crossing tries to send a message containing the current state of the railway crossing, this state is updated locally on the traffic light node, and factors in to the current state decision making process of the traffic light node as per the predefined traffic light state table.

The first server is for override state communication between central control and the rail crossing node. When a central control client sends a override message to the rail crossing server, this override state is immediately implemented as the current state on the rail crossing node. A reply packet of "ok" is also sent.

The second server is for the sharing of the current state of the rail crossing node to central control. When the central control data share client sends a data share packet to the rail crossing data share server, the rail crossing data share server sends a message containing the current state and sensors values of the node.

4.1.3 Central Control

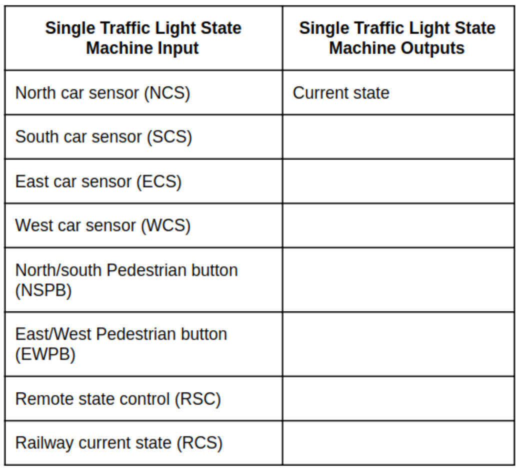
The central control uses a console based program to interact with the traffic and rail crossing nodes. This console has the ability to print the current state and sensor values of each traffic light and rail crossing node. It can also override the current state or pattern mode of the traffic light and rail crossing nodes. Finally, it can disable any applied override on the traffic light and rail crossing nodes.

The central control node runs six separate clients in a one shot manner. Three clients are dedicated to connecting to the override servers of the two traffic light nodes and the rail crossing node, and three clients are dedicated to the connecting to the data share servers of the two traffic light nodes and the rail crossing node. This one shot message passing is activated when a user selects the appropriate option on the console. The client then attaches to a relevant server, passes a message, gets a reply, and detaches from the server. This system was chosen as it proved to be the most reliable way of sending messages. Examples of this may be found in Appendix Figures 25, 26.

4.2 State Overview

4.2.1 Traffic Lights

The following table represents the required inputs and outputs for the traffic light node. These inputs are required in order for the traffic lights to make the correct decision for what to do, enabling them to work autonomously. It includes four car traffic sensor inputs for each side of a intersection, two pedestrian buttons for each direction of pedestrian traffic (ideally this would be four buttons to be perfectly realistic, however due to hardware constraints two button were chosen for the project), a remote state control input for the central control requirements, and the current state of the railway crossing.



Single Traffic Light State Machine Input	Single Traffic Light State Machine Outputs
North car sensor (NCS)	Current state
South car sensor (SCS)	
East car sensor (ECS)	
West car sensor (WCS)	
North/south Pedestrian button (NSPB)	
East/West Pedestrian button (EWPB)	
Remote state control (RSC)	
Railway current state (RCS)	

Figure 11: Traffic light io table

The following diagrams are the traffic light state diagram, and the traffic light state table. It shows the process behind the decision making of the traffic lights, enabling it to operate both autonomously, via remote control.

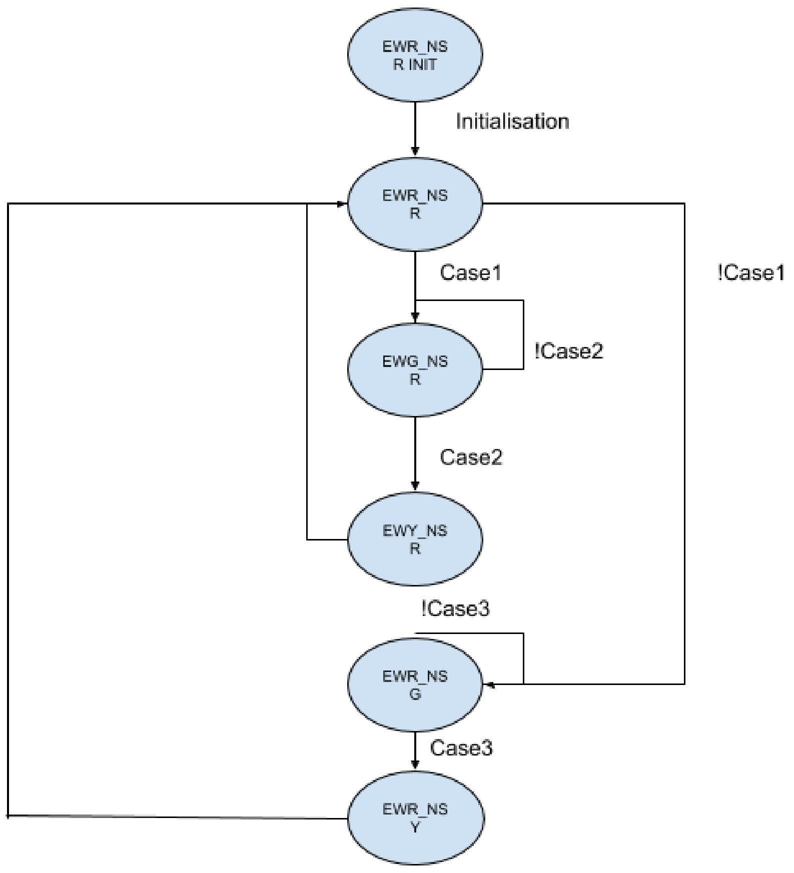
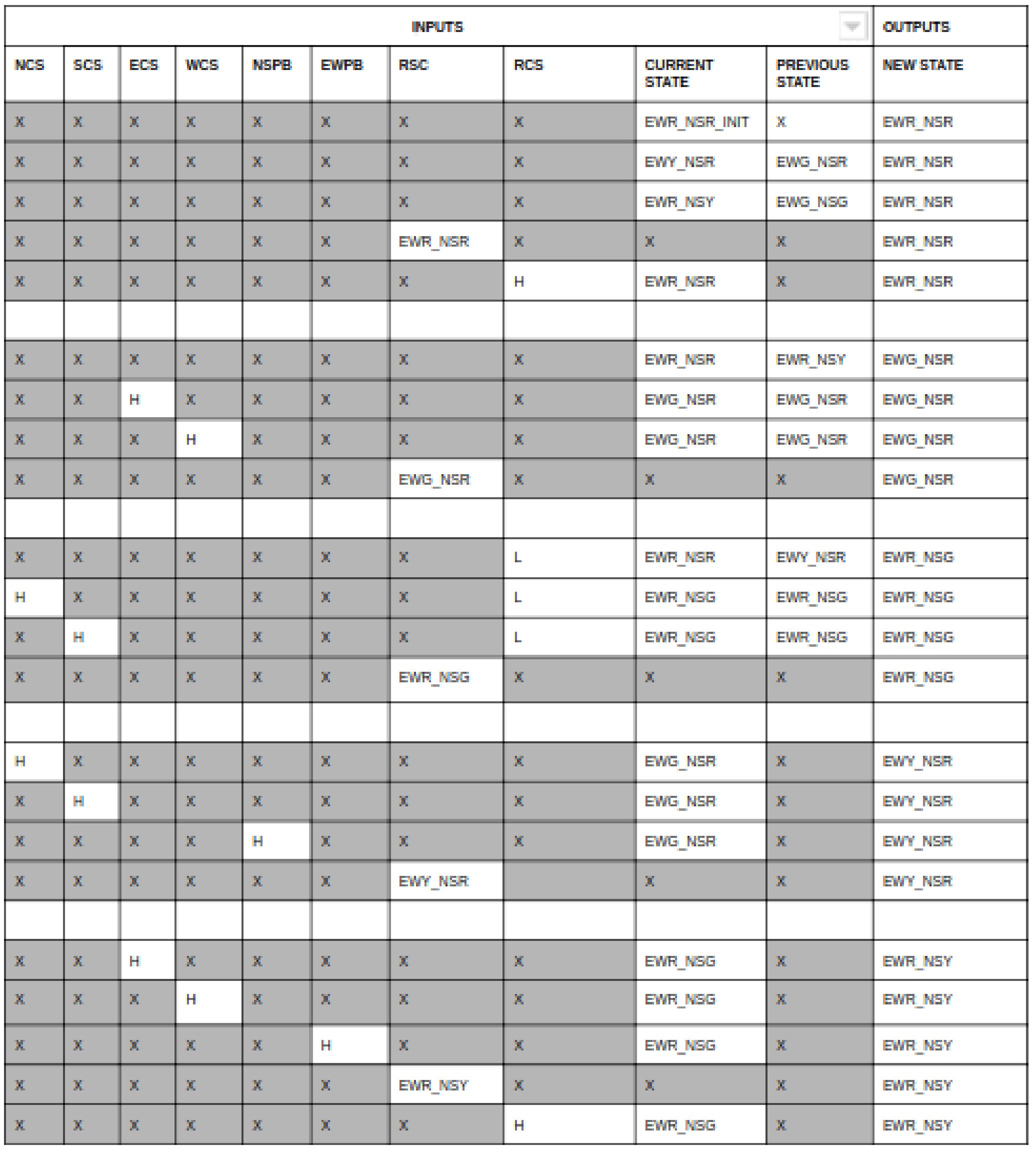


Figure 12: Traffic light state diagram



INPUTS										OUTPUTS
NCS	SCS	ECS	WCS	NSPB	EWPB	RSC	RCS	CURRENT STATE	PREVIOUS STATE	NEW STATE
X	X	X	X	X	X	X	X	EWR_NSR_INIT	X	EWR_NSR
X	X	X	X	X	X	X	X	EWY_NSR	EWG_NSR	EWR_NSR
X	X	X	X	X	X	X	X	EWR_NSY	EWG_NSG	EWR_NSR
X	X	X	X	X	X	EWR_NSR	X	X	X	EWR_NSR
X	X	X	X	X	X	X	H	EWR_NSR	X	EWR_NSR
X	X	X	X	X	X	X	X	EWR_NSR	EWR_NSY	EWG_NSR
X	X	H	X	X	X	X	X	EWG_NSR	EWG_NSR	EWG_NSR
X	X	X	H	X	X	X	X	EWG_NSR	EWG_NSR	EWG_NSR
X	X	X	X	X	X	EWG_NSR	X	X	X	EWG_NSR
X	X	X	X	X	X	X	L	EWR_NSR	EWY_NSR	EWR_NSG
H	X	X	X	X	X	X	L	EWR_NSG	EWR_NSG	EWR_NSG
X	H	X	X	X	X	X	L	EWR_NSG	EWR_NSG	EWR_NSG
X	X	X	X	X	X	EWR_NSG	X	X	X	EWR_NSG
H	X	X	X	X	X	X	X	EWG_NSR	X	EWY_NSR
X	H	X	X	X	X	X	X	EWG_NSR	X	EWY_NSR
X	X	X	X	H	X	X	X	EWG_NSR	X	EWY_NSR
X	X	X	X	X	X	EWY_NSR		X	X	EWY_NSR
X	X	H	X	X	X	X	X	EWR_NSG	X	EWR_NSY
X	X	X	H	X	X	X	X	EWR_NSG	X	EWR_NSY
X	X	X	X	X	H	X	X	EWR_NSG	X	EWR_NSY
X	X	X	X	X	X	EWR_NSY	X	X	X	EWR_NSY
X	X	X	X	X	X	X	H	EWR_NSG	X	EWR_NSY

Figure 13: Traffic light state table

4.2.2 Railway Crossing

The following table represents the required inputs and outputs for the railway crossing node. These inputs are required in order for the railway crossing node to make the correct decision for what to do, enabling it to work autonomously. It includes two train sensor inputs for each side of a crossing, and a remote state control input for the central control requirements.

Single Railway State Machine Input	Single Railway State Machine Output
Train 1 sensor	Current state
Train 2 sensor	
Remote state control	

Figure 14: Rail I/O table

The following diagrams are the railway crossing state diagram, and the railway crossing state table. It shows the process behind decision making process of the railway crossing, enabling it to operate both autonomously, via remote control.

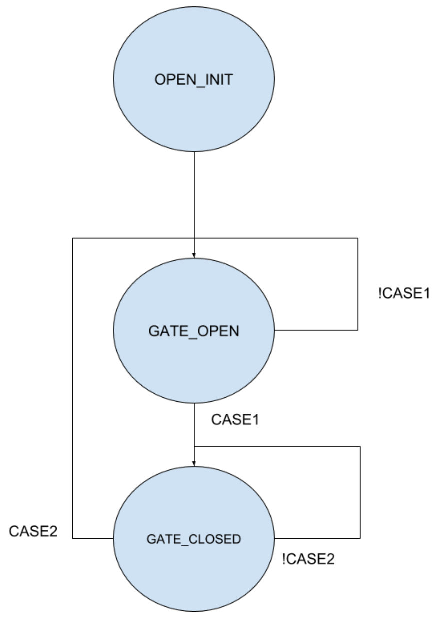
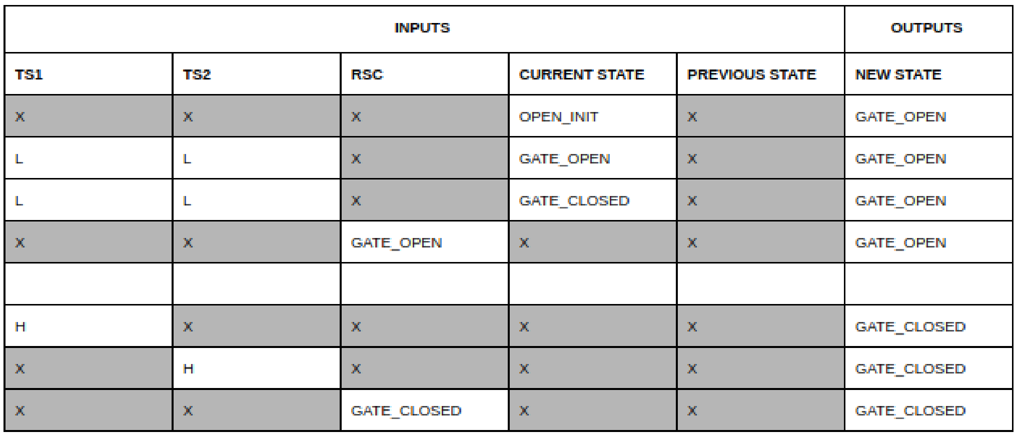


Figure 15: Rail state diagram



INPUTS					OUTPUTS
TS1	TS2	RSC	CURRENT STATE	PREVIOUS STATE	NEW STATE
X	X	X	OPEN_INIT	X	GATE_OPEN
L	L	X	GATE_OPEN	X	GATE_OPEN
L	L	X	GATE_CLOSED	X	GATE_OPEN
X	X	GATE_OPEN	X	X	GATE_OPEN
H	X	X	X	X	GATE_CLOSED
X	H	X	X	X	GATE_CLOSED
X	X	GATE_CLOSED	X	X	GATE_CLOSED

Figure 16: Rail state table

4.3 Messages Passing

For the message passing, QNX native named message passing was used. This QNX message passing tends to be faster and more intuitive than the POSIX alternative. To create a server, the name_attach() function must be used. Messages from clients can then be listening to using the MsgReceive() function. When a message is received, MsgReply() is used to send a reply. On the client side, a client can attach itself to a server by using name_open(), and send a message to the server using MsgSend(). It can then detach itself using name_close();

4.3.1 Traffic Light Messages

Sensor status for the two pedestrian buttons and the four car sensors are sent to central control as the sensor input status changes, as are Changes in traffic light state. The following struct shows the data contained in this message:

```
struct s_traffic_message_data
{
  struct _pulse hdr; // Our real data comes after this header
  int ClientID; // our data (unique id from client)
  unsigned int north_south_green_length;
  unsigned int east_west_green_length;

  enum e_SENSOR_STATE north_car_sensor;
  enum e_SENSOR_STATE south_car_sensor;
  enum e_SENSOR_STATE east_car_sensor;
  enum e_SENSOR_STATE west_car_sensor;

  enum e_SENSOR_STATE north_south_pedestrain_sensor;
  enum e_SENSOR_STATE east_west_pedestrain_sensor;

  enum e_STATE current_state;
  enum e_STATE previous_state;

  enum e_SENSOR_STATE automatic_mode;
};
```

The variables north_south_green_length and east_west_green_length refer to the time the green light is active in a cycle when the traffic lights are in automatic mode. All four traffic sensor values are sent, as well as the two pedestrian sensors. The current traffic state and previous state are included, as well as the mode the traffic light is currently in.

4.3.2 Railway Crossing Messages

Sensor status for the two train sensors are sent to central control as the sensor input status changes. Changes in railway state are also sent to central control as the state changes. The following struct shows the data contained in this message:

```
struct s_rail_message_data
{
  struct _pulse hdr; // Our real data comes after this header
  int ClientID; // our data (unique id from client)
  unsigned int lights_red_length;

  enum e_SENSOR_STATE east_north_train_sensor;
  enum e_SENSOR_STATE east_south_train_sensor;
  enum e_SENSOR_STATE west_north_train_sensor;
  enum e_SENSOR_STATE west_south_train_sensor;
  enum e_SENSOR_STATE train_present_flag;
  enum e_STATE current_state;
  enum e_STATE previous_state;

};
```

All four rail sensor values are sent. The current traffic state and previous state are included, as well as wheather a train presence flag.

4.3.3 Central Control Messages

All sensor statuses from the traffic lights and railway crossing are accepted, as well as the various current states of these nodes. The central control can also send messages to all nodes overriding their error state. The following struct shows the data contained in this message:

```c
struct s_override_data
{
  struct _pulse hdr; // Our real data comes after this header
  int ClientID; // our data (unique id from client)
  unsigned char override_active_flag;
  enum e_STATE override_new_state;
  enum e_SENSOR_STATE automatic_mode;
};
```

The override_new_state variable controls what state a node is forced into. The automatic_mode forces the traffic light nodes into a different mode.

4.4 Traffic Light Timing

Timers are used to implement the time intervals that exist between green, yellow, and red states of traffic lights and rail crossing lights. A timer can be created by using the timer_create() function, and the timer itself can be started by using the startTimer() function. Once a timer is started, the MsgReceive() function can be used to hold a thread/function until the timer has finished. The following data struct is used to hold all timer related variable and data structures. This struct can be passed to various functions that require timers, or are required to start timers.

```c
struct s_traffic_timing_event
{
  pthread_rwlock_t rwlock;
  struct sigevent traffic_timer_event;
  struct itimerspec instant_timer_spec;
  struct itimerspec green_timer_spec;
  struct itimerspec yellow_timer_spec;
  timer_t timer_id;
  int channel_id;
  int receive_id;
  struct s_signal_msg msg;

  struct sigevent poll_timer_event;
  struct itimerspec poll_timer_spec;
  timer_t poll_timer_id;
```

```
    int poll_channel_id;
    int poll_receive_id;
    struct s_signal_msg poll_msg;
};
```

4.5 Node Overview

The following figures show the hardware block diagrams for the traffic light nodes and railway crossing node. They consist of a series of sensors, buttons, switches, lights, and a DE10-Nano board.

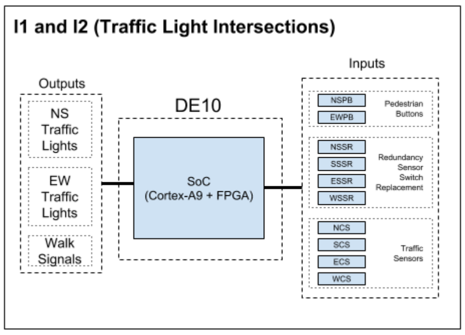


Figure 17: Traffic light block diagram

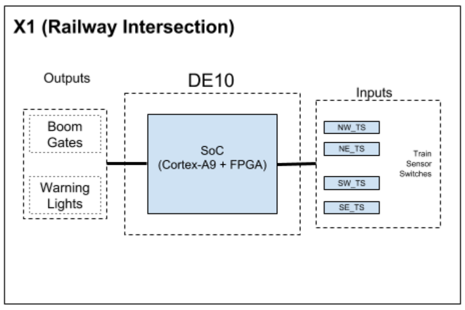


Figure 18: Rail block diagram

4.6 Node I/O

The following figures give a closer look to the inputs and outputs of the FPGA module in the DE10-Nano board for both the traffic light implementation, and the railway crossing implementation.

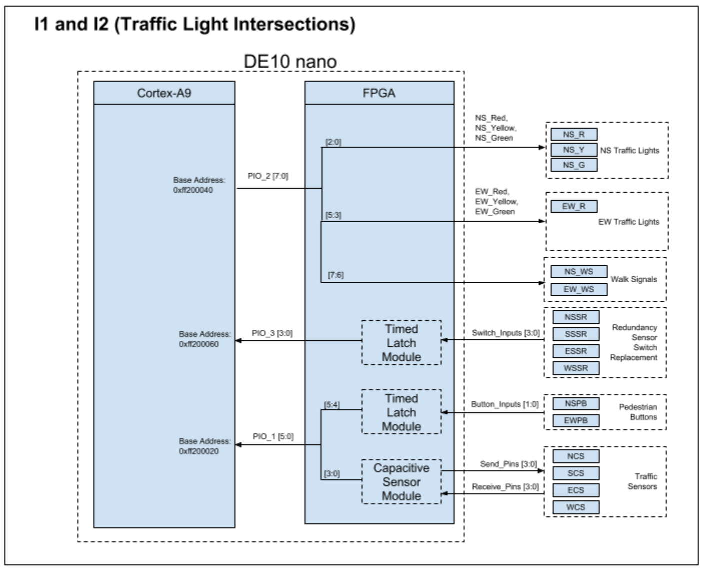
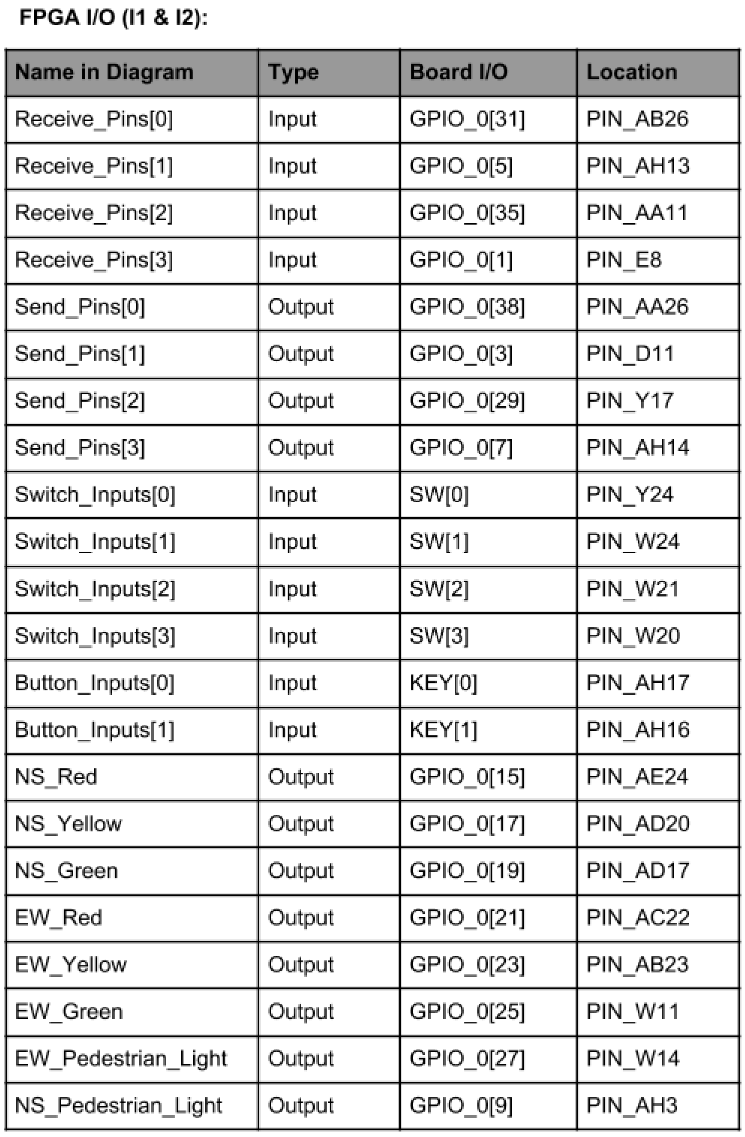


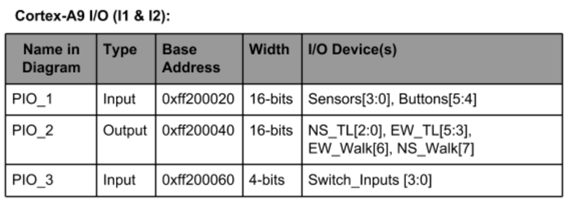
Figure 19: Traffic light I/O diagram



FPGA I/O (I1 & I2):

Name in Diagram	Type	Board I/O	Location
Receive_Pins[0]	Input	GPIO_0[31]	PIN_AB26
Receive_Pins[1]	Input	GPIO_0[5]	PIN_AH13
Receive_Pins[2]	Input	GPIO_0[35]	PIN_AA11
Receive_Pins[3]	Input	GPIO_0[1]	PIN_E8
Send_Pins[0]	Output	GPIO_0[38]	PIN_AA26
Send_Pins[1]	Output	GPIO_0[3]	PIN_D11
Send_Pins[2]	Output	GPIO_0[29]	PIN_Y17
Send_Pins[3]	Output	GPIO_0[7]	PIN_AH14
Switch_Inputs[0]	Input	SW[0]	PIN_Y24
Switch_Inputs[1]	Input	SW[1]	PIN_W24
Switch_Inputs[2]	Input	SW[2]	PIN_W21
Switch_Inputs[3]	Input	SW[3]	PIN_W20
Button_Inputs[0]	Input	KEY[0]	PIN_AH17
Button_Inputs[1]	Input	KEY[1]	PIN_AH16
NS_Red	Output	GPIO_0[15]	PIN_AE24
NS_Yellow	Output	GPIO_0[17]	PIN_AD20
NS_Green	Output	GPIO_0[19]	PIN_AD17
EW_Red	Output	GPIO_0[21]	PIN_AC22
EW_Yellow	Output	GPIO_0[23]	PIN_AB23
EW_Green	Output	GPIO_0[25]	PIN_W11
EW_Pedestrian_Light	Output	GPIO_0[27]	PIN_W14
NS_Pedestrian_Light	Output	GPIO_0[9]	PIN_AH3

Figure 20: Traffic light FPGA I/O table



Cortex-A9 I/O (I1 & I2):

Name in Diagram	Type	Base Address	Width	I/O Device(s)
PIO_1	Input	0xff200020	16-bits	Sensors[3:0], Buttons[5:4]
PIO_2	Output	0xff200040	16-bits	NS_TL[2:0], EW_TL[5:3], EW_Walk[6], NS_Walk[7]
PIO_3	Input	0xff200060	4-bits	Switch_Inputs [3:0]

Figure 21: Traffic light Cortex-A9 I/O table

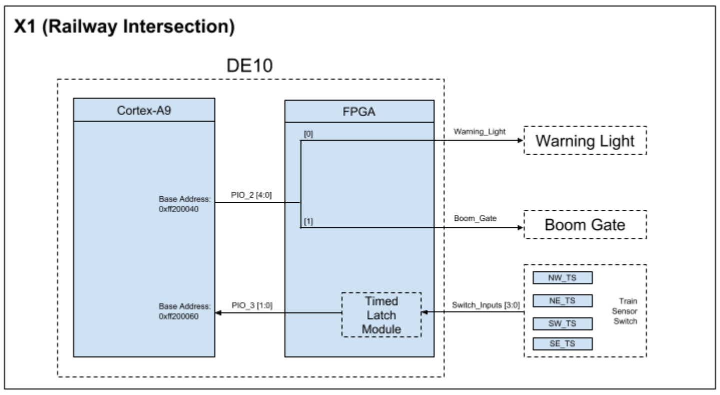
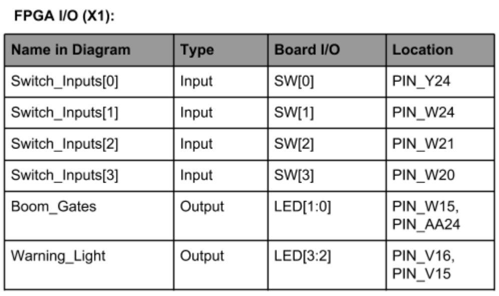
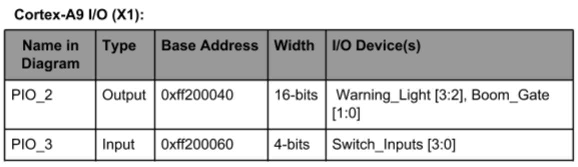


Figure 22: Rail I/O diagram


FPGA I/O (X1):

Name in Diagram	Type	Board I/O	Location
Switch_Inputs[0]	Input	SW[0]	PIN_Y24
Switch_Inputs[1]	Input	SW[1]	PIN_W24
Switch_Inputs[2]	Input	SW[2]	PIN_W21
Switch_Inputs[3]	Input	SW[3]	PIN_W20
Boom_Gates	Output	LED[1:0]	PIN_W15, PIN_AA24
Warning_Light	Output	LED[3:2]	PIN_V16, PIN_V15

Figure 23: Rail FPGA I/O Table


Cortex-A9 I/O (X1):

Name in Diagram	Type	Base Address	Width	I/O Device(s)
PIO_2	Output	0xff200040	16-bits	Warning_Light [3:2], Boom_Gate [1:0]
PIO_3	Input	0xff200060	4-bits	Switch_Inputs [3:0]

Figure 24: Rail Cortex-A9 I/O Table

5 Safety and Fault Tolerance

5.1 Traffic Error State

The traffic light state machine has an error state (TRAFFIC_ERROR), the traffic error state may be entered manually by a central control override state message, or if there is an unexpected error preventing the traffic light from entering a valid state. In our design the lights turn off while the traffic light is in an error state, although in hindsight the traffic lights should all turn yellow, as is commonly used in Australia to indicate that a commuter should proceed with caution. To exit the error state central control can send an override command to force the traffic light back into a valid state.

5.2 Railway Error State

The railway crossing state machine has an error state (RAIL_ERROR), the railway error state may be entered manually by a central control override command, or if there is an unexpected error preventing the railway from entering a valid state. While in an error state the boom gates close and the warning lights turn on. To exit the error state central control can send an override command to force the railway crossing back into a valid state.

5.3 Central Control Node Disconnection

The rail states are sent to the traffic light nodes directly, therefore in the event that the central control node loses power, is manually shut down or disconnects from the network for any other reason the rest of the system will continue to run as normal.

5.4 Traffic Light Node Disconnection

If the traffic light node disconnects from the network, it can continue to run in its current mode (automatic or sensor mode). The traffic light nodes have no control over other nodes, therefore if a traffic light node shuts down entirely, the other nodes will continue to run as usual.

The central control opens a new connection to a traffic light server whenever it is attempting to send a message to override or get the current status of a traffic light node. After a message is sent and a reply is received the connection to the server is closed. If the MsgSend function returns -1 a message is displayed notifying the central control user that no reply was received (i.e. there is a problem with the traffic light node), then the connection is closed.

The railway node sends its state to the traffic light nodes whenever it changes state. Similarly to the central control node, the railway node opens a connection whenever it needs to message the traffic light nodes and closes the connection after a reply (or an error) is received.

Using this method of opening and closing connections each time a message is sent, the central control and railway nodes will be able to reconnect to the traffic light node whenever it comes back online.

5.5 Railway Crossing Node Disconnection

If the railway crossing node disconnects from the network (or is shut down), it will no longer be able to send it's state to the traffic light nodes. In this situation the traffic light node does not know the railway state has been disconnected, therefore the responsibility falls on the central control to override the traffic light node into a safe state if necessary.In hindsight another solution to this issue would be to have the railway node message the traffic light nodes at set intervals, if the traffic light node did not receive a message it would know that there is some issue with the railway node and could enter a safe state without the need for central control intervention.

The central control opens a new connection to a railway node server whenever it is attempting to send a message to override or get the current status of a traffic light node. After a message is sent and a reply is received the connection to the server is closed. If the MsgSend function returns -1 a message is displayed notifying the central control user that no reply was received (i.e. there is a problem with the railway node), then the connection is closed.

Using this method of opening and closing connections each time a message is sent, the central control and railway nodes will be able to reconnect to the traffic light node whenever it comes back online.

6 Solution Process

6.1 Initial Design

The initial design process included a series of group meetings and individual work. In the first group meeting we discussed communication methods that would be used throughout the project. We decided that Google Hangouts would be used for online messaging to discuss the project and to arrange meetings. We decided that we would meet at least once a week on Mondays at RMIT building 10 level 9 to work collaboratively with other meetings to be arranged as required. We also set up a Google drive folder for sharing design documents, and a git repository for managing code.

During the first design meetings we had a focus on defining our goals for the project, including our interpretation of the problem, use cases, behaviour specifications and hardware implementation. Activities in these meeting included discussion and sketching of diagrams. Between meetings the group members worked on producing more detailed design documentation, including state diagrams, state tables, block diagrams, overview of messages sent between tasks and use case diagrams and descriptions. In subsequent meetings we went through the design document to work through any anticipated conflicts in the design.

6.2 Shared Data Structs

To ensure that all code was written coherently, a header file was created containing definitions of attach points to be used for message passing and struct definitions to be used to pass data between threads and processes.

Structs defined in this header file include message data structs (s_traffic_message_data and s_rail_message) containing enum variables for sensors and states that are sent between processes, override data structs (s_override_data) which contains variables that can be sent from central control to either a traffic light or railway node to override it's state, or to change a traffic light between automatic and sensor modes.

6.3 State Machine

State machines for the traffic light and railway nodes were developed in isolation and their performance was verified using printf() statements to indicate changes in state. The structs defined in the shared_data_structures.h file were used to ensure that the state machine could operate with the rest of the system.

6.4 Hardware Implementation

A capacitive proximity sensor module was developed in Quartus using Verilog HDL and tested on the DE10 nano board. The board containing the proximity sensor hardware and the traffic light LEDs was soldered to a perf-board with a headers that can be connected to the DE10-nano via a 40 pin ribbon cable. Once the hardware was known to be working the capacitive sensor modules were included in the Quartus project that was provided as a part of the BSP on canvas and the outputs were wired in Verilog to the GPIO ports that have been accessible in QNX.

Code for initialising the DE10-nano GPIO in QNX was taken from the example code that was provided on canvas. A sensorPoll thread was then written to poll the inputs of the GPIO and the switches, then using logical operators determine which sensors/switches have been triggered and set the data in the s_traffic_message_data struct accordingly, read write locks were used to prevent concurrent changing of the sensor variables. An updateLights function was written to set the LEDs on the board according to the current state of the traffic light state machine.

The state machine thread (trafficLightControl) was then tested with the sensorPoll thread and the updateLights function to ensure they all operated together. There were a few errors related to pin assignments in Quartus or minor errors in the state machine switch case statement, but they were all solved without too much trouble.

6.5 Message passing

Due to intermittent access to hardware (labs and exams in RMIT building 10 level 9), the code for accessing the DE10-nano GPIO was removed from the code and we began development and testing of message passing using virtual machines.

We began development of this part of the project with the message passing between the railway node and the traffic light nodes. The railway node needs to notify the traffic light nodes of it's current state, we considered two ways of achieving this. The first method would be to have a server running on the railway node with a client on the traffic light node, the traffic light would poll the railway node to get it's current state. We quickly realised that this would result in a lot of unnecessary message passing, so we instead decided to have a server on the traffic light node and to have the railway node send a message each time it's state changes.

Initially we had the railway node attempt to open a connection to the traffic lights server every few seconds until it was successful. We soon realised if we use this method and a traffic light node is reset at any point, the railway node will be unable to reconnect to the server, so we instead decided to have the railway node disconnect each time after sending a message, allowing the traffic light/railway nodes to continue to operate as usual even if one of them is reset at some point.

We then moved on to developing the message passing structure for the central control, initially we thought to have the central control node running a server, while the traffic light and railway nodes would connect and message their state each time it changed. Although after some discussion, we decided that a solution such as this would be fine for a system with only 2 traffic light nodes and 1 railway node although if the number of traffic and railway nodes was increased, the central control would be constantly inundated with state change messages from the various nodes. So we decided that the servers should be running on each traffic light and railway node, and the central control node can send a message requesting information from any specific node.

There were a few issue that arose with the message passing between the central control and the traffic light and railway nodes such as data not being received as expected, this was simply an error with our code (sending a pointer rather than the data), these issues were identified and fixed without too much trouble. Additionally we had an error where the traffic light state machine would enter a traffic error state after being released from central control override, which turned out to be because the traffic light was unsure weather it should enter a EWR_NSG or a EWG_NSR state after returning from a EWR_NSR override. This issue was resolved by having the traffic light node default to EWG_NSR after being released from a EWR_NSR override.

7 Conclusion

We believe this design, implemented, will be capable of directing traffic and pedestrians in the set of traffic intersections described, that the system is safe, efficient and robust enough to run autonomously while allowing for monitoring or intervention in the event of an error.

Due to the use of distributed processing and the fact that each node can run autonomously under normal circumstances, the system could be scaled well to include many more traffic intersections within the network.

8 Reflection

This project in general has been a great learning experience into not only the benefits of real-time operating systems in tackling real world issues, but also how to structure the foundation of software design to allow for co-operative team collaboration.

Firstly, QNX and RTOSs in general were a new concept to many of us, yet we learned the reasoning behind and were able to practically apply concepts such as multi-threaded processes and concurrency control of data shared between multiple threads. Furthermore, throughout our previous studies, inter-process communication was never emphasised, however in this project, we effectively used the native QNX networking system, QNET to communicate certain information such as node state between different types of processes.

Not all of the project participants were initially well versed in the C programming language. Thanks to the semester's laboratory classes as well as valuable references provided,[2] a good understanding of the programming language was able to be formed including often poorly explained features such as pointers, structures and enumerated values. Furthermore, offloading certain tasks to an FPGA and communicating with an ARM SOC hardcore was able to be implemented successfully, which is not a simple task.

Soft skills were a large and essential portion of the entire project. Each team member had to communicate with each other to plan which parts of the project were to be implemented by who, and this had to be done effectively to meet deadlines. Utilising Git was another pleasing aspect of the project. Although there were issues at times in merging branches together, it was a good experience in how a professional software project with multiple contributors is handled.

Unfortunately, not everything went perfect throughout the project. Time was running out reaching near the deadline, and coding elegance had to be forgone a little to ensure a working solution by the presentation. Retrospectively, it would have been beneficial to converse with our professor, Dr. Ippolito as a client for an engineering project to ensure all features were being implemented as wanted, such as that of a yellow flashing error state.

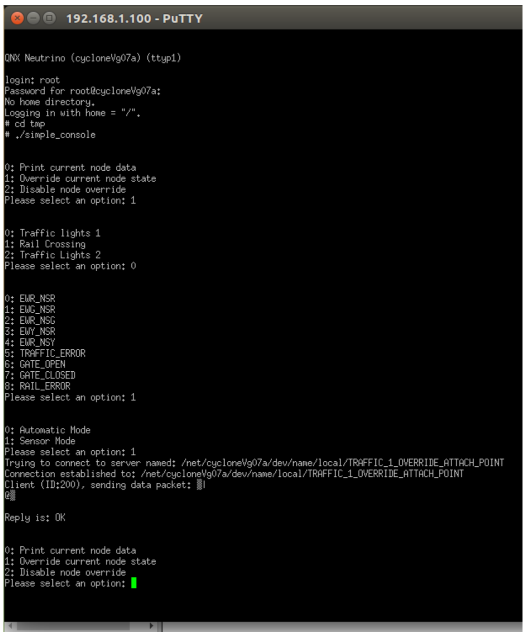
9 Acknowledgements

There were a only couple of shortcomings in regards to the project in its entirety, and therefore it is essential to reiterate that the designed system turned out quite well, and that all participants are proud of the work, culminating into our presentation. A special thanks has to be extended towards Dr. Ippolito for managing the course.

Some sections of this report have been repeated from the our initial report, written by the same contributors. sections include the problem statement, as well as parts of the design discussion.

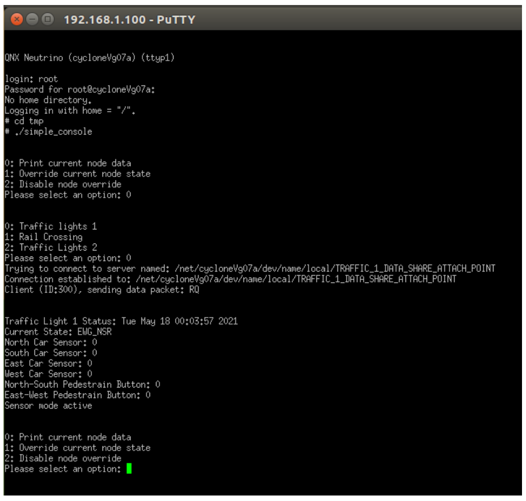
Sections of the source code for the system were derivates of examples taken out of the official QNX 7.0.0 documentation, and provided by Dr. Ippolito.

10 Appendix



```
QNX Neutrino (cycloneVg07a) (ttyp1)

login: root
Password for root@cycloneVg07a:
No home directory.
Logging in with home = "/".
# cd tmp
# ./simple_console


0: Print current node data
1: Override current node state
2: Disable node override
Please select an option: 1


0: Traffic lights 1
1: Rail Crossing
2: Traffic Lights 2
Please select an option: 0


0: EWR_NSR
1: EWG_NSR
2: EWR_NSG
3: EWY_NSR
4: EWR_NSY
5: TRAFFIC_ERROR
6: GATE_OPEN
7: GATE_CLOSED
8: RAIL_ERROR
Please select an option: 1


0: Automatic Mode
1: Sensor Mode
Please select an option: 1
Trying to connect to server named: /net/cycloneVg07a/dev/name/local/TRAFFIC_1_OVERRIDE_ATTACH_POINT
Connection established to: /net/cycloneVg07a/dev/name/local/TRAFFIC_1_OVERRIDE_ATTACH_POINT
Client (ID:200), sending data packet: ▒I
@▒

Reply is: OK


0: Print current node data
1: Override current node state
2: Disable node override
Please select an option: █
```

Figure 25: Console override menu.



```
QNX Neutrino (cycloneVg07a) (ttyp1)

login: root
Password for root@cycloneVg07a:
No home directory.
Logging in with home = "/".
# cd tmp
# ./simple_console


0: Print current node data
1: Override current node state
2: Disable node override
Please select an option: 0


0: Traffic lights 1
1: Rail Crossing
2: Traffic Lights 2
Please select an option: 0
Trying to connect to server named: /net/cycloneVg07a/dev/name/local/TRAFFIC_1_DATA_SHARE_ATTACH_POINT
Connection established to: /net/cycloneVg07a/dev/name/local/TRAFFIC_1_DATA_SHARE_ATTACH_POINT
Client (ID:300), sending data packet: RQ


Traffic Light 1 Status: Tue May 18 00:03:57 2021
Current State: EWG_NSR
North Car Sensor: 0
South Car Sensor: 0
East Car Sensor: 0
West Car Sensor: 0
North-South Pedestrain Button: 0
East-West Pedestrain Button: 0
Sensor mode active


0: Print current node data
1: Override current node state
2: Disable node override
Please select an option: █
```

Figure 26: Console print state menu.

References

[1] “Road safety statistics,” Bureau of Infrastructure, Transport and Regional Economics, October 2018. [Online]. Available: https://bitre.gov.au/statistics/safety/

[2] B. Hall, *Beej’s Guide to C Programming*, May 2007.