An Eco-Traffic Signal System Based on Connected Vehicle Technology

A Thesis

Presented in Partial Fulfillment of the Requirements for the

Degree of Master of Science

with a

Major in Computer Science

in the

College of Graduate Studies at

University of Idaho

by

Anup Chitrakar

Major Professor: Axel Krings, Ph.D.

Committee Members: Robert Rinker, Ph.D.; Ahmed-Abdel Rahim, Ph.D.

Department Administrator: Frederick Sheldon, Ph.D.

April 2016

Authorization to Submit Thesis

This thesis of Anup Chitrakar, submitted for the degree of Master of Science with a major in Computer Science and titled **"An Eco-Traffic Signal System Based on Connected Vehicle Technology,"** has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor:	______________________ Axel Krings, Ph.D.	Date	__________
Committee members:	______________________ Robert Rinker, Ph.D.	Date	__________
	______________________ Ahmed-Abdel Rahim, Ph.D.	Date	__________
Department Administrator:	______________________ Frederick Sheldon, Ph.D.	Date	__________

Abstract

The Intelligent Transportation System uses Dedicated Short Range Communications (DSRC) for vehicle-to-vehicle and vehicle-to-infrastructure communication. This technology is used for applications that intend to increase safety and to improve traffic management and operation. For the latter it promises applications with advanced features in order to reduce fuel consumption.

This research presents the design and implementation of a system architecture, diverse algorithms, and communication methods of an Eco-Traffic Signal System. The application uses vehicle-to-infrastructure communications to control traffic light timing with the goal of avoiding unnecessary stops of heavy vehicles, which in turn results in energy savings. The architecture takes advantage of Basic Safety Messages in connected vehicle technology and executes an application inside of the Road Side Unit employed in future traffic intersections. This unit facilitates the necessary algorithms and communication support to instruct the traffic controller to manage signal timing. A proof of concept of the Eco-Traffic Signal System was implemented and its functionality was verified in field tests using commercial DSRC equipment.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Prof. Dr. Axel Krings for the continuous support on my MS study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all time during the research and writing this thesis. I could not have imagined having a better advisor and mentor for my MS study. I would also like to thank my committee members, Dr. Ahmed-Abdel Rahim and Dr. Robert Rinker for their valuable input in my thesis. I am grateful to all the staffs and faculty from the Computer Science Department of the University of Idaho for providing an enabling environment for me to work. A special thanks to all my professors and teachers who have helped to impart the knowledge in my academic career.

This project would not have been possible without funding from the National Institute For Advanced Transportation Technology (NIATT) at the University of Idaho. Furthermore, I would like to acknowledge the Idaho Global Entrepreneurial Mission (IGEM) for their support, which was used to purchase the Arada Systems DSRC equipment used in the implementation and field experiments.

Last but not least, I would like to thank my friends and family who have been supportive of my endeavors.

Dedication

This work is dedicated to my family, who have always loved me unconditionally and whose good examples have taught me to work hard for the things that I aspire to achieve.

My father, Arun Kumar Chitrakar.

My mother, Ram Devi Chitrakar.

My brother, Anish Chitrakar.

Table of Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

AASHTO	American Association of State Highway and Transportation Officials
ABS	Anti-Lock Brake System
AC	Access Category
AIFS	Arbitration Interframe Space
ASC	Actuated Signal Controller
ASN	Abstract Syntax Notation
ASN.1	Abstract Syntax Notation One
BER	Basic Encoding Rules
BSM	Basic Safety Message
C2F	Center to Field
CCH	Control Channel
CCTV	Closed Circuit Television
CDT	C/C++ Development Tooling
DER	Distinguished Encoding Rules
DSRC	Dedicated Short Range Communications
EDCA	Enhanced Distributed Channel Access
ETSS	Eco-Traffic Signal System
FTP	File Transfer Protocol
GPS	Global Positioning System
GSM	Global System for Mobile Communication
HV	Host Vehicle
I2V	Infrastructure-to-Vehicle
IDE	Integrated Development Environment
IP	Internet Protocol
ISO	International Organizations of Standards
ITE	Institute of Transportation Engineers
ITS	Intelligent Transportation Systems

LLC	Logical Link Control
MAC	Medium Access Control
MIB	Management Information Base
MIPS	Microprocessor without Interlocked Pipeline Stages
NEMA	National Electrical Manufacturers Association
NTCIP	National Transportation Communication for ITS Protocol
OBU	On Board Unit
OFDM	Orthogonal Frequency-Division Multiplexing
OID	Object Identifier
OSI	Open Systems Interconnection
PMPP	Point-to-Multi-Point Protocol
PoE	Power over Ethernet
PPP	Point-to-Point Protocol
RSU	Road Side Unit
RV	Remote Vehicle
SCH	Service Channel
SNMP	Simple Network Management Protocol
STMP	Simple Transportation Management Protocol
TCP	Transmission Control Protocol
TFTP	Trivial File Transfer Protocol
UDP	User Datagram Protocol
USDOT	United States Department of Transportation
V2I	Vehicle-to-Infrastructure
V2V	Vehicle-to-Vehicle
V2X	Vehicle-to-X
VANET	Vehicular Ad Hoc Network
WAVE	Wireless Access in Vehicular Environments
WSM	WAVE Short Message
WSMP	WAVE Short Message Protocol

Chapter 1

Introduction

The United States Department of Transportation (USDOT) uses the term Intelligent Transportation Systems (ITS) to address improvements in transportation safety and mobility through the integration of advanced communications technologies into the transportation infrastructure and vehicles [1]. The ITS encompasses a broad range of diverse communications and electronics technologies. At the core of ITS are wireless and wired communications, e.g, between vehicles and/or the fixed infrastructure which is shown in Figure 1.1. In the figure wireless communication between vehicles is indicated by green circles. Communication between vehicles and the fixed infrastructure is shown by brown lines, converging at the fixed device.

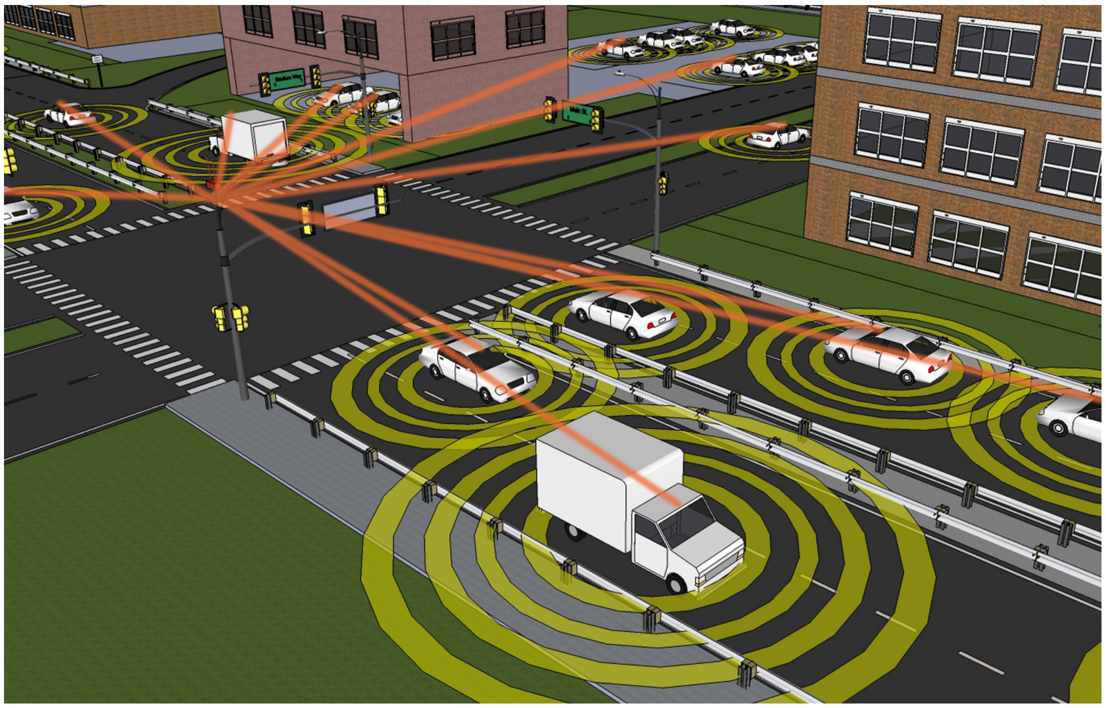


Figure 1.1: Intelligent Transportation System[1]

The USDOT's ITS program focuses on intelligent vehicles and infrastructures. The ITS focuses on improving transportation safety and mobility, with an additional focus on minimizing the energy consumption of vehicles [6].

[1]Picture designed by Manisha Shilpakar, a Graduate Student in the College of Art and Architecture at the University of Idaho.

Furthermore, productivity is enhanced by using various advanced technologies such as computer vision and character recognition for a license plate recognition [2]. The ITS includes a wide suite of technologies and applications which have been widely impacting people's life for the last few decades. Some benefits that ITS provides include the following:

A) **Increased safety**: The ITS attempts to improve the safety of drivers and pedestrians. According to [4], on the world's roadway, annually, there are 1.2 million fatalities. In 2014, there were 32,675 people killed in motor vehicle crashes on U.S. roadways. The accidents caused approximately 2.3 million injuries. ITS began addressing vehicle safety by introducing mandatory installation of seat belts and airbags in the late 90's [3]. Those technologies assumed that there would be crashes. However, now ITS also implies rethinking safety in terms of crash avoidance.

B) **Improved performance by reducing congestion**: Reducing traffic congestion is one of the principal benefits of ITS. Americans spend a total of 4.2 billion hours per year commuting, wasting over 2.8 billion gallons of fuel [3]. There are various ITS applications deployed that enhance the operational performance of a transportation network. Actuated traffic signal lights have contributed to improving traffic flow significantly, reducing stops, cutting gas consumption, and reducing travel time.

C) **Enhanced mobility**: By decreasing congestion and maximizing operational efficiency of the transportation system, ITS has managed to enhance driver mobility. It has also made it easier for the drivers by providing real-time traveler information and enhanced route selection.

D) **Environmental benefits**: The ITS is delivering environmental benefits by reducing congestion, by enabling traffic to flow more smoothly, and by coaching drivers how to drive most efficiently. For example, vehicles traveling at 60 kmph (37 mph) emit 40 percent less CO_2 than vehicles traveling at 20 kmph (12 mph) [4]. Eco-driving enabled vehicles provide feedback to the motorist on how to operate vehicles at the most fuel-efficient speeds across all driving situations.

E) **Improved productivity and economic growth**: By improving the performance of the transportation system, ITS tries to ensure that people and products reach their appointed destinations as quickly and efficiently as possible. This enhances the productivity of workers and businesses, boosting economic competitiveness.

There are many other benefits of ITS in addition to those listed above. Many applications have been proposed, from surveillance of the roadways, rapid evacuation during a natural disaster or threat, to finding open parking spaces.

1.1 Intelligent Transportation Technologies

The ITS uses diverse technologies such as car navigation, traffic signal control systems, automatic number plate recognition, Closed Circuit Television (CCTV) systems, weather-responsive systems. Some key technologies are shown in Figure 1.2, and are described below:

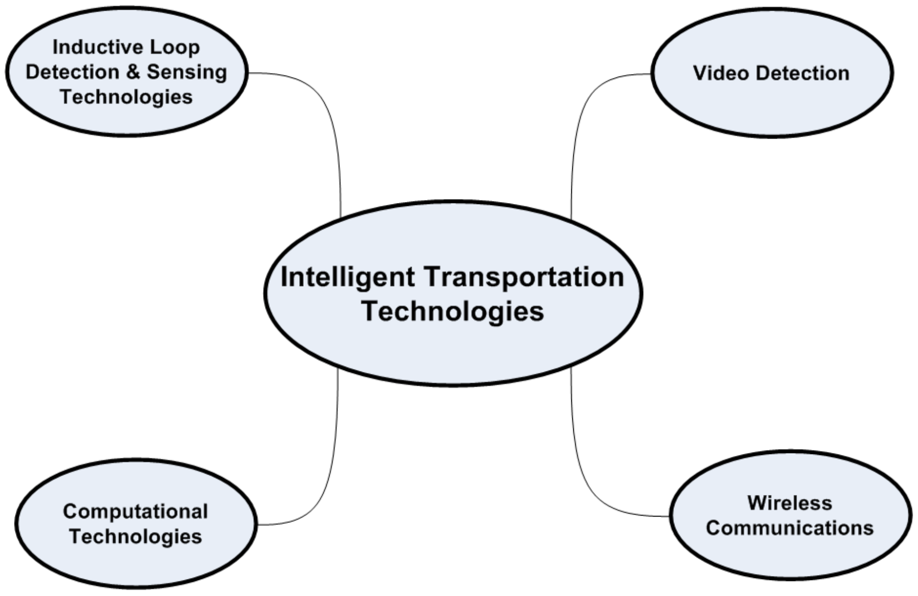

Inductive Loop Detection & Sensing Technologies
Video Detection
Intelligent Transportation Technologies
Computational Technologies
Wireless Communications

Figure 1.2: Subset of Technologies used by ITS

A) **Inductive Loop Detection and Sensing Technologies**: Inductive loop detection mechanisms are often employed in road surveillance. These mechanisms are widely used especially at intersections with actuated traffic signals. An inductive loop detection system consists of an inductive loop and a detector, which is typically installed in a signal cabinet. This system links the signal controller to the inductive loop [5]. When a vehicle enters or crosses the loop, it induces a magnetic field and the detector module will output a detection signal. The actuated signal controller relies on the detector output to decide whether a green signal should be extended for the vehicle passing over an inductive loop.

Installation of such detectors can be intrusive to traffic, as traffic must be interrupted for installation and maintenance. Although an inductive loop detector is able to monitor traffic on a regular basis under all weather and lighting conditions, such detectors can fail under extreme weather conditions, especially snow, ice [5] and often fail to detect light vehicles such as motorcycles or bicycles.

B) **Video Detection**: Unlike inductive loop detectors, video vehicle detection is a non-intrusive method of traffic detection. Usually cameras are mounted on poles or structures either above or adjacent to the roadways. Video captured from the camera is fed into processors for analyzing the changing characteristics of the video image as the vehicle passes the intersection. Such systems need an initial configuration, which trains the device based on the distance between lane lines and the height of the camera above the roadway [5].

C) **Computational Technologies**: In the early 2000s, a typical vehicle would had between 20 and 100 networked micro-controller/programmable logic controller modules with non-real-time operating systems [5]. The current trend focuses on fewer, but more powerful and costly microprocessor modules with hardware management and real-time operating systems. This allows these new embedded system platforms to run computationally complex tasks for more sophisticated software applications.

D) **Wireless Communications**: Various forms of wireless communication technologies have been proposed for ITS. Short-range communication of up to 350 meters can be accomplished using IEEE 802.11 protocols. Dedicated Short Range Communications (DSRC) is being promoted by USDOT and ITS of America. Long-range communications using WiMAX (802.16), Global System for Mobile Communication (GSM), or 3G require extensive and very expensive infrastructure deployment.

1.2 Connected Vehicles

Vehicular Ad Hoc Network (VANET)s are becoming more important to enhance road traffic safety. These networks are the core of applications aiming at increasing passenger safety by exchanging safety messages between vehicles wirelessly. Such wireless communications "increase the line-of-sight of the driver", making vehicles aware of their environment [8]. In VANET vehicles are able to transmit speed, location and vehicle status data to other vehicles and the roadway infrastructure in the form of a Basic Safety Message (BSM). In an Ad hoc network, collection of nodes dynamically forms a network without existing infrastructure or centralized administration [7]. Figure 1.3 shows in instance of a VANET consisting of a set of vehicles equipped with an On Board Unit (OBU) and a stationary unit called Road Side Unit (RSU). Omnidirectional communications, e.g., linear dipole, of OBUs and the RSU are indicated by concentric circles. Communication links between OBUs and/or RSUs are established on the fly via ad-hoc connections when they are in the communication range. Connected vehicles refer to the wireless connectivity enabled vehicles that communicate with external environments. For this reason, two architectures have been proposed - a Vehicle-to-Vehicle (V2V) architecture and a Vehicle-to-Infrastructure (V2I) architecture. Safety systems such as seat belt and air bags help passengers survive during a crash, whereas V2V and V2I communications are used to prevent the accident.

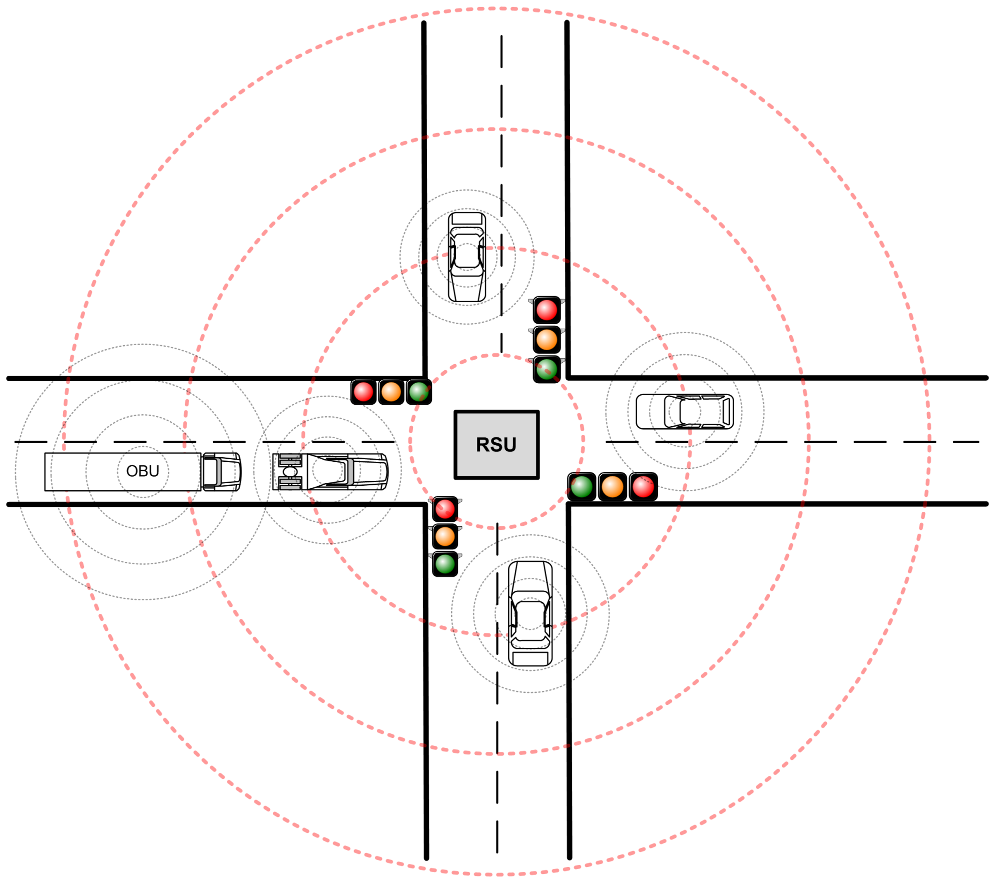


Figure 1.3: Connected Vehicles at an Intersection (adapted from [5])

a) **V2V Communication**: The idea behind V2V communication is that cars will be able to know each other's exact position as they are driving. This can be used to help avoid accidents by informing cars about each other's current location and status, e.g., in a forward collision warning application. In V2V communication, a vehicle is capable of sharing its information such as position, direction and speed using wirelessly with other nearby vehicles on the road.

b) **V2I Communication**: Vehicles can also communicate with the fixed infrastructure, e.g., traffic controllers and signals, via an RSU using V2I communications. It is envisioned that roads will be equipped with RSUs, which allow establishing communication to the infrastructure. Whereas Infrastructure-to-Vehicle (I2V) implies communications between the infrastructure and vehicles, V2I and I2V can be used interchangeably. Collectively, the V2V and the V2I are known as Vehicle-to-X (V2X) communication.

1.3 Motivation

In general, drivers waste fuel and time by stopping vehicles at the signalized intersection. In transportation systems, VANETs can be used to derive various approaches for reducing fuel consumption. This research is driven by an approach for fuel conservation using DSRC technology. The core of this project is to use communicate real-time data from the vehicle to the traffic controller, which then can make decisions resulting in reduce unnecessary stops and delay time at the signalized intersection. Deployment of infrastructure sensors to monitor traffic conditions in current ITS is often expensive. Furthermore, the sensors have limited capabilities to provide accurate traffic information [9]. According to the AERIS simulation [10], such applications attempt to reduce a vehicle's fuel consumption by 1 to 10 percent along the intersection. The data from connected vehicle applications involving vehicle location and speed, can reduce the number of stops by an average of 18%.

1.4 Contributions

This research project[2] focuses on the development of a prototype for an eco-traffic signal system. It provides a proof-of-concept that it is feasible to use connected vehicle technology to control the traffic infrastructure in a adaptive fashion. Specifically, it is demonstrated that it is possible

[2]This research was funded by the U.S. Department of Transportation (USDOT) grant number DTRT12-G-UTC17 through the University of Idaho's National Institute for Advanced Transportation Technology (NIATT)

to communication information from vehicles to legacy traffic controllers via RSUs. Thus, cutting edge technology can be used together with equipment, which could predate it by decades. We hope that this proof-of-concept may inspire practical applications to improve signal timing procedures, e.g., to reduce fuel consumption.

The specifics of the contributions of this thesis can be summarized as follows:

1. A communication framework was established allowing communication between DSRC equipment. Specifically, communication primitives needed to be derived to facilitate V2V and V2I message exchanges.

2. The support framework allowing communication with the traffic controller was derived. This communication included mechanisms to change timing parameters in the traffic controller.

3. A multi-threaded application was developed to implement the ETSS and its associated decision support algorithms.

4. The ETSS was verified in a fully equipped signalized intersection, including the traffic cabinet with a controller, support hardware, and traffic lights.

1.5 Thesis Outline

The remainder of this thesis is organized as follows:

Chapter 2, *Background and Overview*, gives an overview of the application developed. This chapter introduces the DSRC technology and provides a basic outline of the Wireless Access in Vehicular Environments (WAVE) system. Fundamental concepts like accessing the transmission medium in DSRC systems and Basic Safety Message, which is the most important message giving information about the state of a vehicle, are explained in detail. Since in our application the roadside infrastructure needs to communicate with the traffic controller, which is required to be National Transportation Communication for ITS Protocol (NTCIP) compliant, the foundation for this infrastructure will be established and an overview of this protocol along with the traffic signal timing concepts is given.

Chapter 3, *System Architecture*, first presents the system hardware configuration for the proposed application. Subsequently, the designed software architecture is discussed, including the algorithm and implementation details of the application. The motivation for creating our own custom applications are discussed here.

Chapter 4, *Experimental Validation of ETSS*, provides an overview of a field experiment. It further describes various scenarios that the ETSSis designed to work for. Finally, the results obtained from the field experiments are presented.

Chapter 5, *Conclusion and Future Works*, present the conclusions, along with some directions for future work.

Several appendices are provided. Appendix A, *Communicating with the Traffic Controller*, presents framework implementation details. Appendix B, *Toolchain Setup*, explains the procedures for downloading the Software Development Kit (SDK) and installing it on the development machine. Subsequently, Appendix C, *Setup Development Environment*, lists steps to set up the development environment by importing projects into the Eclipse Integrated Development Environment (IDE). The Eclipse IDE has been used throughout this research project to write and debug C programming code. A list of NTCIP Global Objects is given in Appendix D, *NTCIP Object Identifiers*. Finally, Appendix E, *Arada Locomate Application Parameter List*, provides a list of application parameters available in the Arada Locomate application.

Chapter 2

Background and Overview

2.1 Application Overview

The Eco-Traffic Signal System (ETSS) consists of a fixed RSU, which is installed at the intersection, OBU equipped vehicles, and a Traffic Controller connected to traffic signals. A scenario for ETSS is shown in Figure 2.1, which depicts a truck approaching the intersection sending BSMs to the RSU. The RSU first verifies if the vehicle is a heavy load. Next, the application checks to see if the approaching vehicle satisfies certain conditions, i.e., if the speed of the vehicle is over a predefined threshold speed, and if the time to reach the intersection is before a predetermined threshold time, and current signal status. A heavy vehicle satisfying both conditions qualifies for an extension of the green light if the traffic signal is green. Thus, if the vehicle qualifies for the signal extension, the RSU sends a Green Signal Hold command to the traffic controller, thereby allowing the heavy load to pass the intersection. Once the vehicle passes the intersection, the application releases the hold command, allowing the normal traffic cycle to continue. By extending the green cycle, the qualifying vehicle avoids a stop, thereby saving fuel associated with idling and consequent acceleration.

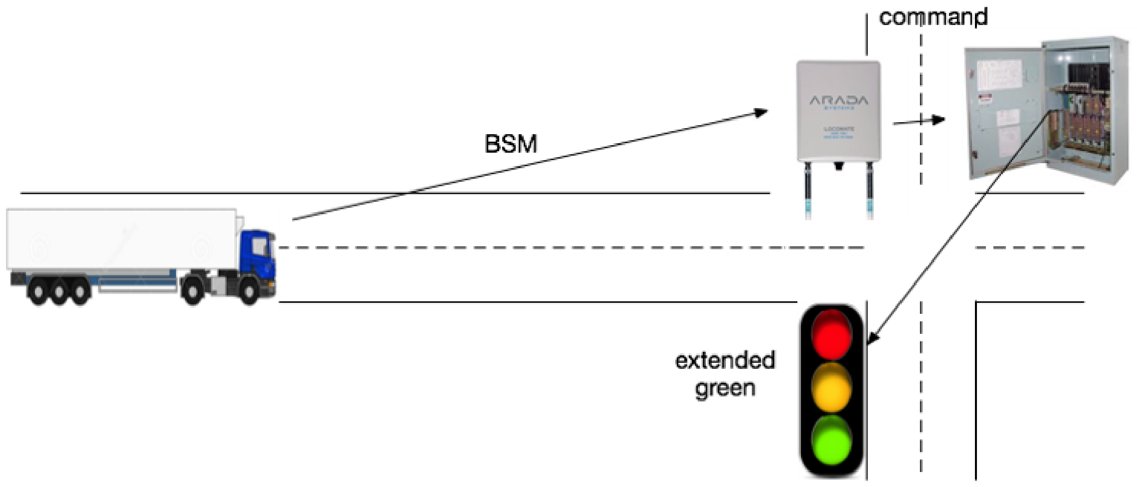


Figure 2.1: Eco-Traffic Signal System Applications

2.2 Dedicated Short Range Communications

As previously indicated, communication of a vehicle's OBU with the RSU and other OBUs is based on DSRC, which provides short to medium range wireless communication support between moving and stationary devices. Specifically, DSRC is a wireless protocol similar to WiFi that implements the V2V and V2I communication. In the United States, the Federal Communications Commission (FCC) has allocated 75 MHz of the spectrum in the 5.9 GHz band for DSRC. The spectrum consists of seven 10 MHz channels from 5.850 GHz to 5.925 GHz, as shown in Figure 2.2. There is one Control Channel (CCH), i.e. channel CH178, and six Service Channel (SCH)s, i.e., CH172, CH174, CH176, CH180, CH182, and CH187. The remaining 5 MHz band is reserved for further use [16] [20] [21].

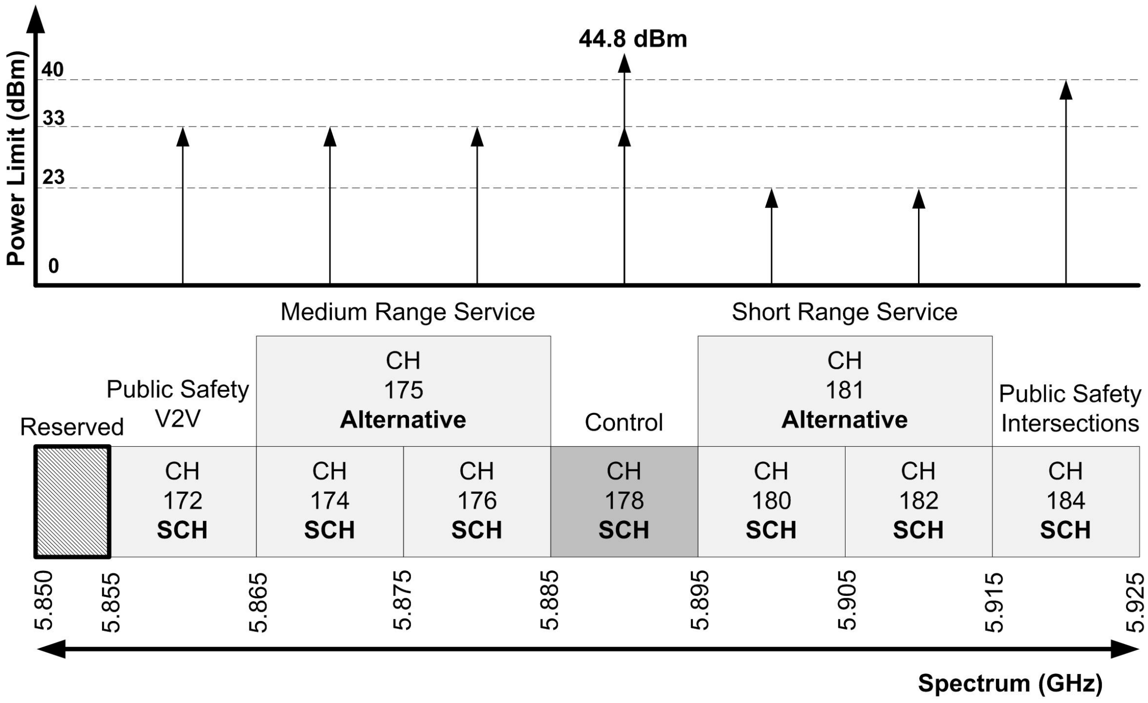


Figure 2.2: DSRC Channel Spectrum

The USDOT proposed power limits of the DSRC standard as shown in the upper part of Figure 2.2. Public safety DSRC channel CH172 is designated for applications involving the safety of life and property. Similarly, private channels CH174, CH175, and CH176 are used to implement small and medium range operations. Transmissions in these three channels should not exceed power levels of 33dBm. In the control channel, i.e., CH178, public safety transmissions should not exceed 44.8dBm, whereas the private transmissions are limited to 33dBm. The DSRC channels CH180, CH181, and CH182 are used for short range services. Public safety and private safety transmissions in these channels should not exceed 23dBm. Finally, DSRC channel CH184 is designated for public safety applications and their operations shall not exceed 40dBm.

2.3 WAVE Protocol Stack

The IEEE 802.11p [13] standard, which is the approved amendment to the IEEE 802.11 standard, added WAVE for vehicular communication. The WAVE protocol architecture with its major components is shown in Figure 2.3.

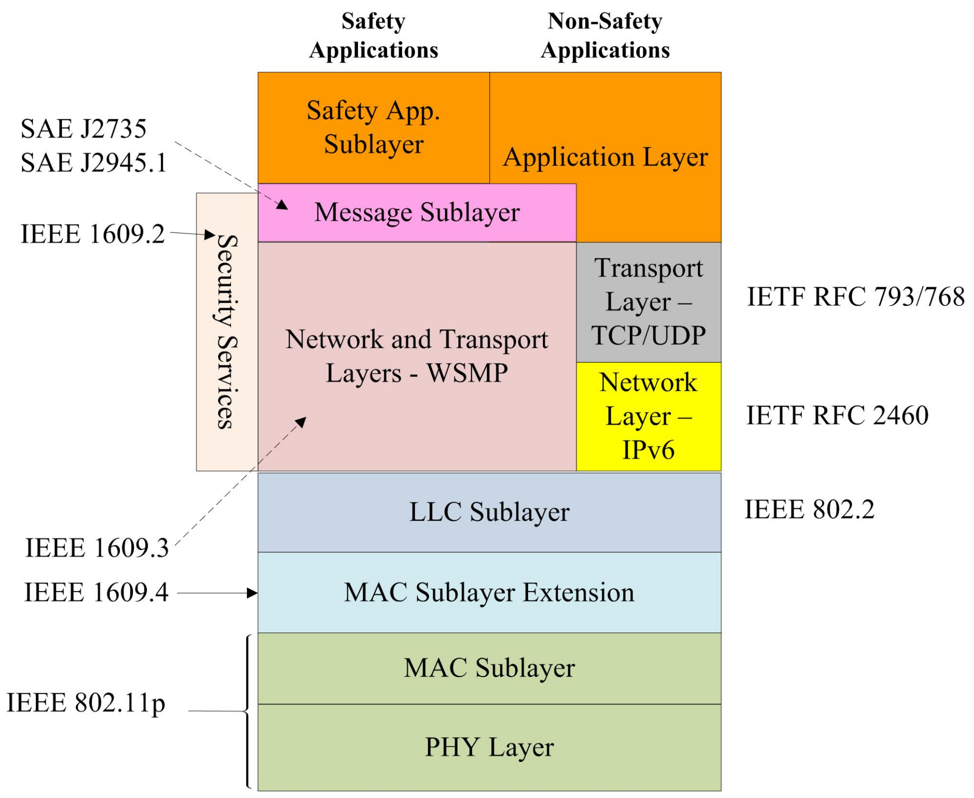


Figure 2.3: WAVE Protocol Stack

The PHY protocol defined in IEEE 802.11 [13], which was published in 2007, specified 5/10/20 MHz channels. A change to this edition has been proposed in IEEE 802.11p [13]. This amended version adopts Orthogonal Frequency-Division Multiplexing (OFDM) modulation on 10 MHz channels in the 5.9 GHz frequency band. In contrast, Wi-Fi, defined in IEEE 802.11, implements the OFDM PHY modulation on the 20 MHz channels. The data rates and subcarrier spacing in WAVE PHY are half that of the Wi-Fi PHY. Therefore, the WAVE OFDM receiver is more sensitive to carrier frequency offset [22].

WAVE units might require time to switch between control and service channels, i.e., the CCH and SCHs. The Medium Access Control (MAC) sublayer extension IEEE 1609.4 [19] is dedicated to controlling such multichannel operation [23]. In general, IEEE 802.11p and IEEE 1609.4 are used to describe the physical and the MAC layer of the WAVE system respectively. The Logical Link Control (LLC) elements are defined in IEEE 802.2 [15], and the Internet Protocol (IP) traffic is sent and received through this layer. Furthermore, IEEE 1609.2 [17], with its two supported stacks, i.e., the Internet Protocol version 6 (IPv6) stack and the WAVE Short Message Protocol (WSMP) stack, i.e. IEEE 1609.3 [18], is used to describe the Network and Transport Layers of the Transmission Control Protocol (TCP)/User Datagram Protocol (UDP) stack. The WSMP is a WAVE network layer protocol that supports high priority and time sensitive communication [24]. The main motivation for developing the WSMP was to reduce the 52 bytes header overload of the typical UDP/IPv6. The format of a WSMP packet is depicted in Figure 2.4. The packet contains 11 bytes of header information. Verification of a valid WSMP

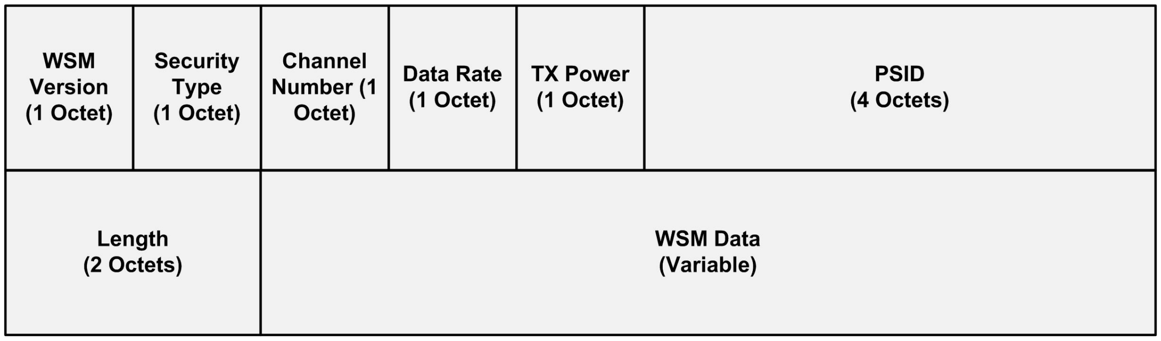


Figure 2.4: WSMP Packet format

packet is done via the WSM version number. If this version number is not supported by the device, the received packet is discarded [22]. The Security Type identifies the nature of the packet, i.e., Unsecured, Signed or Encrypted. The radio parameters are directly controlled by the Channel Number, Data Rate, and TX Power. The Provider Service ID (PSID) field is similar to the port number of the TCP/UDP packet, which identifies the upper layer application that will process the WSM data. The Length field specifies the total number of bytes in the WSM data. Security Services are described by the IEEE 1609.2 standard protocol.

2.4 MAC Protocol

The IEEE 802.11 Distributed Coordination Function (DCF) MAC protocol is based on Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA), and originally developed for Wireless Local Area Networks (WLANs). This DCF MAC protocol has been adopted by the IEEE 802.11p standard for DSRC.

The Carrier Sense Multiple Access (CSMA) mechanism with collision avoidance is shown illustrated in Figure 2.5. In DCF, all nodes compete for channel access by using the CSMA/CA

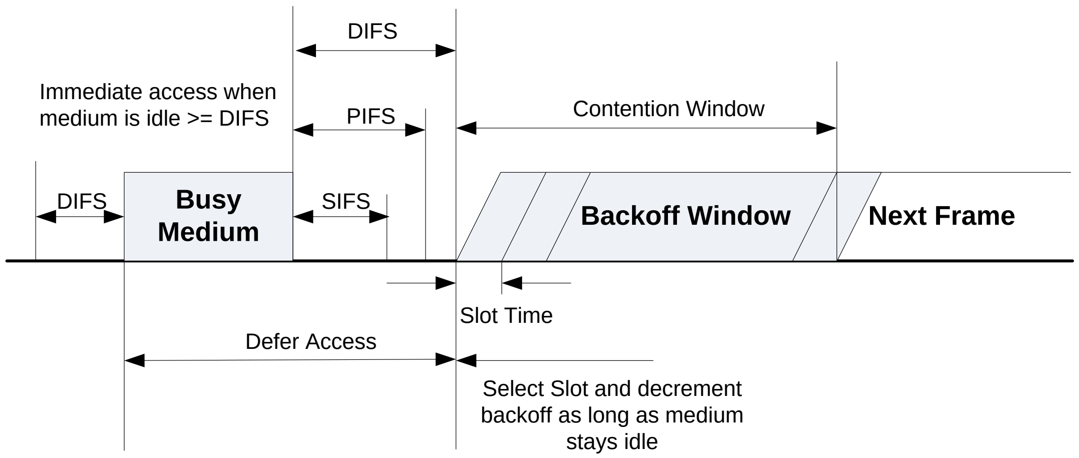

DIFS
Immediate access when medium is idle >= DIFS
PIFS
Contention Window
DIFS
Busy Medium
SIFS
Backoff Window
Next Frame
Slot Time
Defer Access
Select Slot and decrement backoff as long as medium stays idle

Figure 2.5: CSMA

protocol. When a transmitter has a packet to send, the channel must be sensed idle for a certain period of time equal to the distributed interframe space (DIFS). If the channel becomes busy in that period of time, a backoff process is initiated, which randomly sets its backoff counter within the range of its Contention Window (CW). The channel access is deferred until the channel becomes idle again. The backoff timer is decreased as long as the medium is sensed to be idle for a DIFS. It is frozen when a transmission is detected in the medium and resumed when the channel is detected as idle again for a DIFS interval. The station transmits the packet when the backoff counter reaches 0. The size of CW depends on the history of transmissions [32]. In the 802.11p MAC sub-layer an Enhanced Distributed Channel Access (EDCA) mechanism was added with Arbitration Interframe Space (AIFS) for each Access Category (AC).

2.5 ETSS Equipment Details

The ETSS was not just specified, but it was also implemented. The DSRC devices used in the ETSS implementation are the LocoMate$^{\text{TM}}$ OBU and RSU developed by Arada Systems [33]. The block diagram of a LocoMate device is shown in Figure 2.6. These devices are equipped with a 680 MHz Microprocessor without Interlocked Pipeline Stages (MIPS) processor and integrate IEEE 802.11p and Bluetooth capabilities [33, 34]. They also contain 16 MB of Flash memory, 64 MB of SDRAM, a Gigabit Ethernet Interface, an integrated Global Positioning System (GPS) device, and transceivers with external RF antennas. By default, the device transmits its position and other information continuously on the safety channel, encoded in a $BasicSafetyMessage$ format [35].

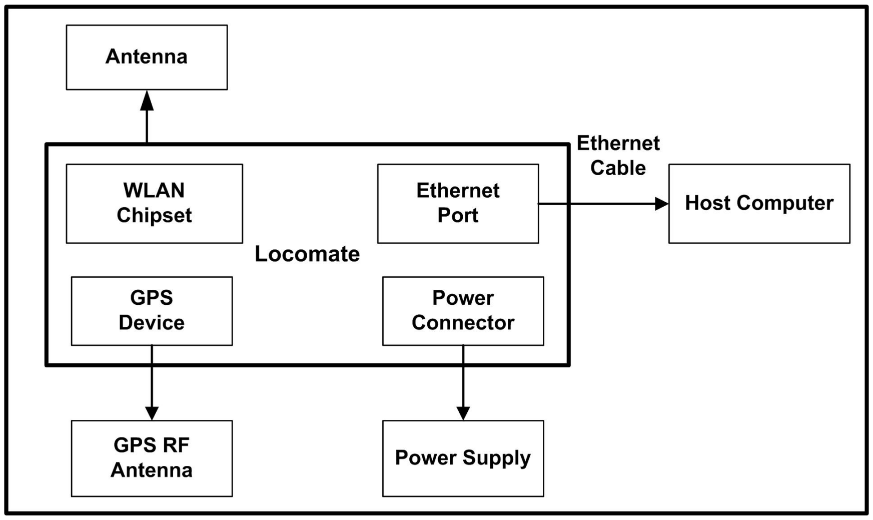
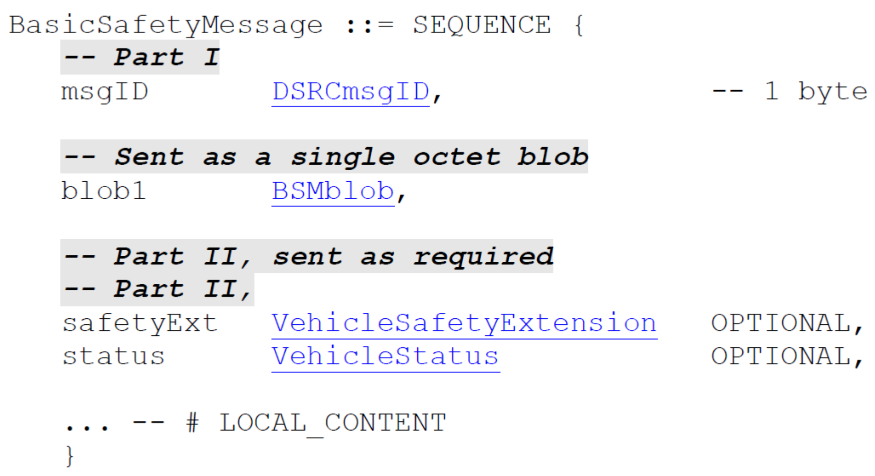


Figure 2.6: Block Diagram of LocoMate with Host Computer Connection (adapted from [35])

2.6 Basic Safety Message

The DSRC Message Set Dictionary [25] specifies a message set, its data frames, and data elements. This dictionary contains 15 message types, 72 Data Frames, 146 Data Elements, and 11 External Data Entries. The connected vehicle safety applications are built around the standard SAE J2735 [25]. The BSM is the most important message type for these applications. BSMs are often referred to as the "heartbeat" messages. A BSM consists of two parts. The first part, which is mandatory, includes the core data elements, such as vehicle size, position, speed, heading, acceleration, and brake system status. The second part of a BSM contains a variable set of data elements that are optional. It is added to Part I and depends on various events, e.g. Anti-Lock Brake System (ABS) activation. Its availability varies and depends on the vehicle model. They are transmitted less frequently in comparison to the Part I of a BSM. The BSM is usually transmitted over DSRC and for safety applications the range of transmission provides a 300m detection range [26]. BSMs are broadcast to nearby vehicles at a rate of 10 messages per second, i.e., one BSM every 100 ms.

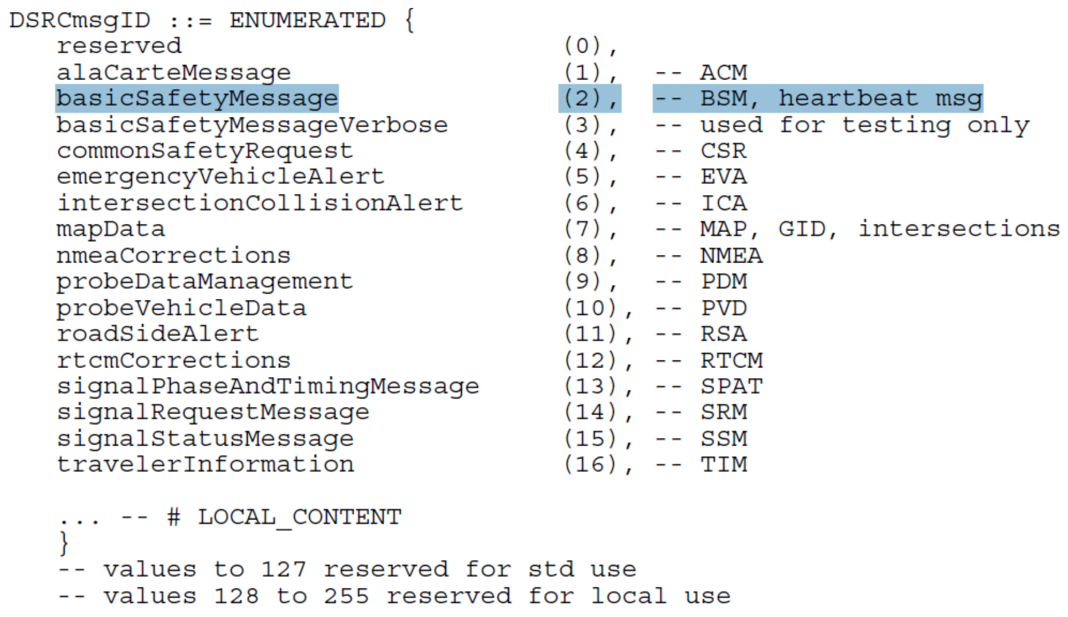
Figure 2.7 shows the Abstract Syntax Notation One (ASN.1) definition of the BSM [25]. The ASN.1 is a standard and a formal notation used for describing data in telecommunications and computer networking. It describes the rules and structures for representing, encoding, transmitting, and decoding the data. Four top level discrete parts (*DSRCmsgID, BSMblob, VehicleSafetyExtension* and *VehicleStatus*) form a BSM. The first one-byte data element, DSR-



```
BasicSafetyMessage ::= SEQUENCE {
   -- Part I
   msgID        DSRCmsgID,                 -- 1 byte

   -- Sent as a single octet blob
   blob1        BSMblob,

   -- Part II, sent as required
   -- Part II,
   safetyExt    VehicleSafetyExtension    OPTIONAL,
   status       VehicleStatus             OPTIONAL,

   ... -- # LOCAL_CONTENT
   }
```

Figure 2.7: ASN.1 definition of the BSM

CmsgID, is an extensible enumeration. This data element differentiates one message type from another. A list of DSRC message types is shown in Figure 2.8 with the BSM ID highlighed.



```
DSRCmsgID ::= ENUMERATED {
   reserved                          (0),
   alaCarteMessage                   (1),   -- ACM
   basicSafetyMessage                (2),   -- BSM, heartbeat msg
   basicSafetyMessageVerbose         (3),   -- used for testing only
   commonSafetyRequest               (4),   -- CSR
   emergencyVehicleAlert             (5),   -- EVA
   intersectionCollisionAlert        (6),   -- ICA
   mapData                           (7),   -- MAP, GID, intersections
   nmeaCorrections                   (8),   -- NMEA
   probeDataManagement               (9),   -- PDM
   probeVehicleData                  (10), -- PVD
   roadSideAlert                     (11), -- RSA
   rtcmCorrections                   (12), -- RTCM
   signalPhaseAndTimingMessage       (13), -- SPAT
   signalRequestMessage              (14), -- SRM
   signalStatusMessage               (15), -- SSM
   travelerInformation               (16), -- TIM

   ... -- # LOCAL_CONTENT
   }
   -- values to 127 reserved for std use
   -- values 128 to 255 reserved for local use
```

Figure 2.8: Enumeration of DE_DSRC_MessageID

Each of the assigned possible legal values for messages is represented as a named item in the enumeration present in the C programming language. It includes message identifiers for *BasicSafetyMessage*(BSM), *CommonSafetyRequest*(CSR), *EmergencyVehicleAlert*(EVA), *IntersectionCollisionAlert*(ICA), *RoadSideAlert*(RSA), *SignalPhaseAndTimingMessage*(SPAT), etc.

Following the DSRCmsgId is the 38 byte BSMblob. The mandatory part of the BSM message is shown in Figure 2.9. Normally, the data element BSMBlob is build up from the

```
BasicSafetyMessageVerbose ::= SEQUENCE {
   -- Part I, sent at all times
   msgID        DSRCmsgID,              -- App ID value, 1 byte

   msgCnt       MsgCount,               -- 1 byte
   id           TemporaryID,            -- 4 bytes
   secMark      DSecond,                -- 2 bytes
   -- pos       PositionLocal3D,
   lat          Latitude,               -- 4 bytes
   long         Longitude,              -- 4 bytes
   elev         Elevation,              -- 2 bytes
   accuracy     PositionalAccuracy,     -- 4 bytes

   -- motion    Motion,
   speed        TransmissionAndSpeed,   -- 2 bytes
   heading      Heading,                -- 2 bytes
   angle        SteeringWheelAngle,     -- 1 bytes
   accelSet     AccelerationSet4Way,    -- 7 bytes

   -- control   Control,
   brakes       BrakeSystemStatus,      -- 2 bytes

   -- basic     VehicleBasic,
   size         VehicleSize,            -- 3 bytes

   -- Part II, sent as required
   -- Part II,
   safetyExt    VehicleSafetyExtension OPTIONAL,
   status       VehicleStatus          OPTIONAL,
   ... -- # LOCAL_CONTENT
   }
```

Figure 2.9: J2735 BSM ASN.1 Notation

GPS position and the vehicle motion readings. In case of the blob defined in Figure 2.9, a large number of other data items must be built up in the correct order [25]. It should be noted that the ASN.1 library does not perform content checking for such blobs, only the length is checked. Any additional checking becomes the responsibility of the application layer to perform.

2.7 National Transportation Communication for ITS Protocol (NTCIP)

The NTCIP is a family of standards for transmitting data and messages between different devices used in ITS [27]. It is the key to managing the ITS infrastructure. According to the American Association of State Highway and Transportation Officials (AASHTO), the Institute of Transportation Engineers (ITE), and the National Electrical Manufacturers Association (NEMA), the NTCIP allows for interoperability and interchangeability between computers and electronic traffic control equipment from different manufacturers [30]. On one hand, interoperability refers to the capability of placing different types of devices on the same communication circuit. On the other hand, interchangeability means the capability to use different brands or models of the same device within a system. The NTCIP compliant devices exchange data interpreted through common communication interfaces. For a successful communication, the same specification is used at each end of a data transmission, which is called a communication protocol. It is somehow analogous to a language consisting of a set of alphabets, vocabulary, and grammar rules [27].

The NTCIP 1202 - Actuated Signal Controller (ASC) defines an open and standard communications protocol for data exchange between software applications and traffic signal hardware. It defines elements for controlling, managing, and monitoring actuated traffic signal controller units such as phases, detectors, coordination, time base control, and preemption. NTCIP provides communications standards for two fundamentally different types of ITS communications, which will be discussed next.

Center to Field (C2F) Communications: This type of communication normally involves devices at the roadside communicating with a central management system. An example of this type of communications is a traffic signal management system communicating with traffic signal controllers at the intersection. Other examples that include C2F are a traffic management system controlling CCTV cameras, advisory radio transmitters, environmental sensors, and traffic count stations on roadways.

Center to Center (C2C) Communications: In this type of communications, messages are send between two or more central management systems. C2C involves peer-to-peer communications between any number of system computers in many-to-many networks [27], which is similar to the Internet. The role of NTCIP in the National ITS Architecture is shown in Figure 2.10.

For both C2F and C2C applications, NTCIP supports systems and devices used in traffic, transit, emergency management, traveler information, and planning systems. Figure 2.11 illustrates how various transportation management systems and devices can be integrated using NTCIP.

2.8 NTCIP Standards Framework

Similar to the layering approach to data communications adopted by the International Organizations of Standards (ISO), NTCIP uses a layered approach to communications standards. The Open Systems Interconnection (OSI) Reference Model defines seven layers, each being responsible for performing a particular role in the transmission of data over a medium. NTCIP

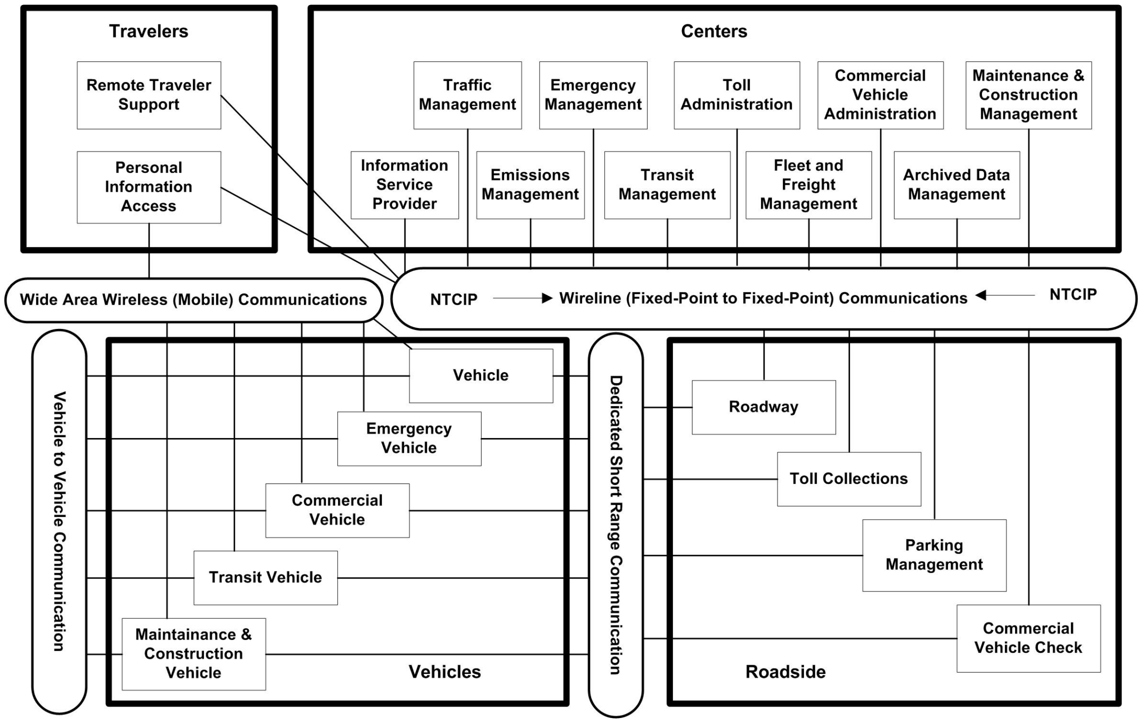

Travelers
Remote Traveler Support
Personal Information Access
Centers
Traffic Management
Emergency Management
Toll Administration
Commercial Vehicle Administration
Maintenance & Construction Management
Information Service Provider
Emissions Management
Transit Management
Fleet and Freight Management
Archived Data Management
Wide Area Wireless (Mobile) Communications
NTCIP
Wireline (Fixed-Point to Fixed-Point) Communications
NTCIP
Vehicle to Vehicle Communication
Vehicle
Emergency Vehicle
Commercial Vehicle
Transit Vehicle
Maintainance & Construction Vehicle
Vehicles
Dedicated Short Range Communication
Roadway
Toll Collections
Parking Management
Commercial Vehicle Check
Roadside

Figure 2.10: NTCIP and National ITS Architecture (Source: [27])

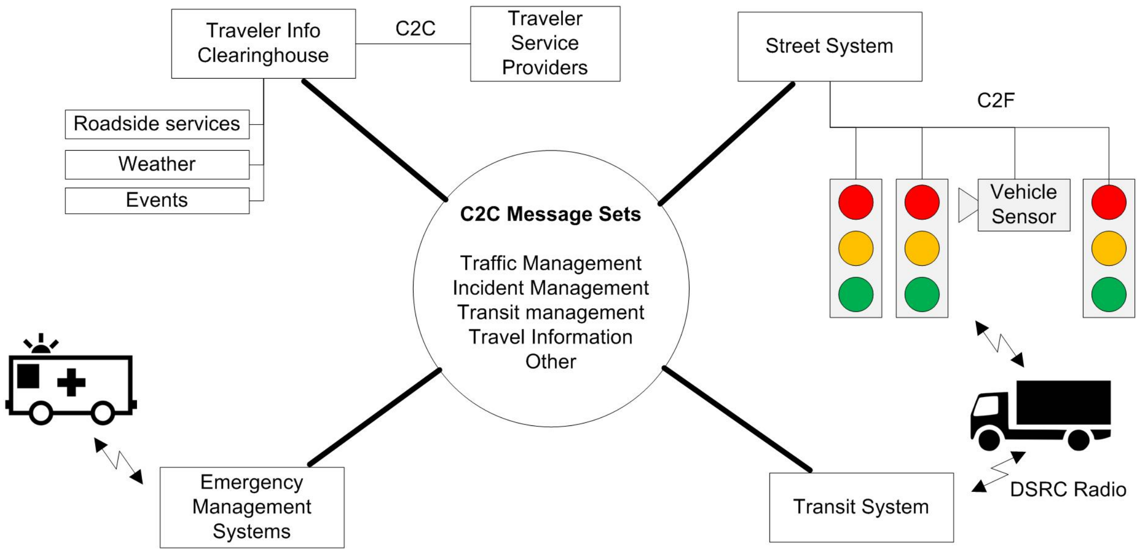


Figure 2.11: ITS Integration Using NTCIP (Source: [27])

uses the term "levels" instead of "layers" to distinguish the hierarchical architecture from those defined by the ISO. The five NTCIP levels are Information Level, Application Level, Transport Level, Subnetwork Level and Plant Level. Figure 2.12 illustrates the five levels of the NTCIP framework and Figure 2.13 shows the mapping of NTCIP levels to the OSI layers. These levels are described below.

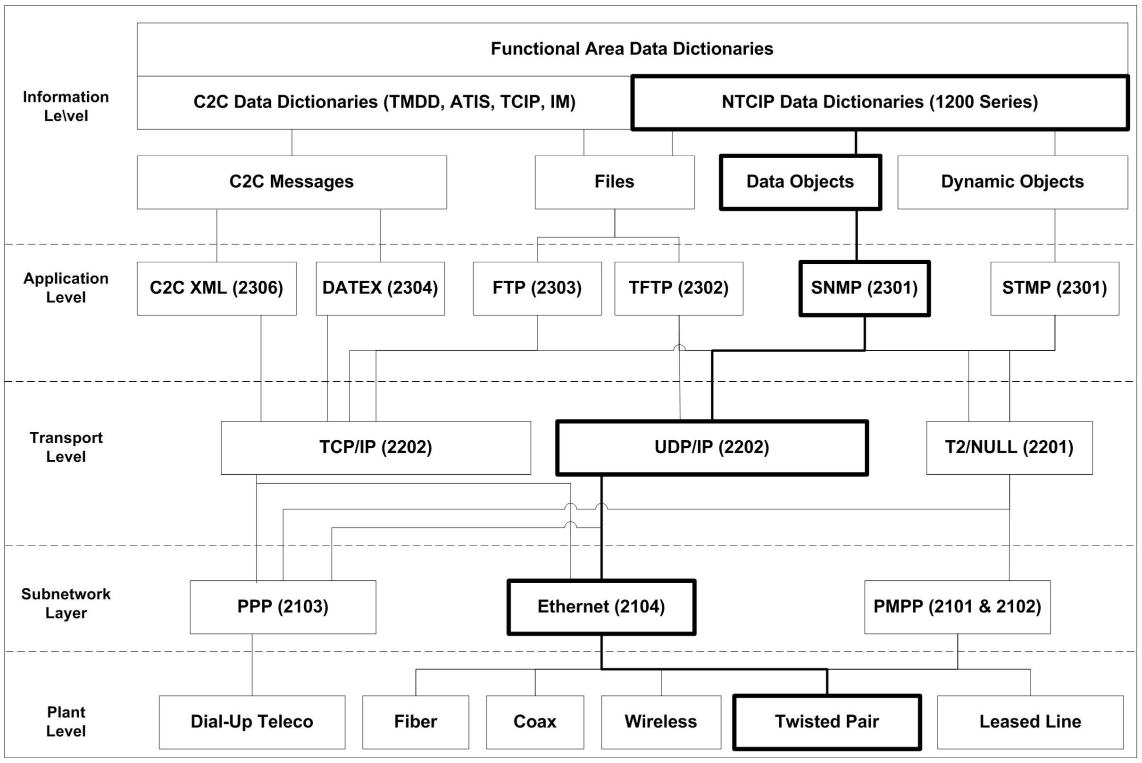


Figure 2.12: NTCIP Framework

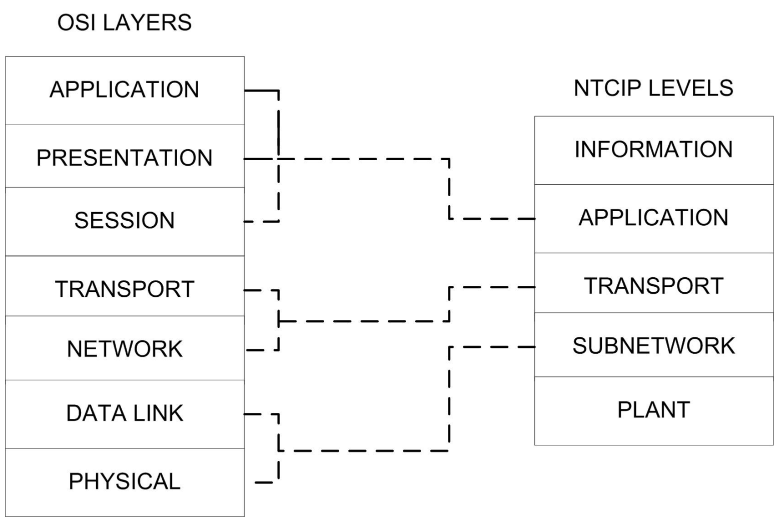


Figure 2.13: OSI Layers and NTCIP Levels mapping

(i) Plant Level: The Plant Level includes the communications infrastructure over which the NTCIP communication standards are to be used. This level consists of the physical transmission media used for communication. It includes copper wire, coaxial cable, fiber optic cable, wireless, and twisted pair. This plant level selection will have a direct impact on the selection of an appropriate Subnetwork Level with which it must interface. However, the NTCIP standards do not prescribe any one media type over another, it is totally an infrastructure choice.

(ii) Subnetwork Level: This Level defines the rules and procedures for exchanging data between two adjacent devices over a chosen communications medium. These standards are roughly equivalent to the Data Link and Physical Layer of the OSI model. This level also contains standards for the physical interfaces, e.g., modem, network interface card, and the data packet transmission methods, such as Point-to-Point Protocol (PPP), Ethernet, or Point-to-Multi-Point Protocol (PMPP).

(iii) Transport Level: The transport level contains standards for data packet subdivision, packet reassembly, and routing, when needed, e.g., TCP, UDP, or IP. The transport level is roughly equivalent to the Transport and Network Layers of the OSI model.

(iv) Application Level: This level contains standards for the data packet structure and session management, e.g., Simple Network Management Protocol (SNMP), Simple Transportation Management Protocol (STMP), File Transfer Protocol (FTP), or Trivial File Transfer Protocol (TFTP). These standards are roughly equivalent to the Session, Presentation and Application Layers of the OSI model.

(v) Information Level: The Information Level contains standards for the data elements, objects, and messages that need to be transmitted. This level comprises of C2C data dictionaries and standard publications from the NTCIP 1200 series, which is illustrated in Figure 2.12. This is similar to defining a dictionary and phrase list within a language. These standards are above the traditional ISO seven-layer OSI model and represent the functionality of the system to be implemented.

2.9 SNMP, MIBs and OIDs

The Simple Network Management Protocol [31], SNMP, is a popular protocol for network management. It allows servers to share information about their current state. Additionally, SNMP also provides a channel through which an administer can modify pre-defined values. The SNMP is an application-level protocol initially designed by the Internet community to run over UDP/IP. However, it can be forced to run over TCP/IP. It is a simple, but bandwidth inefficient protocol for Center to Field (C2F) applications. This protocol is suitable only for networks with high bandwidth or low volumes of messages.

The SNMP consists of an SNMP Manager, Managed Devices, SNMP Agents and Management Information Database or a Management Information Base (MIB) that are explained below.

(i) SNMP Manager: An SNMP manager is a separate entity that is configured to poll an SNMP agent for information. It sends query requests to SNMP agents. This entity is mainly responsible for getting responses from agents and setting variables in agents. The commands defined in the SNMP protocol include *GetRequest*, *GetNextRequest*, *GetBulkRequest*, *SetRequest*, *InformRequest*, and *Response*.

(ii) Managed Devices: Managed devices are the network nodes that require some form of monitoring and management. Such devices include routers, switches, servers, workstations, and printers. These nodes implement SNMP interfaces that allow unidirectional (read-only) or bi-directional (read and write) access to node-specific information.

(iii) Agent: An SNMP agent is a program that runs on a managed device. It gathers information about the local system and stores it in a format that can be queried. The agent collects the Management Information Base (MIB) from the device locally and makes it available to the SNMP manager. An agent configures which managers should have access to its information. The agents respond to commands defined by the SNMP protocol.

(iv) Management Information Base (MIB): The MIB is a hierarchical, pre-defined structure that stores information that can be queried. An MIB module defines a collection of application specific objects. It contains a standard set of control values defined for hardware nodes on a network. SNMP allows the extension of these standard values with values specific to a particular agent through the use of private MIBs.

The MIB is a top-down hierarchical tree, where each node is labeled with an identifying number or string. The identifying number and string for a node should be unique for a particular level of the hierarchy. Successive identifiers of the nodes, starting at the root of the tree, identify each node in the tree by forming a series of unique identifications separated by dots. This address is known as an Object Identifier (OID). MIBs are comprised of managed objects, which are

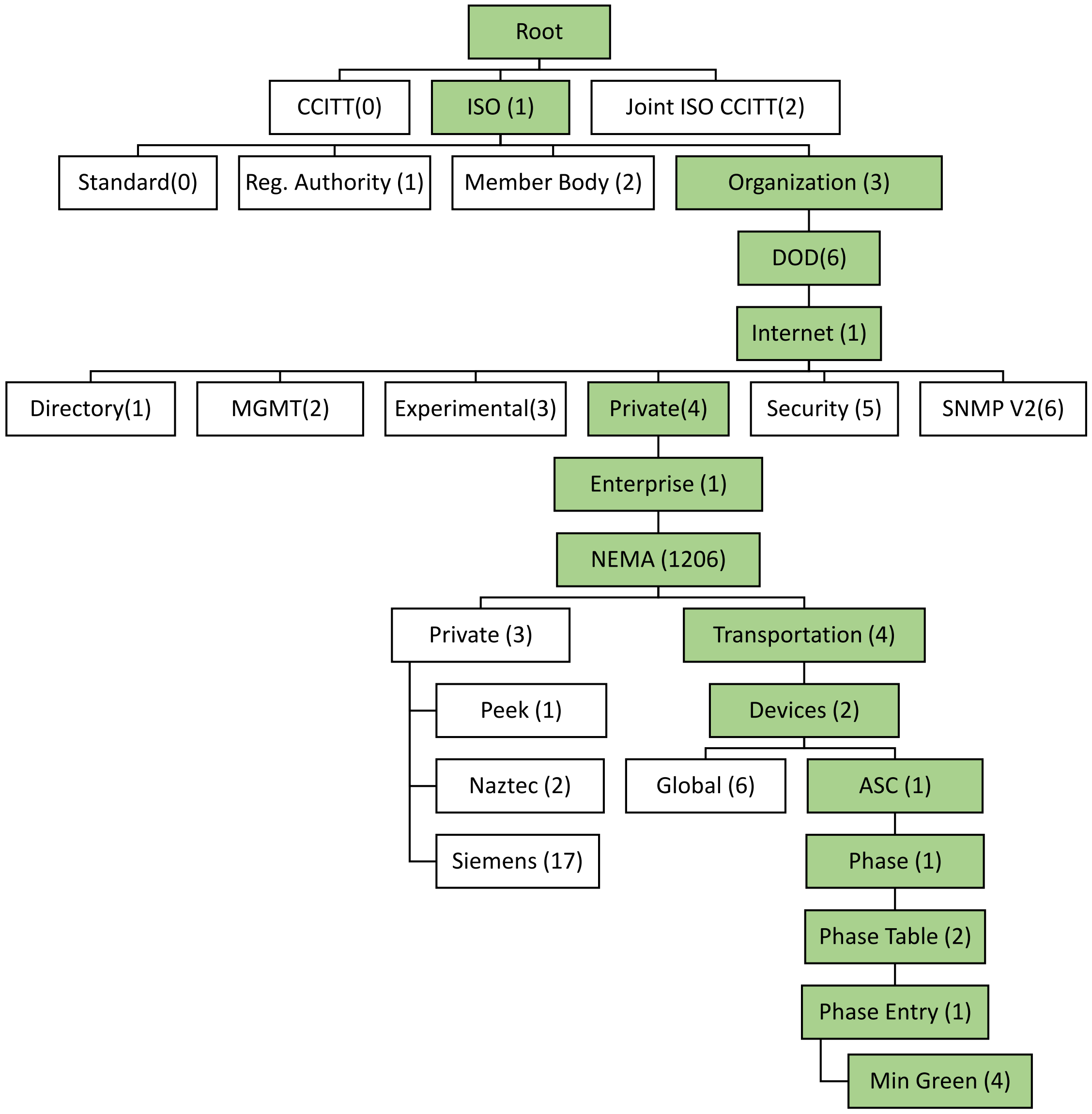


Figure 2.14: Object Identifier for Min Green (1.3.6.1.4.1.1206.4.2.1.1.2.1.4)

identified by the OID and uniquely represent specific characteristics of a managed device. In the MIB, every Object ID is organized hierarchically, which can be represented in a tree structure with individual variable identifier. Typically, an Object ID is a dotted list of integers. For example, the OID for “Min Green” is: 1.3.6.1.4.1.1206.4.2.1.1.2.1.4. The tree structure for this OID is shown in the Figure 2.14. The path of “Min Green” is identified by green fields in the figure.

2.10 Traffic Signal Timing

Traffic system timing will be explained using a specific example. Consider a four-way intersection with one main street and a crossing side street, as shown in Figure 2.15. The main street is assumed to be busy in comparison to the side street. The figure also shows a state diagram of traffic signals for this intersection, one on the main street and the other on the side street. Note that for a single traffic light only one color is displayed at a time. The logical progression between the four states can be seen in the figure.

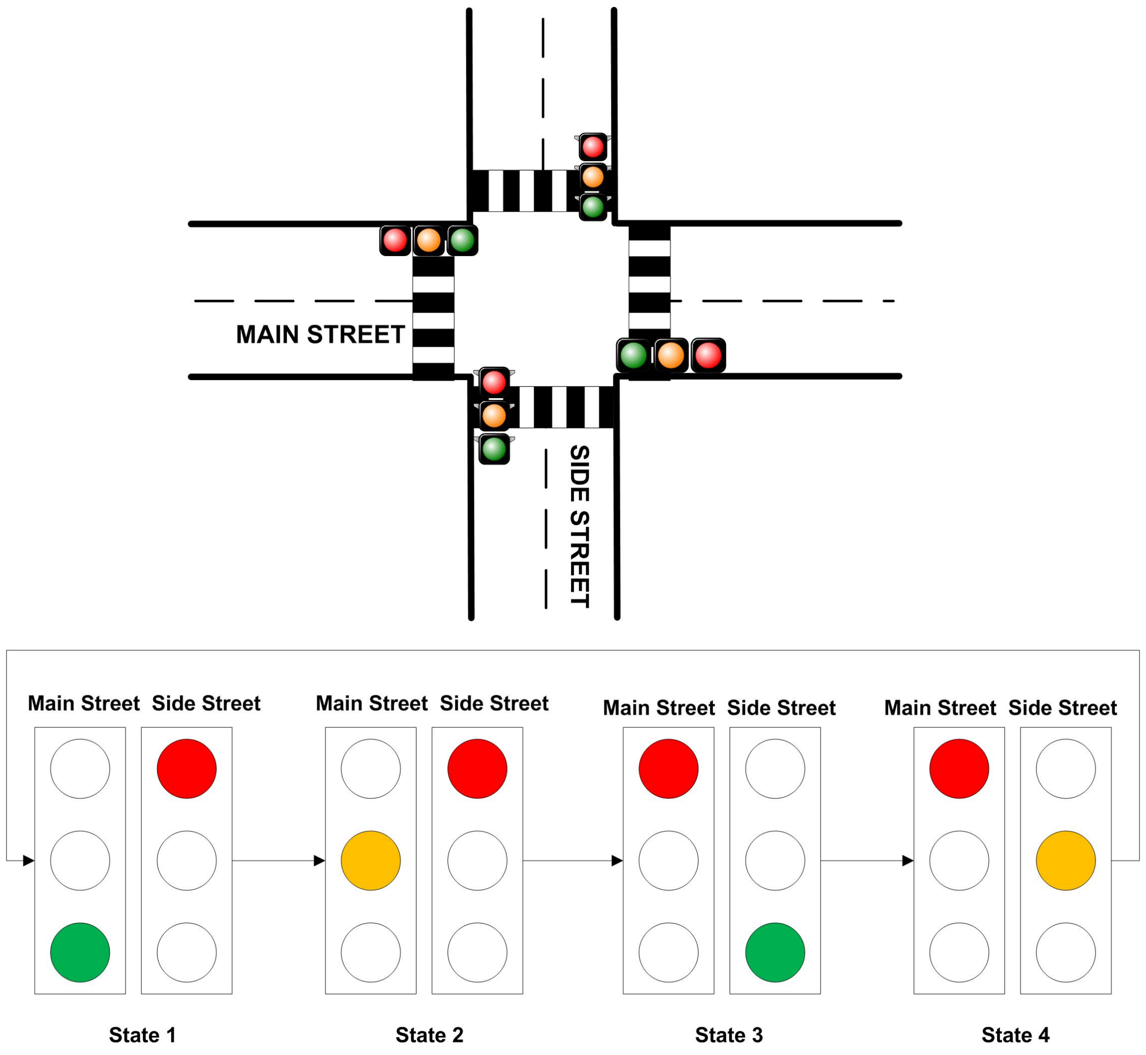


Figure 2.15: Traffic Signal Control at four way intersection

In State 1, the main street signal is Green and the side street signal is Red. This state persists for at least the minimum green time or as long as there is no vehicle waiting on the side street, assuming vehicle detection capability. But, if there is a vehicle waiting in the side street and the minimum green time has passed, the waiting vehicle should get a chance to pass the intersection. Therefore, in the next state, i.e., State 2, other vehicles in the main street are warned about the changing light. This transition should last for an appropriate time. Now, every vehicle on the main street has been warned about the changing light. So, in State 3, the main street signal turns Red and the side street signal turns Green, which allows the vehicle waiting in the side street to pass the intersection. This transition is going to last at least for another minimum green time period or as long as vehicles are detected in the side street, but not larger than the "Max Green" time. In State 4, the side street signal turns yellow warning about the changing light. Finally, the state transition goes from State 4 to State 1. This indication to a particular traffic or pedestrian link is known as a Phase, which is basically a green interval plus the change and clearance intervals that follow it. The concept of phases is illustrated in Figure 2.16. Usually, a four-way intersection with two lanes consists of 8 phases. Even numbered phases (2, 4, 6, 8) imply going straight, whereas the odd numbered phases (1, 3, 5, 7) are for taking a left turn.

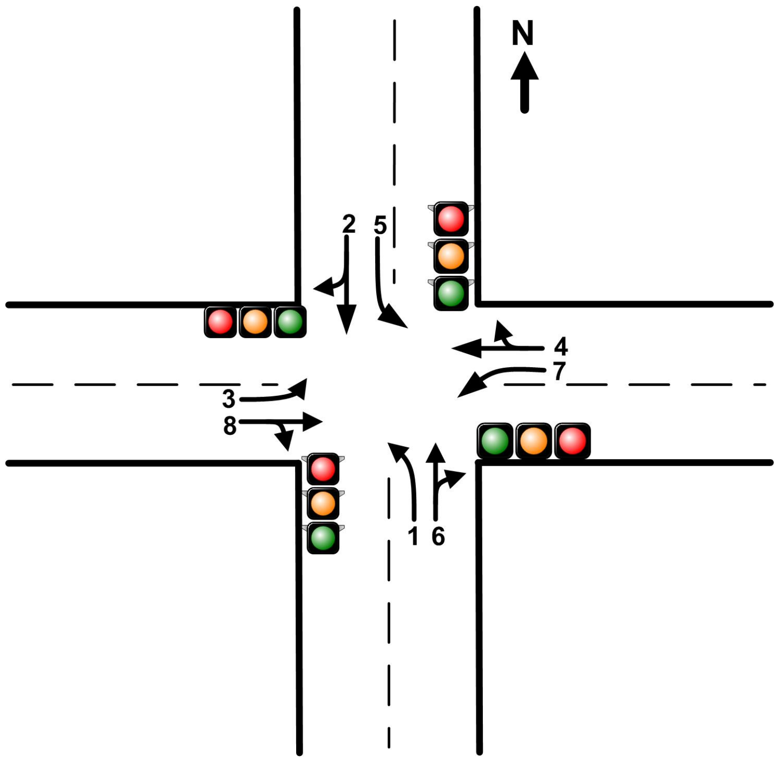


Figure 2.16: Phase Information

Traffic Signal Timing is a technique used to determine the right-of-way at an intersection. Signal timing involves decisions that the traffic lights shall provide. Such decision can be a green time and/or pedestrian walk signal. There are various modes, Pre-timed or Actuated (semi, fully), that a traffic signal can operate in, as will be described next.

2.10.1 Pre-Timed Operation

In pre-timed control, there is a series of intervals that are fixed in duration. Each signal phase is serviced in a programmed sequence, which is repeated. Therefore, the traffic signal behavior at such fixed cycle length intersection is a deterministic sequence, which changes its signal state, e.g., from green to yellow to red. Pre-timed control is suitable for closely spaced intersections with consistent traffic volumes and patterns [36].

2.10.2 Actuated Operation

Vehicle detection in actuated control is used for providing information about the traffic demand to the controller. In response to the vehicle detectors, the intervals in the controller are called and extended. Modern traffic signal controllers are designed for actuated operation. Actuated signals consist of two types: semi-actuated and fully-actuated.

While driving a vehicle, drivers only see the cycle of changing Green, Yellow, and Red colors in traffic lights at the intersection. The timing of such transitions is explained in Figure 2.17. Primarily, there are three parameters, i.e., minimum green, maximum green and extension time, which play an important role in such transition. A signal remains green for at least the Minimum Green time. At the very end of the Minimum Green time, if a vehicle is detected by the vehicle detection technology, a request for more time is sent to the traffic controller so that the detected vehicle can pass the intersection. It should be noted that the total green time, including extension times, can not exceed the maximum green time. Traffic signals at the intersection change their state from green to yellow, to red and this cycle continues. Timing periods for such parameters must be NTCIP compliant, and requests for timing periods violating this compliance are ignored. As a result, it cannot be that a yellow period of a "split second" is initiated, which most certainly would be hazardous.

(a) Semi-Actuated: In Semi-actuated, vehicle detectors are installed on the minor street, whereas the traffic signal on the major street remains green. The signal or phase associated with the major road is non-actuated, i.e., detection information is not provided. In contrast to the pre-timed control, semi-actuated control reduces delay due to the availability of actuated control on the minor streets. However, continuous vehicle calls on the phases associated with

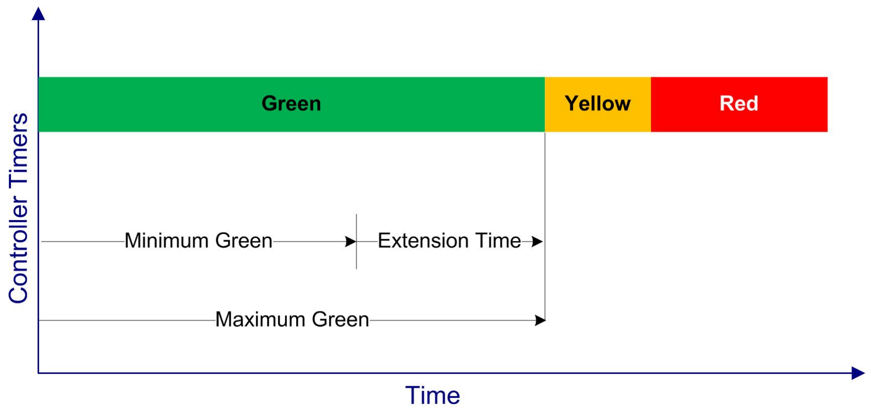


Figure 2.17: Signal Timing

minor streets can cause excessive delay to the major road. Therefore, maximum green and passage time parameters should be set appropriately.

(b) Fully-Actuated: In Fully-Actuated control, all phases at the intersection are actuated by vehicle detection loops and/or pedestrian push buttons. This type of actuated operation is highly suitable for the intersections with high traffic density and nondeterministic traffic patterns that vary especially during the day time. It also helps improve performance at the intersections with lower traffic volume. This results in fewer stops, which ultimately leads to a decrease in fuel consumption.

Chapter 3

System Architecture

3.1 System Hardware Configuration

The modern traffic intersection is a blend of various controllers, sensors, and networking devices. In the ITS, which plays an important role in safety and traffic management, the traffic signal controllers are a core component. This project uses the Econolite Advanced System Controller Series 3 (ASC/3-2100) model, which is currently installed in many intersections. It provides NTCIP compliance and most advanced and innovative traffic technologies.

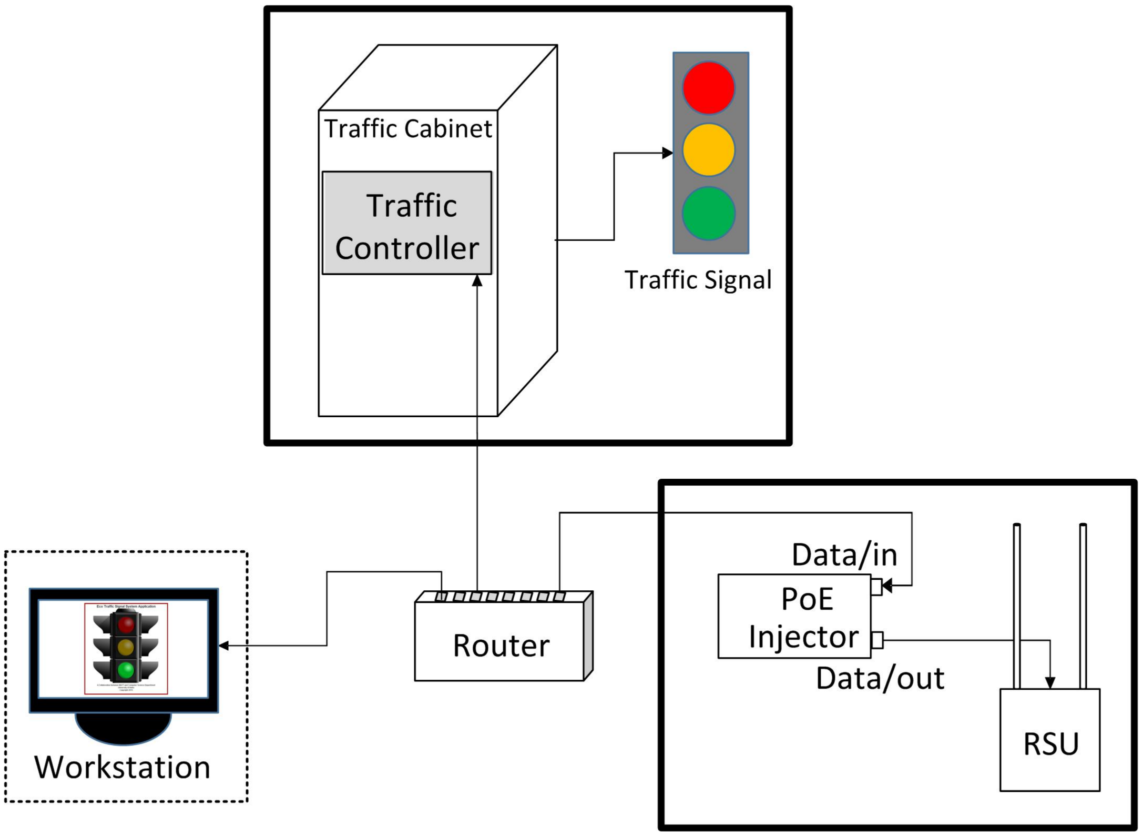


Figure 3.1: System Hardware Configuration

The proposed hardware architecture of the ETSS is shown in Figure 3.1. The traffic controller, which is housed in a metal cabinet and placed near an intersection, is the core of this architecture. Information such as red, yellow, and green timing, along with the coordination plan timing, are all stored in this device. Furthermore, it reads sensor inputs and controls the traffic lights. In order to determine the current state of the traffic lights, sensors are connected directly to the controller, allowing it to combine vehicle detection information with preprogrammed timing controls [37]. The controller is connected to a networking device, e.g., a router, using an Ethernet cable.

Another key component in the architecture shown in Figure 3.1 is the RSU. It is an add-on component, powered by a Power over Ethernet (PoE) injector, which is connected to the same network. The PoE passes electrical power along with data on the Ethernet cable. The injector consists of two ports - "Data IN" and "Data OUT". The Data IN port is connected to the Ethernet hub, the router, and Data OUT port is connected to the PoE device, the RSU. As discussed in Section 2.1, in ETSS, the RSU first collects BSMs from vehicles equipped with OBUs. Then it executes the ETSS algorithm to determine potential action, in which case it sends a command to the traffic controller, e.g., a green extension.

The final component in this architecture is a workstation or a monitor[1], which is an optional component. It should be noted that it is not necessary to have the workstation near every metal cabinet in the intersection. They can be placed remotely and should be able to access the RSU in order to execute various RSU commands, such as starting or halting an application. This workstation could be placed in a traffic center that monitors various traffic signals around the city.

3.2 Software Architecture and Communications

The software architecture of ETSS is shown in Figure 3.2. It consists of three components: an *On Board Component*, a *Road Side Component*, and a *Traffic Controller Component*. The On Board Component communicates only with a Road Side Component, whereas the Road Side Component communicates with both, the On Board, and the Traffic Controller Components. There is no way for an On Board Component to directly communicate with the Traffic Controller Component.

As shown in the figure, the architecture as a whole consist of a total of seven modules, where three modules are in the On Board Component, and four are in the Road Side Component. Each module is assigned to a specific task as described below.

[1]Workstation is an optional module for configuration and observation. It is not part of a functioning eco-traffic signal system.

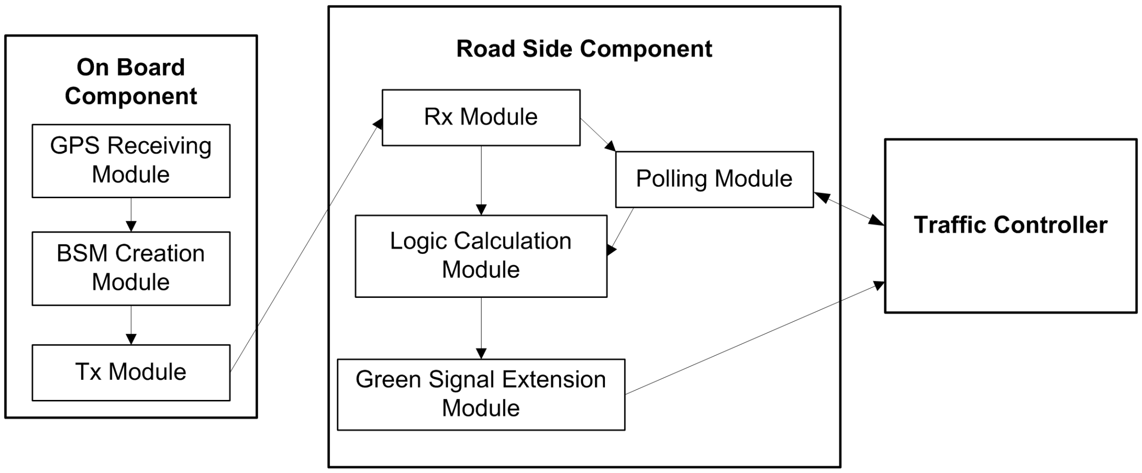


Figure 3.2: Software Architecture

A) On Board Component: The vehicle's OBU implements the following modules in ETSS. Once OBUs will be installed in production vehicles, these modules do not have to be explicitly implemented, as their functionality will be already available from the car manufacturer.

i) **GPS Receiving Module**: The OBU equipped with a GPS device is used as a navigation device for a vehicle. This module is mainly responsible for polling the vehicle's current GPS coordinates and the speed.

ii) **BSM Creation Module**: After receiving the longitude, latitude, elevation, and speed from the GPS Receiving Module, the BSM Creation Module creates and encodes a BSM. Implementation details about structuring a BSM will be discussed in Section 3.2.1.

iii) **Tx Module**: This module takes the encoded BSM and transmits it over the DSRC safety channel. It also logs the number of BSMs transmitted.

B) Road Side Component: The RSU is responsible for providing the following modules, which, unlike the OBUs above, will need to be implemented for ETSS.

i) **Rx Module**: This module continuously senses the DSRC safety channel for BSMs transmitted by Tx Modules of OBUs. During this polling, it furthermore logs the total number of packets received.

ii) **Polling Module**: This is a module in the roadside component that communicates bidirectionally with the traffic controller. Specifically, after a BSM is received, and provided the BSM is of a qualifying vehicle, it polls the traffic controller for the current status of the green light in all available phases of the intersection.

iii) **Logic Calculation Module**: Based on the status of the green signal and the received BSM, this module processes the data and calculates metrics such as the velocity of the vehicle, its distance from the intersection, and the projected time to reach the intersection. An algorithm, which will be described in Subsection 3.3.4, then determines if the Green Signal Extension Module, described next, should be activated or deactivated.

iv) **Green Signal Extension Module**: This module issues the *Green Signal Hold* command to the traffic controller if instructed by the Logic Calculation Module to extend the green light. Such hold command has an impact of a brief duration of time. If no further hold command is issued, e.g., by subsequent requests of the Logic Calculation Module, the hold is deactivated automatically after the brief duration. Alternatively, the green signal hold could be terminated forcefully by a command *phaseControlGroupForceOff*.

C) Traffic Controller Component: This component communicates with the Polling Module and the Green Signal Extension Module of the roadside component in the RSU. There are mainly two tasks this component is designed to perform. In the first task, a value representing the current status of the green signal in all phases is sent to the Polling Module. The second task of this component is to receive a request to hold the green signal for a particular phase. This request is sent to the traffic controller by the Green Signal Extension Module of the Road Side Component. The traffic controller addresses such request sent by issuing a signal hold command only if the green signal is on. Otherwise, the request is discarded. It should be noted that the ETSS does do not require any modifications to the traffic controller hardware or software. It simply reads the timing parameters from the traffic controller and may issue configuration changes.

3.2.1 Creating a Basic Safety Message

The basic structure of a BSM has been discussed in Section 2.6. The usage of these beacon messages is shown in Figure 3.3, where the OBU of the Remote Vehicle (RV) transmits BSMs and the OBU of the Host Vehicle (HV) receives the BSMs at regular intervals. The basic steps for creating and encoding a BSM is shown in Algorithm 1.

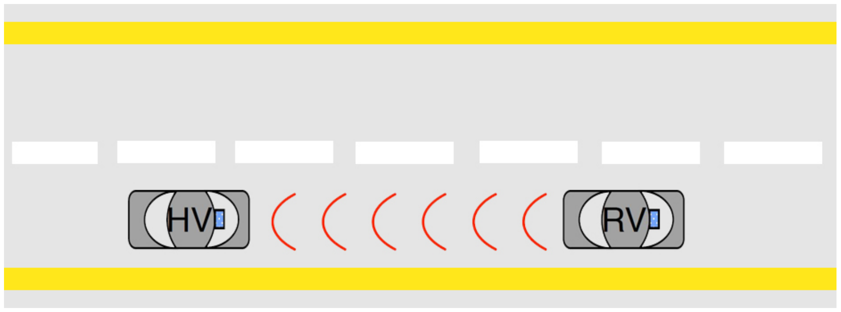


Figure 3.3: Host Vehicle and Remote Vehicle

Algorithm 1 Constructing a Basic Safety Message Algorithm

```
procedure BUILDWSMREQUESTPACKET( )

    $bsm.msgID.buf[0] \leftarrow DSRCmsgID\_basicSafetyMessage$
    $bsm.blob1.size \leftarrow 38$

    $/************PART\ \ I************/$
    $bsm.blob1.buf[0] = count\ \ \%\ \ 127$
    $memcpy(bsm.blob1.buf + 1, val, 4)//setTemporaryID$
    $memcpy(bsm.blob1.buf + 5, val, 2)//setSecMark$
    $memcpy(bsm.blob1.buf + 7, val, 4)//setlatitude$
    $memcpy(bsm.blob1.buf + 11, val, 4)//setlongitude$
    $memcpy(bsm.blob1.buf + 15, val, 2)//setelevation$
    $memcpy(bsm.blob1.buf + 17, val, 4)//setacceleration$
    $memcpy(bsm.blob1.buf + 21, val, 2)//setSpeed$
    $memcpy(bsm.blob1.buf + 23, val, 2)//setHeading$
    $memcpy(bsm.blob1.buf + 25, val, 1)//setAngle$
    $memcpy(bsm.blob1.buf + 26, val, 7)//setAccelerationSet$
    $memcpy(bsm.blob1.buf + 33, val, 2)//setBreakStatus$
    $memcpy(bsm.blob1.buf + 35, val, 3)//setvehicleSize$

    $/************PART\ \ II************/$
    $vhType.buf[0] \leftarrow [0..15]$
    $vehicleIdent.vehicleType \leftarrow vhType$
    $status.vehicleIdent \leftarrow vehicleIdent$
    $bsm.status \leftarrow status$

    $Encode\ \ using\ \ der\_encode\_to\_buffer()$
```

The first level of a BSM is *DSRCMsgID*. It differentiates one message type from another. According to standard SAE J2735 the value for a *basicSafetyMessage* is 2, which was shown in Figure 2.8. This value is assigned to the message id of a BSM in Line 3 of Algorithm 1. The second level of the discrete part of a BSM is BSMBlob. It consists of 38 bytes, packed with information like message count, temporary id, timestamps in milliseconds, position, motion, brake status, and vehicle size. The latitude, longitude, and speed from Part I of the BSM are important elements for the ETSS. The third and fourth levels of a BSM form it's Part II and are optional. However, in ETSS, the fourth level, i.e., VehicleStatus, is an important field. The VehicleStatus element is defined as *DE_VehicleType* in SAE J2735 [25]. A numerical value referring to the vehicle type of an RV is encapsulated in the VehicleStatus element. This value is stored in a variable *vhType*, which is shown in Line 22 of the algorithm. The default value for this variable is 0 (zero), meaning that the vehicle type is not known. The value to be assigned to a vehicle type depends on the type of vehicle. SAE J2735 defines 16 entries for vehicle types, from 0 to 15, as shown in Table 3.1. Graphical representation of each vehicle types is shown in Figure 3.4. Since ETSS considers only heavy loaded vehicles, the vehicle type should be given a value in a range between 7 and 15, inclusive.

After creating the *BasicSafetyMessage*, the packet is encoded into an Abstract Syntax Notation (ASN) format. This encoding is done with the help of a Distinguished Encoding Rules (DER) encoder using the *der_encode_to_buffer*() function, which is shown in Line 27.

BasicSafetyMessage is an abstract data type. Lines 7 to 19 of this algorithm explain how data is stored in their respective placeholders in a BSM. As explained earlier, the latitude, longitude, and speed mentioned in Line 10, Line 11, and Line 14 respectively, are the most important information for ETSS. Details on obtaining those fields are explained below.

Listing 3.1: Calculation Logic for Longitude and Latitude

```
if(wsmgps.latitude == 0){
latitude_val = 900000001;
} else {
latitude_val = (long) ((wsmgps.latitude) * 10000000);
}

if(wsmgps.longitude == 0){
longitude_val = 1800000001;
} else{
longitude_val = (long) ((wsmgps.longitude) * 10000000);
}
```

Latitude and Longitude Calculation: The Data Elements *DE_Latitude* and *DE_Longitude* shown in Figure 3.5 are described in standard SAE J2735. The GPS receiver might provide an invalid location of the OBU if a value fault is experienced. Not receiving a GPS signal could also result in invalid values for latitude and longitude. To avoid such scenarios, maximum range values allowed by the standard for those metrics are set. According to standard SAE J2735, latitude and longitude should be represented in 1/10 micro degrees. The values of latitude and

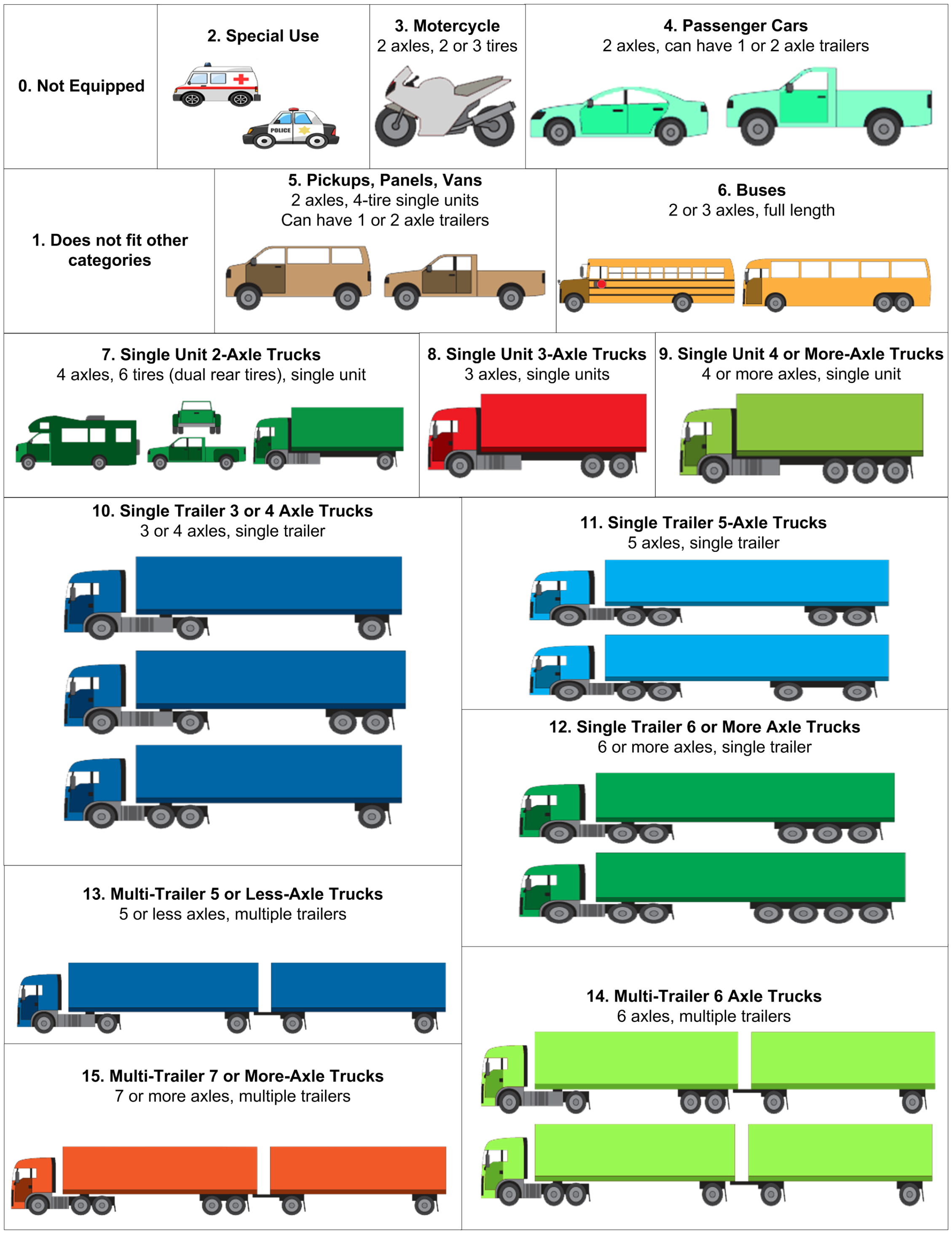

0. Not Equipped
2. Special Use
POLICE
3. Motercycle
2 axles, 2 or 3 tires
4. Passenger Cars
2 axles, can have 1 or 2 axle trailers
1. Does not fit other categories
5. Pickups, Panels, Vans
2 axles, 4-tire single units
Can have 1 or 2 axle trailers
6. Buses
2 or 3 axles, full length
7. Single Unit 2-Axle Trucks
4 axles, 6 tires (dual rear tires), single unit
8. Single Unit 3-Axle Trucks
3 axles, single units
9. Single Unit 4 or More-Axle Trucks
4 or more axles, single unit
10. Single Trailer 3 or 4 Axle Trucks
3 or 4 axles, single trailer
11. Single Trailer 5-Axle Trucks
5 axles, single trailer
12. Single Trailer 6 or More Axle Trucks
6 or more axles, single trailer
13. Multi-Trailer 5 or Less-Axle Trucks
5 or less axles, multiple trailers
14. Multi-Trailer 6 Axle Trucks
6 axles, multiple trailers
15. Multi-Trailer 7 or More-Axle Trucks
7 or more axles, multiple trailers

Figure 3.4: Vehicle Classification (adapted from [38])

longitude are obtained by multiplying the corresponding valid data obtained from the GPS device by 10,000,000 (i.e. 1/10 micro), as shown in Listing 3.1. The maximum range of values

Table 3.1: List of Vehicle Types

Value	Vehicle Type	Description
0	VehicleType_none	Not Equipped, Not known or unavailable
1	VehicleType_unknown	Does not fit any other category
2	VehicleType_special	Special use
3	VehicleType_moto	Motorcycle
4	VehicleType_car	Passenger car
5	VehicleType_carOther	Four tire single units
6	VehicleType_bus	Buses
7	VehicleType_axleCnt2	Two axle, six tire single units
8	VehicleType_axleCnt3	Three axle, single units
9	VehicleType_axleCnt4	Four or more axle, single unit
10	VehicleType_axleCnt4Trailer	Four or less axle, single trailer
11	VehicleType_axleCnt5Trailer	Five or less axle, single trailer
12	VehicleType_axleCnt6Trailer	Six or more axle, single trailer
13	VehicleType_axleCnt5MultiTrailer	Five or less axle, multi-trailer
14	VehicleType_axleCnt6MultiTrailer	Six axle, multi-trailer
15	VehicleType_axleCnt7MultiTrailer	Seven or more axle, multi-trailer

for the latitude is $\pm 90°$, i.e. $-900,000,000$ to $900,000,001$. Similarly, the maximum range of values for the longitude is $\pm 180°$, i.e. $-1800,000,000$ to $1800,000,001$.

```
-- DE_Latitude (Desc Name) Record 57
Latitude ::= INTEGER (-900000000..900000001)
   -- LSB = 1/10 micro degree
   -- Providing a range of plus-minus 90 degrees

-- DE_Longitude (Desc Name) Record 63
Longitude ::= INTEGER (-1800000000..1800000001)
   -- LSB = 1/10 micro degree
   -- Providing a range of plus-minus 180 degrees
```

Figure 3.5: Data Elements for Latitude and Longitude

Speed Calculation: The ETSS requires the Data Frame *DF_TransmissionAndSpeed*. This frame stores the Data Elements *DE_TransmissionState* and *DE_Speed*, which are shown in Figure 3.6. According to standard SAE J2735 *DF_TransmissionAndSpeed* is a string of two bytes (16 bits). The 3 most significant bits (bits 16 - 14) are for the Transmission State and the remaining 13 bits (bits 13 - 1) are for the Speed. In our case the latter is obtained from the GPS device[2].

```
-- DF_TransmissionAndSpeed (Desc Name) Record 63
TransmissionAndSpeed ::= OCTET STRING (SIZE(2))
    -- Bits 14~16 to be made up of the data element
    -- DE_TransmissionState
    -- Bits 1~13 to be made up of the data element
    -- DE_Speed


-- DE_TransmissionState (Desc Name) Record 125
TransmissionState ::= ENUMERATED {
   neutral          (0), -- Neutral, speed relative to the vehicle alignment
   park             (1), -- Park, speed relative the to vehicle alignment
   forwardGears     (2), -- Forward gears, speed relative the to vehicle alignment
   reverseGears     (3), -- Reverse gears, speed relative the to vehicle alignment
   reserved1        (4),
   reserved2        (5),
   reserved3        (6),
   unavailable      (7), -- not-equipped or unavailable value,
                         -- speed relative to the vehicle alignment

   ... -- # LOCAL_CONTENT
   }


-- DE_Speed (Desc Name) Record 99
Speed ::= INTEGER (0..8191) -- Units of 0.02 m/s
          -- The value 8191 indicates that
          -- speed is unavailable
```

Figure 3.6: Data Frame and Data Element for Transmission and Speed

If the GPS device receives invalid data, the transmission state and speed will be calculated using the maximum speed allowed by the standard. The maximum value allowed for the speed is $(2^{13} - 1)$, i.e. 8191, based on the 13 bits allowed for storing a value of the speed. According to standard SAE J2735, a value of 8191 for data element *DE_Speed* indicates that the speed is unavailable, which is shown in Figure 3.6. The derivation of the Transmission State and Speed is shown in Figure 3.7. Level L0 of the figure depicts the original data of 13 bits, representing the speed of the vehicle as obtained from the GPS device. The transmission state and speed needs 16 bits, as required by the standard SAE J2735. Therefore, 3 preceding zeros are added to the original data in Level L0 to create a new data stream of 16 bits as shown in Level L1. Two sets of operations are performed in this newly created stream of 16 bits data, each set producing 8 bits.

In Set B, a mask of 0x00FF in Level L2 is used in an AND operation with the data in Level L1 to get the result in Level L3, as shown in the figure. At Level L4, 8 bits of data are extracted from the data by ignoring preceding 8 zeros available in the data stream at Level L3.

[2]This may be different in future production vehicles, as the information may be provided from the vehicles themselves.

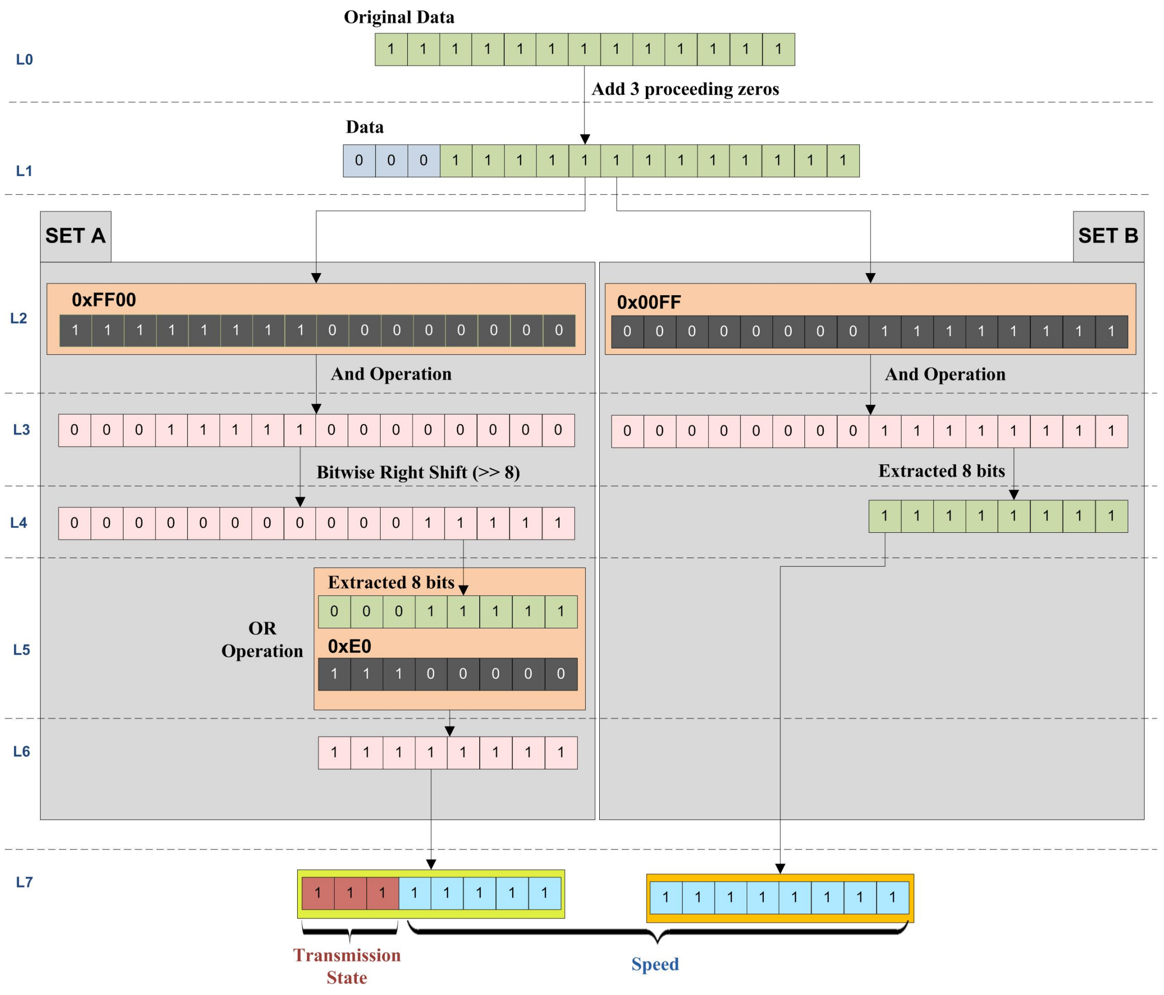


Figure 3.7: Calculating Speed and Transmission

In Set A, a mask of 0xFF00 in Level L2 is used to extract the 8 leftmost bits of the data from Level L1. This AND operation, however still results in 16 bits in Level L3 with the 8 rightmost bits of this steam being all zeros. Each bit at Level L3 is then shifted right 8 times, as shown in Level L4. At Level L5, the preceding 8 zeros are ignored and only the rightmost 8 bits are considered for further operations. The extracted 8 bits at Level L5 are now OR'ed with a mask of 0xE0 and the result is stored at Level L6. This mask 0xE0, i.e., 11100000_2, is currently a static value in ETSS, and the three most significant bits of this mask are all 1s, which indicates that the transmission state is currently unavailable. Later, the value of this mask should be obtained dynamically based on the input provided by sensors available in the vehicle.

Finally, the result in Level L6 of Set A and the result in Level L4 of Set B is brought together in Level L7, which represents the transmission state and the speed of the vehicle. In summary, as shown in Figure 3.7, the three most significant bits of the result from Set A are the transmission state, whereas the remaining 13 bits from both Set A and Set B represent the speed as specified by Standard SAE J2735. Note that the speed of a vehicle is an integer representing units of 0.02 m/s (i.e. $\frac{1}{50} = 0.02$). Therefore, the value received for speed should be multiplied by 50 before performing the operation above at Level L0. The implementation of the above-described approach using the C programming language is shown in Listing 3.2.

Listing 3.2: Calculation Logic for Transmission State and Speed

```
if(wsmgps.speed != GPS_INVALID_DATA){
transmission_speed[0] = (uint8_t)(((uint8_t)(wsmgps.speed
    *50) & 0xFF00) >> 8);
transmission_speed[1] = (uint8_t)(((uint8_t)(wsmgps.speed
    *50) & 0x00FF));
transmission_speed[0] = transmission_speed[0] | 0xE0;
}
else{
transmission_speed[0] = ((8191 & 0xFF00) >> 8);
transmission_speed[1] = (8191 & 0xFF00);
transmission_speed[0] = transmission_speed[0] | 0xE0;
}
```

3.2.2 Transmit Basic Safety Message

The general mechanism for a BSM transmission is given in Algorithm 2. The main idea behind the algorithm is to get valid GPS information, create a BSM, transmit this packet and log the total number of packets transmitted. This algorithm begins by defining two variables *count* and *drops* in Line 2 and Line 3 respectively. These variables are initially set to zero. It should be noted that the number of packets that were not transmitted for some reason was also logged. However, such scenario was never observed.

First of all, the most recent valid GPS information about the RV is extracted, as shown in Line 6 of Algorithm 2. Then, a BSM packet is created. Creation of a BSM has been described in Section 3.2.1. The function *txWSMPacket(int pid, WSMRequest *req)* in Line 8 allows the application to pack a WAVE Short Message (WSM) packet, i.e., $WSMPacket$ and pass a pointer to it for transmission. Based on the status of the transmission of every packet, either the packet sent (*count*) or the packet non-transmitted (*drops*) counter is increased by 1.

3.2.3 Receive Basic Safety Message

The steps to extract the BSM transmitted from the RV is shown in the Algorithm 3. This algorithm also explains how the received packets are decoded. Furthermore, the algorithm

Algorithm 2 Transmit Basic Safety Message Algorithm

```
procedure TXWSMPPKTS( $pid$ )
    $count \leftarrow 0$
    $drops \leftarrow 0$

    while 1 do
        if ($getGPSInfo() == 0$) then
            $buildWSMRequestPacket()$
            if ($txWSMPacket(pid, wsmreq) < 0$) then
                if $drops > (maxInt - 1)$ then
                    $drops = 0$
                    $drops \leftarrow drops + 1$
                else
                    $drops \leftarrow drops + 1$
            else
                if $count > (maxInt - 1)$ then
                    $count = 0$
                    $count \leftarrow count + 1$
                else
                    $count \leftarrow count + 1$
```

implements the steps outlined in the Rx Module of the Road Side Component, which was described in Section 3.2.

Algorithm 3 Receive Basic Safety Message Algorithm

```
procedure RXCLIENT()
    $count \leftarrow 0$
    $pid \leftarrow getpid()$

    while 1 do
        if ($rxWSMMessage(pid, rxmsg) > 0$) then
            extract contents
            decode bsm using ber_decode
            if $count > (maxInt - 1)$ then
                $count = 0$
                $count \leftarrow count + 1$
            else
                $count \leftarrow count + 1$
```

From an implementation point of view, the algorithm to receive a BSM seems to be much simpler than that for transmitting BSMs. The HV receives the WSM packets using the *rxWSMMessage*() function as shown in Line 6 of the algorithm. Received packets are in ASN encoded format, and they are decoded using the generic Basic Encoding Rules (BER) decoder, i.e., *ber_decode*(). Variable *count* defined in Line 2 will be incremented by 1 for every successful reception of the BSM. An attempt to receive a BSM is a continuous poll. In the absence of RVs in the surrounding of the HV, there is a chance for many blank polls, where nothing is received [3].

3.3 Communication with Traffic Controller

In this section, first we list all the object identifiers that will be used in the implementation. Then various data structures along with some support functions that have been created will be explained. Finally, the mechanism to access the traffic controller will be discussed.

3.3.1 NTCIP Objects

Selected NTCIP Global Objects that were used in ETSS are listed in Table 3.2 and will be discussed next. A complete list of such objects is provided in Appendix D.

Table 3.2: List of NTCIP Objects

NTCIP 1202 OID	**Object Name**	**Accessibility**
1.3.6.1.4.1.1206.4.2.1.1.4.1.1	phaseStatusGroupNumber	read-only
1.3.6.1.4.1.1206.4.2.1.1.4.1.2	phaseStatusGroupReds	read-only
1.3.6.1.4.1.1206.4.2.1.1.4.1.3	phaseStatusGroupYellows	read-only
1.3.6.1.4.1.1206.4.2.1.1.4.1.4	phaseStatusGroupGreens	read-only
1.3.6.1.4.1.1206.4.2.1.1.5.1.4	phaseControlGroupHold	read-write
1.3.6.1.4.1.1206.4.2.1.1.5.1.5	phaseControlGroupForceOff	read-write

phaseStatusGroupNumber: This row stores a read-only integer, which contains the Phase StatusGroup number of objects [27]. This value shall not exceed the maximum number of phase groups. The Actuated Controller Unit used during this research project supports a maximum of 8 phases per group. The value for phaseStatusGroupNumber is equal to TRUNCATE[(maxPhases + 7) / 8].

phaseStatusGroupGreens: This is a read-only object that provides a value representing the status of a green signal in all phases at an intersection. It stores an integer ranging from 0 to 255. When a bit is 1, the Phase Green is currently active. When a bit is 0, the Phase Green is NOT currently active. The bits are interpreted as follows:

[3]The reason for using polling, rather than interrupt driven operation for sensing available packets, was due to the DSRC equipment manufacturers' [ARADA] software.

Bit 7 = Phase number = (phaseStatusGroupNumber * 8)

Bit 6 = Phase number = (phaseStatusGroupNumber * 8) - 1

Bit 5 = Phase number = (phaseStatusGroupNumber * 8) - 2

Bit 4 = Phase number = (phaseStatusGroupNumber * 8) - 3

Bit 3 = Phase number = (phaseStatusGroupNumber * 8) - 4

Bit 2 = Phase number = (phaseStatusGroupNumber * 8) - 5

Bit 1 = Phase number = (phaseStatusGroupNumber * 8) - 6

Bit 0 = Phase number = (phaseStatusGroupNumber * 8) - 7

When the traffic controller is queried for the *phaseStatusGroupGreens*, it returns an integer value. The equivalent 8 bit binary number for the integer returned by the traffic controller can be used to identify phases that have their green signal on. This mapping between the individual bit with the associated phase number can be obtained by the above relation. As can be seen in above expression, *phaseStatusGroupNumber* is required to determine the relation between a bit and a phase. The formula to calculate the value for *phaseStatusGroupNumber* is given by:

$phaseStatusGroupNumber = TRUNCATE[(maxPhases + 7)/8]$.

If we consider a scenario with a traffic controller supporting a maximum of 8 phases, then $maxPhases = 8$, and

$phaseStatusGroupNumber = TRUNCATE[(8 + 7)/8]$, or

$phaseStatusGroupNumber = TRUNCATE[1.875]$, results in

$phaseStatusGroupNumber = 1$.

When this value for *phaseStatusGroupNumber* is substituted in the above Bit/Phase number calculation logics, the Bit to Phase mapping is obtained as shown in Figure 3.8. As the bit number increases from 0 to 7, the phase number for the corresponding bit decreases from 8 to 1. For example, let us assume the traffic controller is queried for *phaseStatusGroupGreens* and it returned the value 34_{10}. This value is equivalent to 00100010_2 and thus represents that Phase 2 and Phase 6 have the Green Signal turned on.

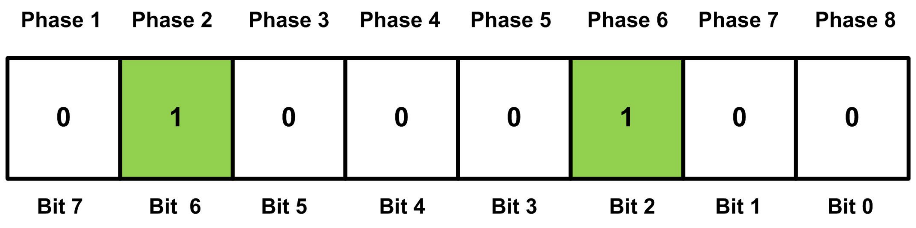


Figure 3.8: Bits to Phases assignment

phaseControlGroupHold: This is a read-write object that is used to allow a remote entity to hold phases in the device when they are Green. When a bit is 1, the device activates the System Phase Hold control for that phase. When a bit is 0, the device does not activate the System Phase Hold control for that phase. The bits are interpreted as follows:

Bit 7 = Phase number = (phaseControlGroupNumber * 8)

Bit 6 = Phase number = (phaseControlGroupNumber * 8) - 1

Bit 5 = Phase number = (phaseControlGroupNumber * 8) - 2

Bit 4 = Phase number = (phaseControlGroupNumber * 8) - 3

Bit 3 = Phase number = (phaseControlGroupNumber * 8) - 4

Bit 2 = Phase number = (phaseControlGroupNumber * 8) - 5

Bit 1 = Phase number = (phaseControlGroupNumber * 8) - 6

Bit 0 = Phase number = (phaseControlGroupNumber * 8) - 7

phaseStatusGroupReds: This is a read-only object that provides phase group red status. It stores an integer ranging from 0 to 255. When a bit is 1, the Phase Red is currently active. When a bit is 0, the Phase Red is NOT currently active. The bits are interpreted as follows:

Bit 7 = Phase number = (phaseStatusGroupNumber * 8)

Bit 6 = Phase number = (phaseStatusGroupNumber * 8) - 1

Bit 5 = Phase number = (phaseStatusGroupNumber * 8) - 2

Bit 4 = Phase number = (phaseStatusGroupNumber * 8) - 3

Bit 3 = Phase number = (phaseStatusGroupNumber * 8) - 4

Bit 2 = Phase number = (phaseStatusGroupNumber * 8) - 5

Bit 1 = Phase number = (phaseStatusGroupNumber * 8) - 6

Bit 0 = Phase number = (phaseStatusGroupNumber * 8) - 7

phaseStatusGroupYellows: This is a read-only object that provides phase group yellow status. It stores an integer ranging from 0 to 255. When a bit is 1, the Phase Yellow is currently active. When a bit is 0, the Phase Yellow is NOT currently active. The bits are interpreted as follows:

Bit 7 = Phase number = (phaseStatusGroupNumber * 8)

Bit 6 = Phase number = (phaseStatusGroupNumber * 8) - 1

Bit 5 = Phase number = (phaseStatusGroupNumber * 8) - 2

Bit 4 = Phase number = (phaseStatusGroupNumber * 8) - 3

Bit 3 = Phase number = (phaseStatusGroupNumber * 8) - 4

Bit 2 = Phase number = (phaseStatusGroupNumber * 8) - 5

Bit 1 = Phase number = (phaseStatusGroupNumber * 8) - 6

Bit 0 = Phase number = (phaseStatusGroupNumber * 8) - 7

phaseControlGroupForceOff: This is also a read-write object similar to the phaseControl-GroupHold. A phase hold command can be terminated in two ways: either by not sending a phaseControlGroupHold command or by sending a phaseControlGroupForceOff. The latter forcefully terminates the hold command on a per phase basis. When a bit is 1, the device activates the System Phase Force Off control for that phase. When a bit is 0, the device does not activate the System Phase Force Off control for that phase. After terminating the phase green, the associated bit is set to 0. The bits are interpreted as follows:

Bit 7 = Phase number = (phaseControlGroupNumber * 8)

Bit 6 = Phase number = (phaseControlGroupNumber * 8) - 1

Bit 5 = Phase number = (phaseControlGroupNumber * 8) - 2

Bit 4 = Phase number = (phaseControlGroupNumber * 8) - 3

Bit 3 = Phase number = (phaseControlGroupNumber * 8) - 4

Bit 2 = Phase number = (phaseControlGroupNumber * 8) - 5

Bit 1 = Phase number = (phaseControlGroupNumber * 8) - 6

Bit 0 = Phase number = (phaseControlGroupNumber * 8) - 7

3.3.2 Data Structures

A keyword called **typedef** is used in the C programming language to give a type a new name. All such identifiers used in ETSS are shown in Listing 3.3. Using **typedef**, one can give a name to user defined data types as well. In ETSS, primitive data types such as char, unsigned char, int, unsigned int, long and unsigned long, are all expressed as new user defined data type. The convention is to express them as Int followed by a number that represents the total number of bits used to store that particular data type. If the data type is unsigned, ‘U’ is inserted at the beginning of the new user defined data type.

Listing 3.3: Typedef for some primitive data types

```
1         typedef char             Int8;
2         typedef unsigned char    UInt8;
3         typedef unsigned char    Byte;
4         typedef int              Int16;
5         typedef unsigned int     UInt16;
6         typedef long             Int32;
7         typedef unsigned long    UInt32;
```

Consider the typedef in Line 1. Here char is a data type that requires 8 bits to store data. So it is expressed as Int8. UInt8 in Line 2 represents the unsigned version of Int8 in Line 1. As a whole, a char is represented as Int8 in ETSS, int as Int16, and long as Int32. The unsigned version of int and long are expressed as UInt16, and UInt32 respectively, whereas an unsigned char can be represented either as UInt8 or a Byte.

Messages of the SNMP will be explained next. Each SNMP message contains a Protocol Data Unit (PDU), which is used for communication between the SNMP manager and SNMP agents. The ASN.1 structure of an SNMP Message is shown in Listing 3.4. It contains four types of PDUs: *GETREQUEST PDU* [A0], *GETNEXTREQUEST PDU* [A1],*GETRESPONSE PDU* [A2], and *SETREQUEST PDU* [A3]. The SNMP manager sends the GetRequest PDU to retrieve one or more requested MIB variables and the GetNextRequest PDU to retrieve the next MIB variable that is specified in the PDU [31]. The SNMP manager also sends the SetRequest PDU to set one or more MIB variables specified in the PDU with the value specified in the PDU. On the other hand, the GetResponse PDU is sent by the SNMP agent in response to a GetRequest, GetNextRequest, or SetRequest PDU, which are sent by the SNMP manager.

Listing 3.4: SNMP Message structure

```
Message::= SEQUENCE {
version INTEGER { version-1(0)},
community OCTET STRING,
data CHOICE {
get-request GetRequest-PDU (with a data type value of 0xA0),
get-next-request GetNextRequest-PDU (data type value = 0xA1),
get-response GetResponse-PDU (data type value = 0xA2),
set-request SetRequest-PDU (data type value = 0xA3
}
}
```

Table 3.3 shows a sample byte stream used in various fields for the SetRequest PDU data. The SNMP message fields for this PDU, which is sent over a typical UDP/IP Transport Profile over an Ethernet Subnetwork profile, is shown in Figure 3.9. This message format is made up of several layers of nested fields. In an SNMP message, the outer layer is a sequence representing the entire SNMP message consisting of the SNMP version (Integer), Community String (Octet

Table 3.3: SetRequest-PDU (set-request)

Field	**Byte Stream**
SEQUENCE - Type and Length (Value is below)	[30][2f]
version - INTEGER of 1 byte, value 0	[02] [01] [00]
community - OCTET STRING of 6 bytes (”public”)	[04] [06] [70] [75] [62] [6c] [69] [63]
data - Type and Length (Value below)	[a3] [22]
request-id	[02] [01] [00]
error-status	[02] [01] [00]
error-index	[02] [01] [00]
variable-bindings SEQUENCE OF	[30] [17]
SEQUENCE	[30] [15]
name	[06] [0d] [2b] [06] [01] [04] [01] [89] [36] [04] [02] [06] [03] [01] [00]
value	[02] [04] [37] [31] [9a] [28]

String), and SNMP PDU (GetRequest, or SetRequest). The *SEQUENCE* has been always a combination of sequence type i.e., 0x30 and the length or the number of bytes that follow. The hexadecimal values for the most common types that are defined by the SNMP are listed below:

- INTEGER: 0x02
- OCTET STRING: 0x04
- NULL (Placeholder): 0x05
- OBJECT IDENTIFIER: 0x06
- SEQUENCE: 0x30
- COUNTER: 0x41

The SNMP *Version* field is represented as an Integer (0x02) of 1 byte (0x01) and the value is zero (0x00). Similarly, the *Community* field is an Octet string (0x04) of 6 bytes (0x06) for “public”, i.e., 0x70 0x75 0x62 0x6c 0x69 0x63. The *SNMP PDU* field contains the body of the SNMP message. It consists of a PDU type, request id, error status, error index and variable bindings. For a SetRequest PDU, the PDU type consists of a PDU tag of 0xA3, as explained earlier in Listing 3.4, followed by the length representing a number of bytes that follow. The *Request Id* is an Integer (0x02) that identifies a particular SNMP request. The *error status*

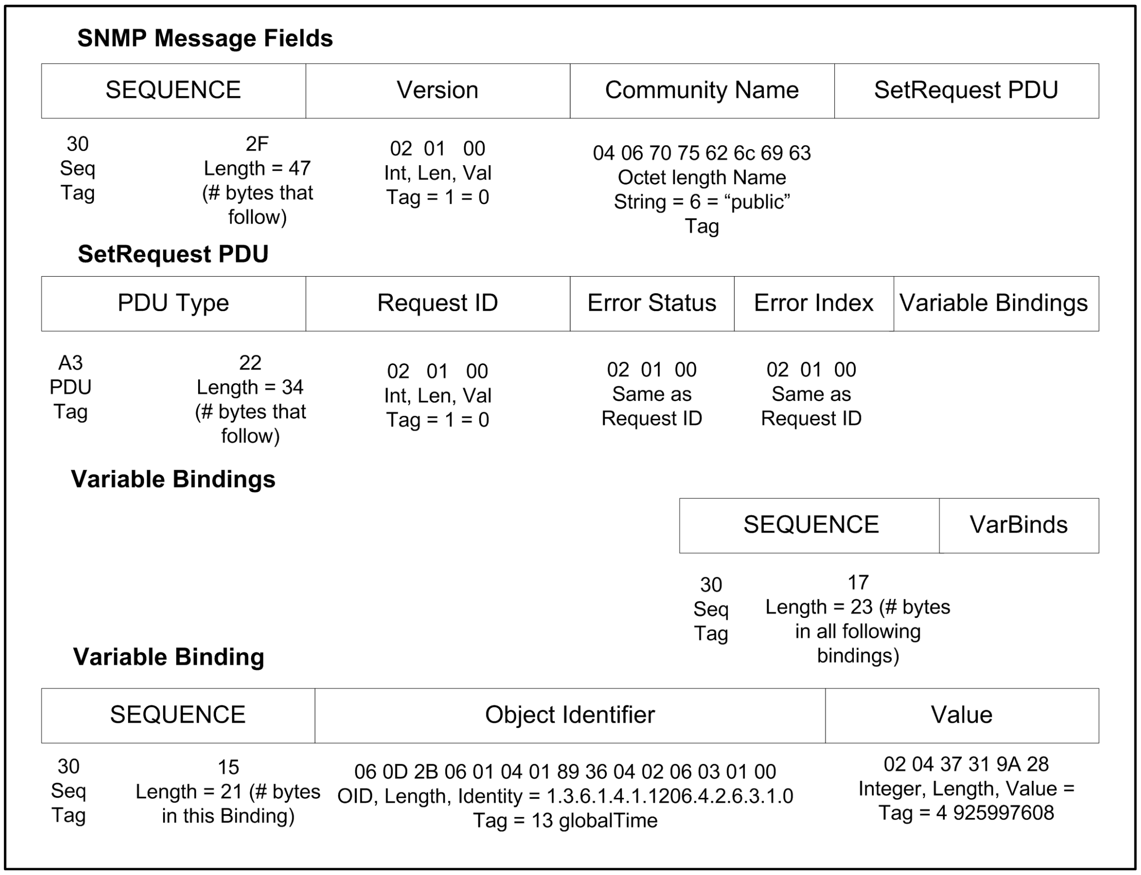

SNMP Message Fields
SEQUENCE
Version
Community Name
SetRequest PDU
30 Seq Tag
2F Length = 47 (# bytes that follow)
02 01 00 Int, Len, Val Tag = 1 = 0
04 06 70 75 62 6c 69 63 Octet length Name String = 6 = “public” Tag
SetRequest PDU
PDU Type
Request ID
Error Status
Error Index
Variable Bindings
A3 PDU Tag
22 Length = 34 (# bytes that follow)
02 01 00 Int, Len, Val Tag = 1 = 0
02 01 00 Same as Request ID
02 01 00 Same as Request ID
Variable Bindings
SEQUENCE
VarBinds
30 Seq Tag
17 Length = 23 (# bytes in all following bindings)
Variable Binding
SEQUENCE
Object Identifier
Value
30 Seq Tag
15 Length = 21 (# bytes in this Binding)
06 0D 2B 06 01 04 01 89 36 04 02 06 03 01 00 OID, Length, Identity = 1.3.6.1.4.1.1206.4.2.6.3.1.0 Tag = 13 globalTime
02 04 37 31 9A 28 Integer, Length, Value = Tag = 4 925997608

Figure 3.9: Set Operation Using SNMP Over UDP/IP/Ethernet

Table 3.4: SNMP Error codes

Error Code	**Description**
[00]	No error occurred
[01]	Response message too large to transport
[02]	The name of the requested object was not found
[03]	A data type in the request did not match the data type in the SNMP agent
[04]	The SNMP manager attempted to set a read-only parameter
[05]	General Error (some error other than the ones listed above)

field is also an Integer, which is set to 0x00 in the request sent by the SNMP manager. The SNMP agent places an error code in this field in the response message if an error occurred while processing the request. The list of error codes is shown in Table 3.4. If an error occurs, the

Error Index field holds a pointer to the Object that caused the error, otherwise the Error Index is (0x00).

Listing 3.5: Data structure for the GetRequest PDU

```
typedef struct {
//*******************SNMP MESSAGE FIELDS*************//
Byte seqTag;
Byte seqLen;
Byte versionTag;
Byte versionLen;
Byte versionVal;
Byte communityNameTag;
Byte communityNameLen;
Byte communityNameVal[6];

//*******************SET REQUEST PDU NUMBER**********//
Byte pduType;
Byte pduLen;
Byte requestIdTag;
Byte requestIdlen;
Byte requestIdVal;
Byte errorStatusTag;
Byte errorStatusLen;
Byte errorStatusVal;
Byte errorIndexTag;
Byte errorIndexLen;
Byte errorIndexVal;

//*******************VARIABLE BINDINGS***************//
Byte seqVarBinding;
Byte seqVarBindingLen;

//*******************BINDING***********************//
Byte seqBinding;
Byte seqLenOrId;
Byte varTag;
Byte varLen;
Byte varId[15];
Byte varValueTag;
Byte varValueLen;
} SnmpMsgPublicOidGet;
```

Listing 3.6: Data structure for the GetResponse PDU and SetRequest PDU

```
typedef struct {
//**************SNMP MESSAGE FIELDS****************//
Byte seqTag;
Byte seqLen;
Byte versionTag;
Byte versionLen;
Byte versionVal;
Byte communityNameTag;
Byte communityNameLen;
Byte communityNameVal[6];

//********SETREQUEST OR GETRESPONSE PDU NUMBER*****//
Byte pduType;
Byte pduLen;
Byte requestIdTag;
Byte requestIdlen;
Byte requestIdVal;
Byte errorStatusTag;
Byte errorStatusLen;
Byte errorStatusVal;
Byte errorIndexTag;
Byte errorIndexLen;
Byte errorIndexVal;

//*************VARIABLE BINDINGS*****************//
Byte seqVarBinding;
Byte seqVarBindingLen;

//******************BINDING*******************//
Byte seqBinding;
Byte seqLenOrId;
Byte varTag;
Byte varLen;
Byte varId[15];
Byte varValueTag;
Byte varValueLen;
Byte varValueVal;
} SnmpMsgPublicOidSizeOXOF;
```

The *Variable Bindings* field is the nested field that consists of a *SEQUENCE* and another nested field called *VarBinds*. The nested field *VarBinds* further contains a *SEQUENCE*, *Object Identifier* and a *Value*. This *Object Identifier* field points to a particular parameter in the SNMP agent. Finally, the *Value* field of the SetRequest PDU contains a value that is applied to the specified OID of the SNMP agent. However, this value could vary for other PDUs. For a GetRequest PDU, this value is set to NULL and acts as a placeholder for the return data, whereas for a GetRequest PDU, the value in this field represents the returned value from the specified OID of the SNMP agent.

Next, the data structures derived from the NTCIP 9001 Standard [27], which were explained earlier for the PDUs, are presented in Listing 3.5 and Listing 3.6. With reference to the standard NTCIP 9001 [27], there are two main data types defined. First, a 47 bytes *SnmpMsgPublicOid-Get* shown in Listing 3.5 and is mainly used by the GetRequest PDU. Similarly, a 48 bytes *SnmpMsgPublicOidSizeOXOF* is defined in Listing 3.6 and is used by either the GetResponse PDU or the SetRequest PDU. The first 47 bytes of the *SnmpMsgPublicOidSizeOXOF* are all identical to the bytes of the *SnmpMsgPublicOidGet*. One extra byte in *SnmpMsgPublicOid-SizeOXOF* is used to store values obtained by the GetResponse PDU or the SetRequest PDU via the SNMP agent.

Listing 3.7: SnmpMsgGet

```
union {
SnmpMsgPublicOidGet snmpMsg;
char buff[SNMP_MSG_SIZE_Get];
} SnmpMsgGet;
```

Listing 3.8: SnmpMsgPOXOF

```
union {
SnmpMsgPublicOidSizeOXOF snmpMsg;
char buff[SNMP_MSG_POID15_SIZE];
} SnmpMsgPOXOF;
```

Listing 3.9: SnmpMsgPOXOFresp

```
union {
SnmpMsgPublicOidSizeOXOF snmpMsgResp;
char buffer[SNMP_MSG_POID15_SIZE];
} SnmpMsgPOXOFresp;
```

In any application, memory management is one of the main concerns in order to run a program effectively and efficiently. The structures shown in Listing 3.5 and Listing 3.6 occupy a fixed size memory of 47 bytes and 48 bytes respectively. Such a huge memory usage can affect the performance of an application in the long run. The memory usage can be optimized to some extent by using the concept of the union data structure in the C programming language. It allows storing different data types in the same memory location. A union with many members can be defined, but only one member can contain a value at any given time, thereby providing an efficient way of using the same memory location for multiple purposes. The data structures shown in Listing 3.7, Listing 3.8, and Listing 3.9 use this concept of union to effectively handle memory usage. First, the union *SnmpMsgGet* shown in Listing 3.7 is used when the application is sending a GetRequest PDU to the traffic controller. This union uses the *SnmpMsgPublicOid-Get* structure from Listing 3.5. Next, the union *SnmpMsgPOXOFresp*, shown in Listing 3.9, is used while the application actually gets some reply from the traffic controller, i.e., the GetResponse PDU. Finally, the union *SnmpMsgPOXOF*, shown in Listing 3.8, is used to send a request to the traffic controller for changing some timing parameters in the controller, i.e., the SetRequest PDU. Note that the variable *SNMP_MSG_SIZE_GET* in Listing 3.7 is equivalent to *sizeof(SnmpMsgPublicOidGet)*, i.e., 47 bytes. Similarly, the variable *SNMP_MSG_POID15_-SIZE* in Listing 3.8 and Listing 3.9 is equivalent to *size(SnmpMsgPublicOidSizeOXOF)*, i.e., 48 bytes.

The last data structure that is introduced is for the Phase Status and is defined in Listing 3.10. This data structure keeps track of the status (1 for On and 0 for Off) of each color (Red, Yellow, and Green) of the traffic light at each phase (1-8) in the ETSS. It contains 3 member variables: *phaseNumber*, *phaseColor* and *phaseStat*. The variable *phaseNumber* can store values from 1 to a maximum number of phases (8 in our case). The variable *phaseColor* stores a character either R for Red, Y for Yellow, and G for Green. The variable *phaseStat* indicates if the selected color/signal in any phase is On (1) or off (0).

Listing 3.10: PhaseStatus

```
        typedef struct {
        int       phaseNumber;      // 1 to 8 (MAXPHASE)
        char      phaseColor;       // R, Y, G
        int       phaseStat;        // 1 for ON and 0 for OFF
        } PhaseStatus;
```

3.3.3 Important C Functions in ETSS

In this section, important functions used to send and get requests to and from the traffic controller are explained in more detail. This section includes the most important functions that are used to initiate communication. It also explains some support functions used to set values in the traffic controller as well as to get data from the traffic controller. Furthermore, it contains a number of functions that are required to extract phase related information from the traffic controller, such as, whether the green signal is on, or to get the current status of a signal in a particular phase. Each of those functions will be explained in detail next.

Controller Access Utility Functions: Establishing a connection is the first step to perform in many types of communication. Consider a client-server configuration, where the traffic controller is the server, and the computer or RSU that requests data from the server is the client. The server and client communicate via a socket. A socket is an endpoint of a two-way communication link between two programs running on a network. It is bound to a port number so that the TCP layer can identify the application that the data is destined to be sent to. In general, an IP address along with a port number form a socket. Figure 3.10 shows the client-server relation with sockets.

In a client-server communication, a client initiates the communication process, whereas the server responds to incoming client requests. It should be noted that a server typically starts before the client does, and waits for the client to request connections. Figure 3.11 shows the

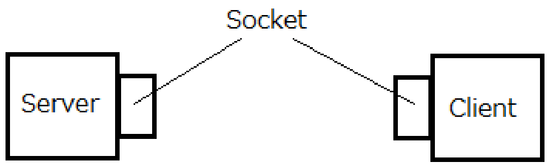


Figure 3.10: Socket Concept

interaction between a server and a client as a flowchart, where every block in the flowchart has an important role in the network connection. First, a server and client create a socket S. Then the server binds the socket S to a local address, which is optional for a client. The server then listens to the incoming connection requests from clients to alert the TCP/IP machine of the willingness to accept connections. The client now connects socket S to a foreign host. By this time, the server accepts the connection and receives a second socket NS. The server uses this new socket NS, whereas the client uses socket S to read and write data by using sendto() and recv() calls until all data has been exchanged. Finally, the server closes socket NS, the client closes socket S and ends the TCP/IP session. The server again accepts a connection if there are any pending connection requests from the client and this phenomenon continues.

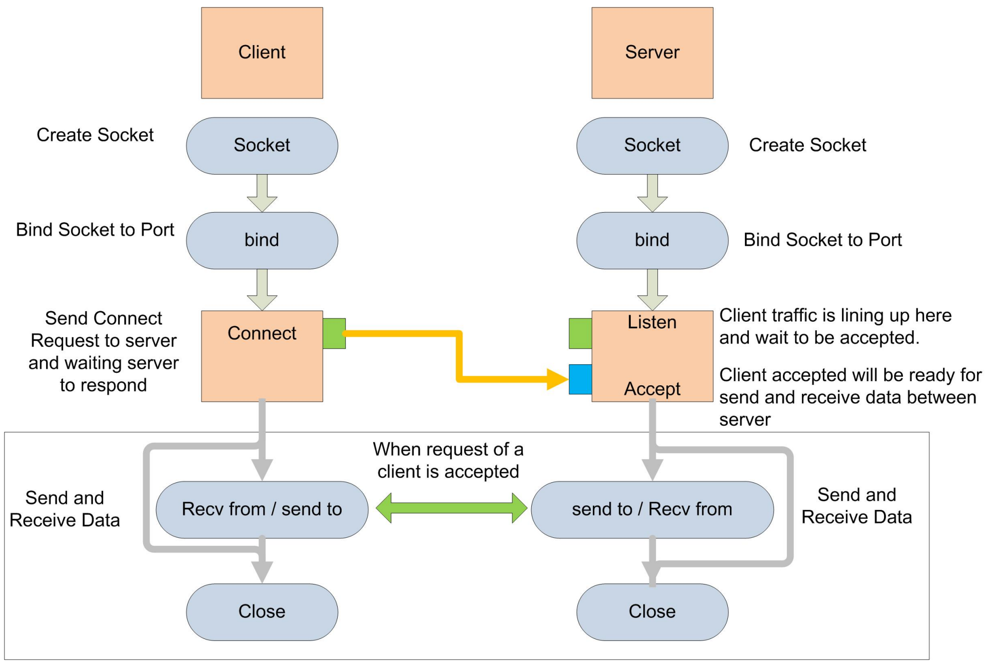

Client
Server
Create Socket
Socket
Socket
Create Socket
Bind Socket to Port
bind
bind
Bind Socket to Port
Send Connect Request to server and waiting server to respond
Connect
Listen
Accept
Client traffic is lining up here and wait to be accepted.
Client accepted will be ready for send and receive data between server
When request of a client is accepted
Send and Receive Data
Recv from / send to
send to / Recv from
Send and Receive Data
Close
Close

Figure 3.11: Flowchart showing Socket Connection

Keeping these concepts of client-server communication in mind, a connection manager utility has been created in the ETSS. Shown in Listing 3.11 are some important functions that are used to establish the communication channel between a server and a client, which will be explained next.

Listing 3.11: Functions for accessing Traffic Controller

```c
void dsc_init_global_vars(void);
void dsc_init(void);
Byte tc_get(const unsigned char * oid, Byte phase);
Byte tc_get_groups(const unsigned char * oid);
Byte tc_set(const unsigned char * oid, Byte phase, Byte value);
int  getValue();
```

1. ***void dsc_init_global_vars(void)***: The main purpose of this function is to initialize some global variables. It initializes two socket status flags to keep track of the message sent to the traffic controller and a message received from the traffic controller during the communication. This function also allocates 48 bytes for storing the SNMP GetResponse PDU or/and SNMP SetRequest PDU information and 47 bytes for storing SNMP GetRequest PDU information.

2. ***void dsc_init(void)***: This function is mainly responsible for creating a new socket. It initializes all the storage elements by calling function *dsc_init_global_vars().* This function requires the IP address and the port number of the traffic controller.

3. ***Byte tc_get(const unsigned char * oid, Byte phase)***: The first parameter in this function is the object identifier of the traffic controller parameter whose value needs to be extracted. The second parameter is the phase number (1 to 8) whose information needs to be queried. This function returns 1 if the traffic controller fails to provide data and returns 2 otherwise.

4. ***Byte tc_get_groups(const unsigned char * oid)***: Using function *tc_get*() we can get phase information about a specific phase. However, function *tc_get_groups*() is used to extract information about all phases at once. This function works only for certain object identifiers.

5. ***Byte tc_set(const unsigned char * oid, Byte phase, Byte value)***: This function takes three parameters, i.e., the object identifier of the traffic controller parameter to set, the phase number (1 to 8), and a value to be written to the traffic controller. This function returns 1 if the traffic controller failed to write data and returns 2 if it succeeded.

6. ***int getValue()***: Using the functions *tc_get*() and *tc_get_groups*() one can obtain data from the data structure *SnmpMsgPublicOidSizeOXOF*. It is obtained from the *varValueVal* field of *SnmpMsgPublicOidSizeOXOF*.

Data Access Utility Functions: The functions provided in Listing 3.12 are used by the application to query for a particular color of light in a specific phase at the intersection.

Listing 3.12: Functions Providing Phase Information

```
int  isGreenLightOn(int phase);
int  queryForLight(int phase, char color) ;
int  isLightOn(int phaseNum, char color) ;
int  *decimalToBinary(int num) ;
void getPhaseInformation(int *x, char color);
```

1. ***int isGreenLightOn(int phase)***: This function queries the status of green signal in a particular phase that is provided as an argument to the function. It returns *TRUE* if the signal at the given phase is green and returns *FALSE* otherwise.

2. ***int queryForLight(int phase, char color)***: This function takes two arguments. The first argument is the phase number and the second argument is the signal for which this function wants to get the status. The function checks if the Green (G), Yellow (Y), or Red (R) signal is active at a given phase.

3. ***int isLightOn(int phaseNum, char color)***: This function is called by *queryForLight*(). It contains the main logic to check if any of the signals (G, R, or Y) is active in a particular phase. The phase number is sent as the first argument of this function, whereas the signal color is sent as the second parameter to this function.

4. ***int *decimalToBinary(int num)***: This function converts an integer into a binary number. Every bit of this stream of the converted binary number is later stored in an array of integers.

5. ***void getPhaseInformation(int *x, char color)***: This function gathers information about the signal status of each phase. Data Structure mentioned in Listing 3.10 is used to store information about each signal status.

3.3.4 ETSS Algorithm

The ETSS considers heavy loaded vehicles only, but could be easily modified to include other vehicles types. This application allows a heavy loaded vehicle to pass the intersection by extending a green traffic signal. However, the vehicle has to satisfy certain conditions to qualify for an extension.

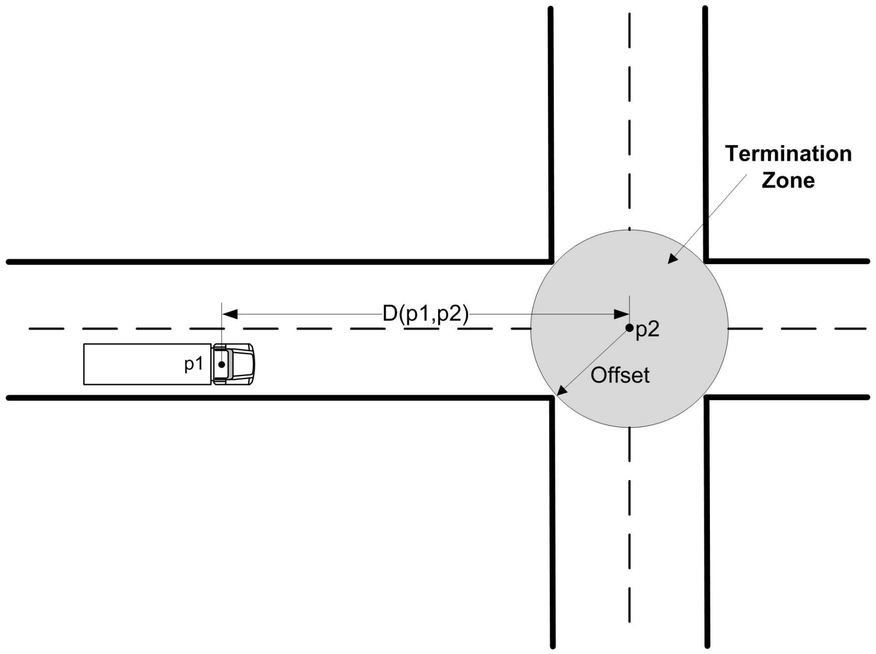


Figure 3.12: A vehicle approaching an intersection in ETSS

The ETSS will be described using Figure 3.12. In the figure, an intersection is shown with the RV at point p1 approaching the intersection. The RSU, which is mounted somewhere near the intersection is initialized with the GPS coordinates representing the center of the intersection, which is point p2. The shaded area bounded by a circle indicates the termination zone with the center being point p2. The distance from p2 to the outbound of this area is denoted by *offset*. This offset is later used to cancel the granted extension request. The distance from the RV to the center of the intersection is calculated from the GPS information. However, the distance alone is not sufficient to grant the extension, as it depends on the speed as well. Based on the current speed and the distance, the projected time for a vehicle to reach the intersection is calculated. Several cases will be differentiated. First, if the current speed is higher than the predefined speed threshold and the projected time to reach the intersection is less than the time threshold, then an extension can be granted for the target vehicle. However, if the speed is too low and the distance to traverse takes too much time, then the extension will not be granted. Finally, if the speed and time are sufficient, but later on the application determines the vehicle slowed down so that the thresholds are violated, the extension is revoked and the request is terminated by not sending any more extension requests to the controller.

Recall the offset of the intersection. When the vehicle is determined to be in this offset region, the hold is terminated as it indicates the vehicle is in the process of passing.

Conditions that need to be met so that the ETSS triggers an extension are summarized below.

1. The vehicle should be heavy loaded.
2. The heavy loaded vehicle should travel at a speed greater than a predefined threshold speed.
3. The heavy loaded vehicle should reach the intersection within a predefined threshold time.
4. The hold request implementing the extension applies only if the signal is Green in that instance.
5. Once the intersection is reached, a hold request should get terminated.
6. The extension shall not be granted beyond maximum green (MAX GRN1).

The flowchart in Figure 3.13 shows how a vehicle gets qualified for the extension. The logic of the flowchart is implemented in Algorithm 4. The algorithm indicates that it needs to determine if the target vehicle type is correct. This can be done by querying if the vehicle type is in the range from 7 to 15. In Line 7 of the algorithm, *VehicleType_axleCnt2* = 7 and *VehicleType_axleCnt7MultiTrailer* = 15 represent the vehicle types and the values within this range signifying heavy loaded vehicles. A complete set of vehicle types was shown in Table 3.1. If the vehicle type of the RV falls in this range, then the variable *isHeavyLoad* defined in Line 2 of this algorithm, is set to *TRUE*. The rest of the algorithm executes only if the vehicle type of the RV qualifies as a heavy load. Once the ETSS verifies the message from the heavy load, it checks if the target vehicle qualifies for an extension based on various parameters explained earlier. The logic to determine if the RV qualifies for an extension, i.e., *extensionDecisionLogic*, is presented in Algorithm 5. The status of the green light is queried only if the vehicle qualifies for an extension. The main reason for this approach is to avoid unnecessary communication between the RSU and the traffic controller. Finally, if all the extension qualifications are met and the current status of green signal is ON, then a request to hold the current green signal is sent to the traffic controller.

The aforementioned logic that determines if the target vehicle qualifies for an extension is implemented in Algorithm 5. Three main parameters that should be considered for the

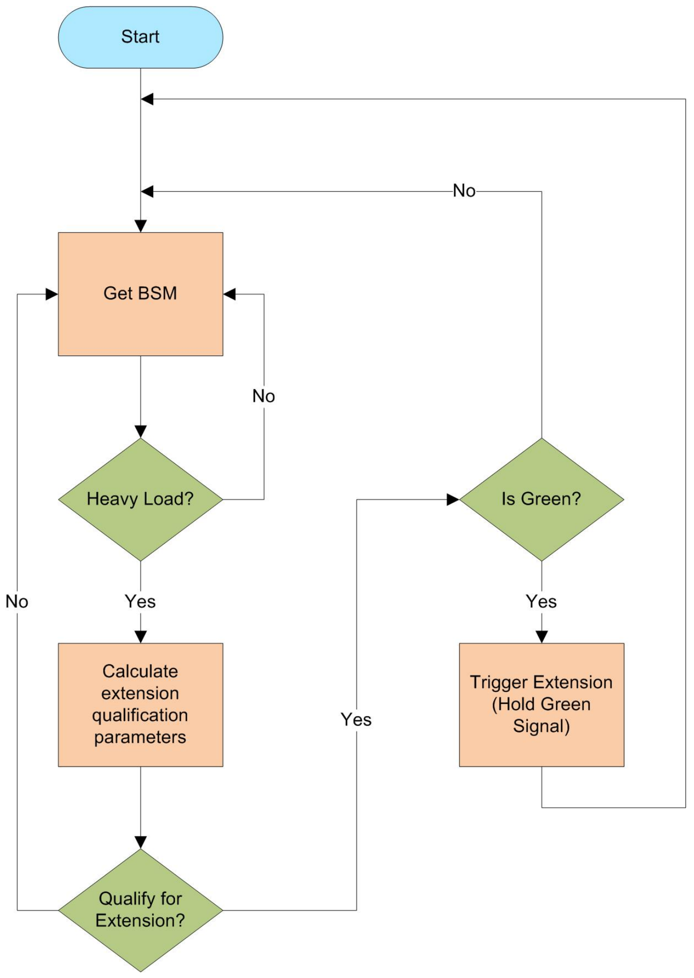

Start
Get BSM
Heavy Load?
No
Yes
Calculate extension qualification parameters
Qualify for Extension?
No
Yes
Is Green?
No
Yes
Trigger Extension (Hold Green Signal)

Figure 3.13: Flowchart for Eco-Traffic Signal Application

extension qualification are Speed (S), Time (T) and Distance (D). First, the vehicle speed S in mph should be greater than some predefined threshold speed. This criterion is set to avoid granting an extension to slow moving vehicles, under the assumption that they will benefit less in terms of energy savings from an avoided stop. Another qualification parameter is the time T measured in seconds. Since the RSU can get BSMs from vehicles far away from the intersection, only those target vehicles qualify, which are projected to reach the intersection within some pre-configured threshold time. Even if the vehicle is beyond the threshold speed, if it takes a longer time to reach the intersection, or if the fast moving vehicle slows down below the threshold, the ETSS will not qualify the target vehicles for an extension.

The time T to reach the intersection is the function of speed S and distance D, i.e., $T = \frac{D}{S}$. The distance between the vehicle at point $p1$ and the center of an intersection at point $p2$ is

Algorithm 4 Eco Traffic Application Algorithm

```
1: procedure ECOTRAFFICLOGIC()
2:     $isHeavyLoad \leftarrow UNDEFINED$
3:     $qualifyForExtension \leftarrow FALSE$
4:
5:     if $count > 0$ then
6:
7:         if $7 \leq (bsm \rightarrow vehicleType) \leq 15$ then
8:             $isHeavyLoad \leftarrow TRUE$
9:
10:        if $isHeavyLoad \leftarrow TRUE$ then
11:            $qualifyForExtension \leftarrow extensionDecisionLogic()$
12:
13:            if $qualifyForExtension \leftarrow TRUE$ $and$ $isGreenOn \leftarrow TRUE$ then
14:                $Send\ Green\ Hold\ Request\ to\ Controller$
15:                $Print$ “ $Extension\ Qualified$ ”
16:            else
17:                $Print$ “ $Extension\ NOT\ Qualified$ ”
```

Algorithm 5 Extension Decision Logic Algorithm

```
1: procedure EXTENSIONDECISIONLOGIC()
2:     $distance, velocity, time \leftarrow 0.0$
3:     $currentOBUPosition \leftarrow (bsm \rightarrow longitude), (bsm \rightarrow latitude)$
4:     $speed \leftarrow (bsm \rightarrow speed)$
5:
6:     $velocity \leftarrow getSpeedInMilesPerHour(speed)$
7:     $distance \leftarrow getDistanceInMiles(rsuPosition, currentOBUPosition)$
8:
9:     if $0 \leq distance \leq OFFSET$ then
10:        $return\ FALSE$
11:
12:    if $velocity! = 0.0$ then
13:        $time \leftarrow (distance/velocity) * 3600.0$
14:    else
15:        $time \leftarrow 100000000$
16:        $return\ FALSE$
17:
18:    if $(velocity < THRESHOLD\_SPEED)$ OR $(time > THRESHOLD\_TIME)$ then
19:        $return\ FALSE$
20:    else
21:        $return\ TRUE$
```

denoted by $D_{p1,p2}$. This distance is calculated based on the longitude and latitude, which are given either in radian or in degrees and can be expressed by the following formula:

$D_{p1,p2} = ACOS[SIN(\theta_{lat1}){*}SIN(\theta_{lat2}){+}COS(\theta_{lat1}){*}COS(\theta_{lat2}){*}COS(\alpha_{lon2}{-}\alpha_{lon1})]{*}6371000$
In ETSS, the longitude and latitude obtained are in degrees, therefore

$\theta_{lat1} = lat1{*}\pi/180,$

$\theta_{lat2} = lat2{*}\pi/180,$

$\alpha_{lon1} = lon1{*}\pi/180,$

$\alpha_{lon2} = lon2{*}\pi/180,$

where $lat1$ and $long1$ are the latitude and the longitude of p1 of the vehicle, and $lat2$ and $long2$ are the latitude and the longitude of p2 at the center of the intersection.
However, if the longitude and latitude are obtained in radians, the values could directly be obtained as follows:

$\theta_{lat1} = lat1,$

$\theta_{lat2} = lat2,$

$\alpha_{lon1} = lon1,$

$\alpha_{lon2} = lon2.$

Once the extension is granted, it needs to be terminated if the vehicle reached the intersection. For this reason, the term *Offset* was introduced to define the Termination Zone. When the qualified target vehicle reaches this zone, the traffic controller should terminate the extension, signaling that this vehicle is in the process of passing the intersection. The green signal hold can be terminated in two ways. First, if no more signal hold command requests are sent to the traffic controller, the existing hold automatically gets terminated after a brief duration of time. Otherwise, it can be terminated forcefully. However, ETSS will not forcefully terminate the green signal hold. If the vehicle reaches the Termination Zone, it simply does not get qualified for an extension.

3.4 Arada Locomate Application

Arada Systems provides the default application *getwbsstxrxencdec* for its locomate OBUs and RSUs, that is used to transmit and receive BSMs. This integrated application can be configured with different parameters and supports different message sets. However, one first has to log into the Arada devices in order to start any application. This can be done by using the default username and password provided by the device manufacturer. In a real systems these defaults should be changed however according to the security policies in place. A simple example of the usage of the application is shown below.

Transmit BSM: After a user logs in onto the device, the Command Line Interface (CLI) is used to execute commands. To transmit the BSMs, the following sample command can be used:

getwbsstxrxencdec -s 172 -t BSM -o norx -a 0

This command registers and starts a provider, which sends the BSMs. The option *-s* signifies using a service channel. This option is followed by the channel number used for transmitting the message, in this case CH172. Similarly, option *-t* represents the type of message, which in the above example signifies the message to be a BSM. Other message types that are supported are: Prove Vehicle Data (PDV), Road Side Alert (RSA), Intersection Collision Alert (ICA), Signal Phase And Timing (SPAT), Map Data (MAP), and Traveler Information Message (TIM). Option *-o* is used to indicate the transmission mode. This option is followed by *norx*, which means this command only transmits the message but does not receive anything from other transceivers. Finally, option *-a* is used to identify the service channel access mode. The value 1 represents alternating mode, whereas 0 represents a continuous mode.

Receive BSM: The procedure for executing a command to receive BSMs is similar to that of transmitting a BSM. However, the arguments for the command are different in this case. For receiving the BSMs, the following command can be used:

getwbsstxrxencdec -w User -s 172 -o notx -u 2 -x 1

The command *getwbsstxrxencdec* is followed by the option *-w*, which represents a service type. A service type could be either a User or a Provider. A provider transmits the BSMs, whereas a user receives the BSMs. The next option *-s 172* is similar to the option used by the BSM transmitter and indicates the user is sensing BSMs in channel 172. Furthermore, option *-o notx* indicate the user is not transmitting any message but is sensing BSMs. User option *-u* represents a user request type. The value for this option is set to 2, which means that the user request type is unconditional. Finally, the option *-x* is used by the user, which is similar to the option *-a* used by the provider. In this case, the parameter for *-x 1* represents continuous extended access.

A complete set of parameter options with their default values is given in Appendix E.

3.5 Multi-threaded ETSS Application

The application *getwbsstxrxencdec* provided by Arada Systems contains many useful features for V2V and V2I communication. The ETSS is not limited only to V2I communication, but needs to communicate with the Traffic Controller. To facilitate communication between an RSU and a traffic controller, a new custom application has been designed.

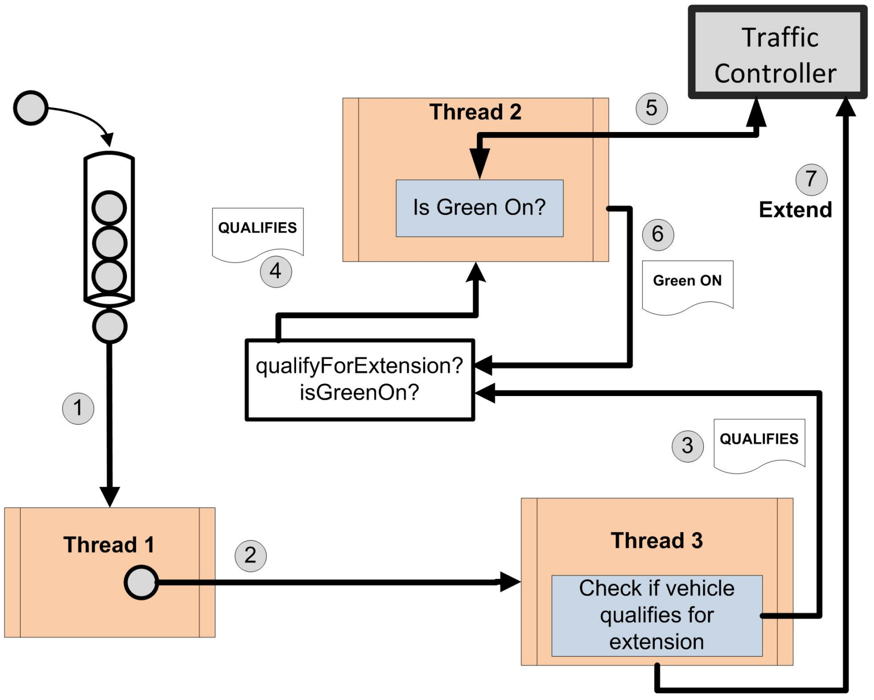


Figure 3.14: Multi-threaded Application

The ETSS is a multi-threaded application and consists of three threads, as shown in Figure 3.14. The responsibility of the first thread, i.e., Thread 1, is to receive BSMs from vehicles, if there are any in the surrounding. The second thread, i.e., Thread 2, communicates with the traffic controller to check if the green signal is active in a particular phase that ETSS is configured to monitor. The third thread, i.e., Thread 3, identifies if the approaching vehicle qualifies for an extension. Thus, it makes the decision if an extension should be granted. Furthermore, if the extension is granted for the approaching vehicle, this thread sends a command to the traffic controller in order to hold the current green signal.

As can be seen in Figure 3.14, the application uses two boolean flags: *isGreenOn* for checking the green status and *qualifyForExtension* for storing information if the vehicle qualifies for an extension. The flags can be accessed by Thread 2 and Thread 3. First, Thread 1 receives a BSM from a pool of BSMs, which is forwarded to Thread 3 in order to verify if the vehicle qualifies for an extension based on its speed and current geographical position. If the vehicle qualifies, it updates the *qualifyForExtension* variable. When Thread 2 gets notified of this extension qualification, it queries the traffic controller about the status of the green light. If the green light is on, Thread 2 updates the *isGreenOn* variable and Thread 3 gets notified. Finally, after getting notified that the green signal is on, Thread 3 sends the green signal hold command to the traffic controller.

The exact detail of the RSU and controller communication is given in Appendix A.

Chapter 4

Experimental Validation of ETSS

The ETSS was tested in the field. Due to liability reasons, the tests could not be conducted using real traffic lights in the Idaho Transportation Department's traffic system. Therefore, an intersection was emulated on the University of Idaho campus. Figure 4.1 shows a map of the field test area, where the blue dotted line indicates the path of the test vehicle on 7th Street, with the intersection positioned at the west end of the test area. The red marker indicates the center of the intersection, in front of the Engineering Physics Building (EPB). The GPS coordinates of the red marker in the figure were also the reference point for the RSU, which was mounted on the traffic controller cabinet installed on the concrete platform between the marker and the EP building to the north.

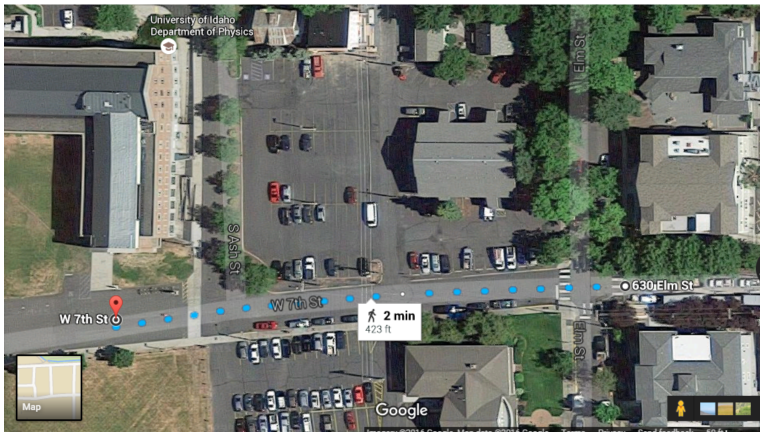


Figure 4.1: Location for Experiment Setup

The following is the list of equipment used in the experiment:

- One Econolite Traffic Controller, Model ASC-3/2100, and all supporting hardware installed in the traffic controller cabinet.
- One RSU (Arada Locomate), mounted on top of the traffic controller cabinet.
- One OBU (Arada Locomate) installed in the test vehicle emulating the heavy loaded vehicle.
- One laptop to observe the activities of the RSU and one laptop in the test vehicle to observe the functions in the OBU.

- Traffic Lights for a simple intersection connected to the Traffic Controller.

The test parameters for the RSU are summarized in Table 4.1. The phase indicating the street traveled by the test vehicle was phase 1. Whereas the RSU was positioned next to the intersection, the GPS latitude and longitude coordinates used by the application are the coordinates of the center of the intersection, i.e., those of the red marker in Figure 4.1. For the demonstration of the ETSS, the speed threshold was set to 5 mph. This speed was intentionally set so low, as the intersection was positioned in the campus walkway area. Accordingly, the time threshold was set to 20 seconds, given this slow speed threshold. The offset, which determines the radius of the termination zone from the center of the intersection was set to 5 meters.

Table 4.1: RSU Field Test Configuration

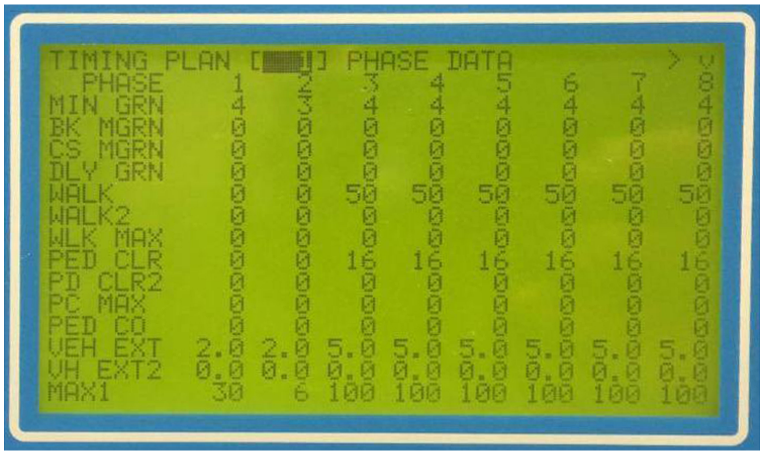
Parameters	**Value**
Phase Number	1
Latitude	46.72888
Longitude	-117.01
Speed Threshold	5 miles per hour
Time Threshold	20 seconds
Offset	5 meters

The traffic controller timing parameters had to be adjusted for the experiment. Figure 4.2 shows the timing plan, which was configured in the traffic controller. Whereas the figure shows information for eight phases, only phase 1 and 2 were used in the experiment, representing the test vehicle's street and the crossing side street respectively. As can be seen in the figure, the minimum green times (MIN GRN) for phase 1 and 2 were set to 4 and 3 seconds respectively. These short times were used for reasons of practicality for the experiment only. The maximum green times (MAX1) were also specified.

4.1 Experiment Details

Several experiments were conducted to verify the ETSS functionalities. In all cases, the test vehicle with the installed OBU was configured to represent a heavy load. There are three scenarios that needed to be verified as explained in the experiments below.

A) **Experiment 1 - Vehicle qualifies for an extension:** In this experiment, the test vehicle approached the intersection at a speed higher than the minimum threshold. Recall



```
TIMING PLAN [███1] PHASE DATA          > v
  PHASE      1     2     3     4     5     6     7     8
MIN GRN      4     3     4     4     4     4     4     4
BK MGRN      0     0     0     0     0     0     0     0
CS MGRN      0     0     0     0     0     0     0     0
DLY GRN      0     0     0     0     0     0     0     0
WALK         0     0    50    50    50    50    50    50
WALK2        0     0     0     0     0     0     0     0
WLK MAX      0     0     0     0     0     0     0     0
PED CLR      0     0    16    16    16    16    16    16
PD CLR2      0     0     0     0     0     0     0     0
PC MAX       0     0     0     0     0     0     0     0
PED CO       0     0     0     0     0     0     0     0
VEH EXT    2.0   2.0   5.0   5.0   5.0   5.0   5.0   5.0
VH EXT2    0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0
MAX1        30     6   100   100   100   100   100   100
```

Figure 4.2: Traffic Controller - Timing plan

that vehicles only qualify for a possible extension if their projected time to reach the intersection is less than the time threshold, in our case 20 seconds, and the speed of the vehicle is greater than the speed threshold, which is 5 mph in our case. As soon as the green light of phase 1, corresponding to the street of the test vehicle, was on, the vehicle started accelerating toward the intersection. The starting point of the vehicle was selected so that it would not be able to reach the intersection within MIN GRN. Thus, without the ETSS the vehicle would have needed to stop on red.

Expected Outcome: After MIN GRN (4 seconds), the green light should get extended until the test vehicle reaches the intersection, allowing it to continue without stopping. After the vehicle crosses the intersection, the ETSS should terminate the green extension.

B) **Experiment 2 - Vehicle qualifies first, but gets disqualified later:** Similar to the previous experiment, the test vehicle approached the intersection. But this time, in the beginning, the vehicle approached the intersection at a speed higher than the minimum threshold, while satisfying the time threshold. However, this higher speed was later dropped below the minimum threshold before reaching the intersection. This represents a scenario, where a fast-moving test vehicle has to slow down before reaching the intersection due to some circumstance, e.g., a vehicle in front moving slowly.

Expected Outcome: An extension should be granted while the vehicle qualifies, given the high speed, but when the vehicle slows down below the threshold, the extension should get revoked.

C) **Experiment 3 - Vehicle does not qualify for an extension:** In this experiment the test vehicle approached the intersection at a speed lower than the minimum threshold. Similar to previous experiments, as soon as the green light of phase 1 turned on the vehicle started moving slowly towards the intersection. The ETSS should not qualify this test vehicle for an extension and let the traffic signal continue its normal cycle.

   Expected Outcome: In this experiment the test vehicle should not be granted an extension and this would not affect regular flow of the traffic cycle. Recall that the minimum speed threshold was introduced to avoid extensions for vehicles with low energy saving for avoided stops.

The summary of the expected output of the experiments above are shown in Table 4.2.

Table 4.2: Expected output from the experiment

Experiment Number	**Exp 1**	**Exp 2**	**Exp 3**
Green signal extended?	Y	Y	N
Extension revoked before reaching the intersection?	N	Y	-

Two scenarios needed to be verified for each of the experiments. First, it should be verified if the green signal was extended. Secondly, if the green signal was extended, it should be validated if at any point in the experiment the extension got revoked.

4.2 Results

The results for three interesting scenarios from the field experiments are shown in Figure 4.3. The figures represent scenarios in which all BSM messages were captured and processed in real-time to determine if extensions should be granted. The BSM indices are shown on the x-axis of the plots, for 125 messages, which covers approximate 12.5 seconds for the experiment. The y-axis shows the speeds that were extracted from the relevant BSMs. The speed is indicated as a solid plot. Of interest is the dotted plot, which identifies when an extension is granted or revoked. A value of 1 indicates that an extension is granted, and 0 implies that it is not granted or revoked. The red horizontal line represents the speed threshold, which was set to 5 mph.

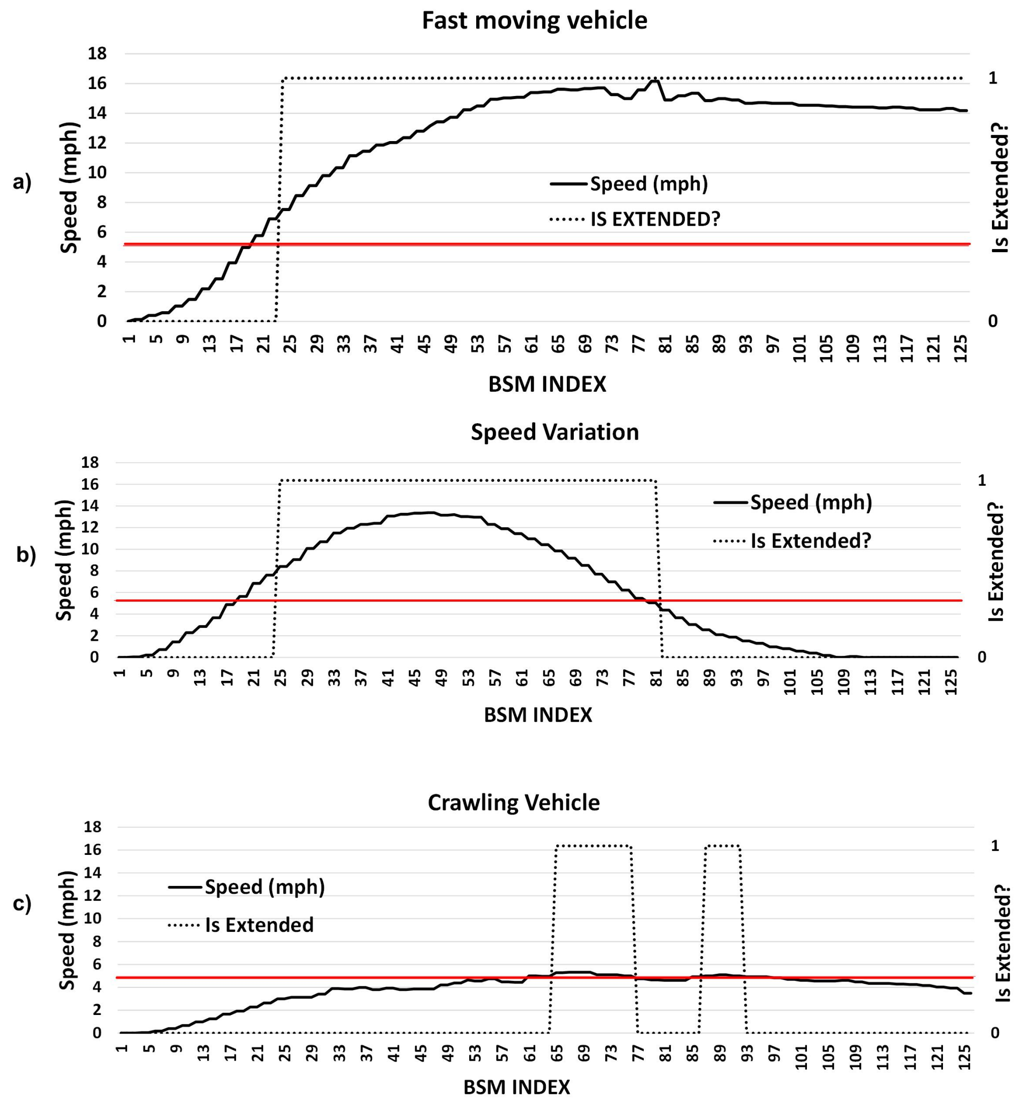


Figure 4.3: Experiment Results for Different Scenarios

The plot in Figure 4.3a) refers to the first experiment. When the vehicle's speed surpassed the predetermined speed threshold and the projected time to reach the intersection became lower than the predefined time threshold, the ETSS requested an extension from the traffic controller. It can be seen that the speed and time threshold constraints were met when the 22^{nd} BSM, i.e., BSM_{22}, was received.

The plot in Figure 4.3b) depicts the scenario when the vehicle first traveled at a speed higher than the threshold. The traffic controller granted the extension for the time interval corresponding to BSMs from BSM_{24} to BSM_{81}, when the speed and time threshold constraints were met. Later, the test vehicle reduced its speed below the threshold, which caused the granted extension to be revoked.

Finally, a scenario is shown, which was originally intended to demonstrate that no extension would be granted when the speed threshold was not met. However, the interesting result shown in Figure 4.3c) prompted us to keep this case where the threshold was not surpassed initially, but where the final speed oscillated around the threshold. Specifically, due to the fact that a speed of below 5 mph was difficult to maintain, the vehicle's speed occasionally rose above and consequently fell below the threshold. Even in this situation did the ETSS maintain its correct function, by revoking the extension each time when the vehicle slowed below the threshold. The functionality of not granting an extension if the speed threshold was not met is verified by the behavior during the first 64 BSMs, i.e., no extension was granted during that period.

It should be noted that in the current implementation of the ETSS extensions are initiated by sending signal hold commands to the controller. Each qualifying BSM causes such command to be sent. However, sequences of hold commands are non-cumulative, i.e., a new hold command overrides the hold timing for the previous hold command.

Chapter 5

Conclusion and Future Work

The thesis described an eco-traffic signal system application, where connected vehicle technology is used to avoid unnecessary stops of heavy vehicles at signalized intersections in order to conserve fuel. The work is a proof of concept that demonstrates the usage of V2I communication to control the fixed signaling infrastructure in an adaptive way. It gives testimony that the new technologies of connected vehicles can be used in applications with legacy traffic controllers in the intersections. Specifically, it was shown that BSM information from OBUs of the target vehicles can be used to communicate important information to the RSU, positioned in an intersection. The speed and distance information in the BSM, together with the GPS coordinates of the vehicle, also encapsulated in the BSM, and the center of an intersection, can be used to calculate if the green period should be extended to allow the vehicle to pass, thereby avoiding stops with the associated energy loss. An algorithm was presented that determined that a heavy vehicle qualifies for an extension of the green period if it satisfied certain speed and distance requirements. This algorithm also revokes extensions from vehicles should they fall short of maintaining their qualifying parameters. The communication framework for allowing the RSU to send commands to the NTCIP compliant traffic controllers were also developed. A mechanism to extend the green light was implemented using the object identifier phaseControlGroupHold. The extension is revoked when the target vehicle enters the termination zone in the intersection.

The architecture of the eco-traffic system application has been specified and implemented. To validate the functionality, field tests with commercially available DSRC equipment, in this case, Arada Systems Locomate OBU and RSU were conducted. The field tests, which used a fully equipped signalized intersection, including the traffic cabinet with a controller, support hardware and traffic lights, allowed testing different scenarios of the ETSS.

Several aspects of this project could be investigated in future work. It would be interesting to determine the computational impact such application has on the RSU. This may be most important if the traffic density is high, as computations would appear to scale linear with the number of vehicles in the RSU's reception range. Such overhead investigation should also consider computations associated with features necessary for real implementations. Specifically, the ETSS in its current form is a proof of concept only and lacks features needed for real implementations. First, currently only one heavy loaded vehicle is considered. However, in real traffic multiple vehicles need to be considered. This however will require that the RSU manages all qualifying vehicles in a way that will not cause potential monopolizing of the phase when many such vehicles approach. A mechanism needs to be derived that will comply with the maximum green time of the traffic controller. Secondly, in the current implementation, the direction of the vehicle is not considered. Incorporating the heading of a vehicle will allow for different phases to be considered concurrently. In addition, the vehicle's direction can be used to terminate the green extensions after it has reached the intersection. Currently the sign of the two distances is used to differentiate approaching from departing vehicles. Thirdly, the elevation information of the BSM is currently not used by the application. Including elevation may further refine the accuracy of the vehicle positions. This will be needed to resolve situations where bridges or overpasses are close to or part of the intersection.

There were several important lessons learned, which may be of interest to system implementors. Most importantly, the lack of detailed information for the DSRC equipment and the communication specifications for communicating with the traffic controllers was challenging in the development of solutions. The term “bleeding edge” technology came to mind at many times. The second important lesson was that field testing is very time and resource consuming. A fair number of personnel was necessary to prepare field tests, get vehicles ready and verify that all equipment was fully functional, all under the assumption that the weather would cooperate, which was not always a given.

Bibliography

[1] United States Department of Transportation, Office of the Assistant Secretary for Research and Technology, Intelligent Transportation Systems Joint Program Office, http://www.its.dot.gov/

[2] C. Anagnostopoulos, I. Anagnostopoulos, V. Loumos and E. Kayafas, "A license plate-recognition algorithm for intelligent transportation system applications", *IEEE Trans. Intell. Transp. Syst.* , vol. 7, no. 3, pp.377 -392, 2006.

[3] Ezell, S., "Explaining International IT Application Leadership: Intelligent Transportation Systems", *ITIF - The Information Technology & Information Foundation, Washington, USA*, 2010.

[4] *Traffic Safety Facts 2014*. Washington, DC: National Highway Traffic Safety Administration; 20015, http://www-nrd.nhtsa.dot.gov/Pubs/812219.pdf.

[5] N. Daiheng, "Traffic Flow Theory: Characteristics, Experimental Methods, and Numerical Techniques", November 9, 2015.

[6] M. K. Nasir, R. Md Noor, M. A. Kalam, and B. M. Masum, "Reduction of fuel consumption and exhaust pollutant using intelligent transport systems", *The Scientific World Journal*, vol. 2014, Article ID 836375, 13 pages, 2014.

[7] H. Omar, W. Zhuang and L. Li, *VeMAC: A TDMA-based MAC protocol for reliable broadcast in VANETs*, IEEE Trans. Mobile Comput., vol. 12, no. 9, pp.1724 -1736 2013.

[8] J. Petit and Z. Mammeri, "Authentication and consensus overhead in vehicular ad hoc networks", *Telecommun. Syst.*, vol. 52, no. 4, 2013

[9] R. Bauza, J. Gozalvez, and J. Sanchez-Soriano., Road traffic congestion detection through cooperative vehicle-to-vehicle communications, *In Proc. IEEE Conf. on Local Computer Networks*, pages 606-612, 2010.

[10] Chang, J., et al. *Estimated Benefits of Connected Vehicle Applications: Dynamic Mobility Applications, AERIS, V2I Safety, and Road Weather Management Applications.* No. FHWA-JPO-15-255. 2015.

[11] H. Alturkostani, A. Chitrakar, R. Rinker and A. Krings, “On the Design of Jamming-Aware Safety Applications in VANETs”, in Proc. 10th Cyber and Information Security Research Conference, (CISR 2015), Oak Ridge, Tennessee, April 7-9, 2015.

[12] *IEEE 802.11, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications*, IEEE Std., June 2007.

[13] *IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 6: Wireless Access in Vehicular Environments*, IEEE Std 802.11p, 2010.

[14] “WiMAX^{TM} operators and vendors from around the world announce new deployments, growing commitments at the 2nd Annual Annual WiMAX Forum® Global Congress”. *New release* (WiMAX Forum). IEEE Std 802.16^{TM}, June 4, 2009.

[15] *IEEE Standard for Information technology – Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements.* Part 2: Logical Link Control. New York: The Institute of Electrical and Electronics Engineers. IEEE Std 802.2^{TM}, May 7, 2008.

[16] IEEE Draft Guide for Wireless Access in Vehicular Environments (WAVE) -Architecture, IEEE P1609.0^{TM}/D5, September 2012.

[17] *IEEE Standard for Wireless Access in Vehicular Environments - Security Services for Applications and Management Messages*, IEEE Std 1609.2^{TM}, 2013.

[18] *IEEE Standard for Wireless Access in Vehicular Environments (WAVE) - Networking Services*, IEEE Std 1609.3^{TM}, 2010.

[19] *IEEE Standard for Wireless Access in Vehicular Environments (WAVE) - Multi-Channel Operation*, IEEE Std 1609.4^{TM}, 2010.

[20] Amendment of the Commission’s Rules Regarding Dedicated Short-Range Communication Services in the 5.850-5.925 GHz Band (5.9 GHz Band), Federal Communications Commission FCC 03-324, 2004.

[21] Standard Specification for Telecommunications and Information Exchange Between Roadside and Vehicle Systems - 5 GHz Band Dedicated Short Range Communications (DSRC) Medium Access Control (MAC) and Physical Layer (PHY) Specifications, ASTM E2213-03, 2010.

[22] Y.J. Li, "An Overview of the DSRC/WAVE Technology", in Quality, Reliability, Security and Robustness in Heterogeneous Networks, Springer, 2012.

[23] R. A. Uzcategui and G. Acosta-Marum, "WAVE: A Tutorial", *IEEE Commun. Mag.*, vol. 47, no. 5, pp. 126-133, 2009.

[24] S.A.M. Ahmed, S.H.S. Ariffin, N. Fisal, "Overview of Wireless Access in Vehicular Environment (WAVE) Protocols and Standards"", Indian Journal of Science and Technology, 2013.

[25] *Dedicated Short Range Communications (DSRC) Message Set Dictionary. Society of Automotive Engineers*, SAE J2735, November 2009.

[26] Status of the Dedicated Short-Range Communications Technology and Applications. Report to Congress, Final Report, July 2015 FHWA-JPO-15-218.

[27] *NTCIP 9001: National Transportation Communications for ITS Protocol - The NTCIP Guide*, Version, v04, American Association of State Highway and Transportation Officials, Washington, D.C.; Institute of Transportation Engineers, Washington D.C.; and National Electrical Manufacturers Association, Rosslyn, VA, July 2009.

[28] A. Krings, A. Serageldin, and A. Abdel-Rahim, *A Prototype for a Real-Time Weather Responsive System*, Proc. Intelligent Transportation Systems Conference, ITSC2012, Anchorage, Alaska, 16-19 September, pp. 1465 - 1470, 2012.

[29] A. Serageldin, and A. Krings, *The Impact of Redundancy on DSRC Safety Application Reliability under Different Data Rates*, Proc. 6^{th} International Conference on New Technologies, Mobility and Security, (NTMS 2014), Dubai, March 30 April 2, 2014.

[30] NTCIP website: http://www.ntcip.org/.

[31] J. Case *et al.*, "A Simple Network Management Protocol", IETF RFC 1175, 1990.

[32] V. Harigovindan, A. Babu and L. Jacob, "Ensuring fair access in IEEE 8002.11p-based vehicle-to-infrastructure", *EURASIP J. Wireless Commun. Netw.*, vol. 2012, no. 168, 2012.

[33] Arada Systems, www.aradasystems.com.

[34] *Transportation Systems and Engineering: Concepts, Methodologies, Tools, and Applications*, Information Resources Management Association, USA, 2015.

[35] Arada Systems, *LocoMate User's Guide*, Version 1.26, 2013.

[36] Traffic Signal Timing Manual, Kittelson & Associates, Inc. For FHWA, 2008.

[37] B. Ghena, W. Beyer, A. Hillaker, J. Pevarnek, and J.A. Halderman. "Green lights forever: Analyzing the security of traffic infrastructure", *8th USENIX Workshop on Offensive Technologies*, August 2014.

[38] Federal Highway Administration (FHWA), Texas Department of Transportation(txdot), http://onlinemanuals.txdot.gov/txdotmanuals/tri/images/FHWA_Classification_Chart_FINAL.png

[39] "MIPS32 Architecture",. Imagination Technologies. Retrieved 4 Jan 2014.

Appendix A

Communicating with the Traffic Controller

A.1 Global Variables

This utility file contains a set of global variables. It also redefines some of the primitive data types.

```
/*
 * DTC_GLOBAL_DEFS.h
 *
 *   Created on: Apr 1, 2015
 *       Author: Anup Chitrakar
 */

/*** BeginHeader */
//********************************
#ifndef DTC_GLOBAL_DEFS_H
#define DTC_GLOBAL_DEFS_H
#endif

// GENERAL DEFINATIONS
//********************************
#ifndef CLEAR
#define CLEAR 0
#endif
#ifndef SET
#define SET     1
#endif
#ifndef FALSE
#define FALSE   0
#endif
#ifndef TRUE
#define TRUE    1
#endif
#ifndef ON
#define ON      1
#endif
#ifndef OFF
#define OFF     0
#endif

//PRIMITIVE DATA TYPES DEFINATIONS
//********************************
typedef char              Int8;
typedef unsigned char     UInt8;
typedef unsigned char     Byte;
typedef int               Int16;
typedef unsigned int      UInt16;
typedef long              Int32;
typedef unsigned long     UInt32;
/*** EndHeader */
```

A.2 DYNAMIC_SIGNAL_CONTROL_S

This file defines a set of data structures that are required for communicating with the traffic controller. Furthermore, it also defines a collection of required object identifiers.

```
/*
 * DYNAMIC_SIGNAL_CONTROL_S.h
 *
 *  Created on: Apr 2, 2015
 *      Author: Anup Chitrakar
 */

/*** BeginHeader */
//*********************************
#ifndef DYNAMIC_SIGNAL_CONTROL_C_H
#define DYNAMIC_SIGNAL_CONTROL_C_H
#endif

// DEBUG DEFINATIONS
//*********************************
#ifdef DSC_DEBUG
#define _DSC_DEBUG debug
#else
#define _DSC_DEBUG nodebug
#endif

//ENTERNET SPECIFIC DEFINATIONS
//*********************************
/* IP Address of Traffic Controller*/
#define TC_IP        "192.168.1.102"

/* Traffic Controllers Port. See TC Setup*/
#define TC_PORT          2222

/*Required for SNMP Data*/
#define TC_SET_MSG_PORT      162

//APPLICATION SPECIFIC DEFINATIONS
//*********************************
#define SNMP_MSG_POID15_SIZE sizeof(SnmpMsgPublicOidSizeOXOF)
#define SNMP_MSG_SIZE_Get   sizeof(SnmpMsgPublicOidGet)

//SNMP TAGS AND SIZE
//*********************************
/* Currently supports only OID size of 15*/
#define OID_SZ  0x0F

//SNMP DYNAMIC OBJECT CONTROL COMMANDS
// -- MSN->cmd and LSN->object number
//*********************************
#define SNMP_SET 0x90
#define SNMP_GET 0x80
#define DYN_OBJ_REQ_PACKET_SZ 0x01 //snmp get packet size (1
    byte)

#define TC_GETRES_MSG_RECV      0x02
#define TC_SETRES_MSG_RECV      0x04

static Byte _set_sock_status_flag;
static Byte _get_sock_status_flag;

/*** BeginHeader */
//********************************
//CONSTANT BYTES STREAMS FOR SNMP GET MESSAGES ...................................
//SEE PAGE 53-56 OF THE NTCIP GUIDE 9001 FOR DECODING SNMP MESSAGES PAGE 129/250
const Byte _get_global_time_encoded[] = {0x30, 0x2b, 0x02, 0x01, 0x00, 0x04, 0x06, 0x70, 0x75, 0x62, 0x6c, 0x69, 0x63, 0xa0, 0x1e, 0x02, 0x01, 0x00, 0x02, 0x01, 0x00, 0x02, 0x01, 0x00, 0x30, 0x13, 0x30, 0x11, 0x06, 0x0d, 0x2b, 0x06, 0x01, 0x04, 0x01, 0x89, 0x36, 0x04, 0x02, 0x06, 0x03, 0x01, 0x00, 0x05, 0x00};
const Byte _init_snmp_set_msg[] = {0x30, 0x2e, 0x02, 0x01, 0x00, 0x04, 0x06, 0x70, 0x75, 0x62, 0x6c, 0x69, 0x63, 0xa3, 0x21, 0x02, 0x01, 0x00, 0x02, 0x01, 0x00, 0x02, 0x01, 0x00, 0x30, 0x16, 0x30, 0x14, 0x06, 0x0f, 0x2b, 0x06, 0x01, 0x04, 0x01, 0x89, 0x36, 0x04, 0x02, 0x01, 0x01, 0x02, 0x01, 0x06, 0x01, 0x02, 0x01, 0x23 };
const Byte _init_snmp_get_msg[] = {0x30, 0x2d, 0x02, 0x01, 0x00, 0x04, 0x06, 0x70, 0x75, 0x62, 0x6c, 0x69, 0x63, 0xa0, 0x20, 0x02, 0x01, 0x00, 0x02, 0x01, 0x00, 0x02, 0x01, 0x00, 0x30, 0x15, 0x30, 0x13, 0x06, 0x0f, 0x2b, 0x06, 0x01, 0x04, 0x01, 0x89, 0x36, 0x04, 0x02, 0x01, 0x01, 0x02, 0x01, 0x06, 0x01, 0x05, 0x00};

/*Object Identifiers*/

const Byte _phase_min_green_encoded_oid[] = {0x2b, 0x06, 0x01, 0x04, 0x01, 0x89, 0x36, 0x04, 0x02, 0x01, 0x01, 0x02, 0x01, 0x04};

const Byte _phase_passage_encoded_oid[] =  0x2b, 0x06, 0x01, 0x04, 0x01, 0x89, 0x36, 0x04, 0x02, 0x01, 0x01, 0x02, 0x01, 0x05};

const Byte _phase_max1_encoded_oid[] = {0x2b, 0x06, 0x01, 0x04, 0x01, 0x89, 0x36, 0x04, 0x02, 0x01, 0x01, 0x02, 0x01, 0x06};

const Byte _phase_max2_encoded_oid[] = {0x2b, 0x06, 0x01, 0x04, 0x01, 0x89, 0x36, 0x04, 0x02, 0x01, 0x01, 0x02, 0x01, 0x07};

const Byte red_clear[] = {0x2b, 0x06, 0x01, 0x04, 0x01, 0x89, 0x36, 0x04, 0x02, 0x01, 0x01, 0x02, 0x01, 0x09};

const Byte _phase_yellow_encoded_oid[] = {0x2b, 0x06, 0x01, 0x04, 0x01, 0x89, 0x36, 0x04, 0x02, 0x01, 0x01, 0x02, 0x01, 0x08};

const Byte _phase_timeToReduce_encoded_oid[] = {0x2b, 0x06, 0x01, 0x04, 0x01, 0x89, 0x36, 0x04, 0x02, 0x01, 0x01, 0x02, 0x01, 0x0f};

const Byte phase_stats_group_number[] = {0x2b, 0x06, 0x01, 0x04, 0x01, 0x89, 0x36, 0x04, 0x02, 0x01, 0x01, 0x04, 0x01, 0x01};

const Byte phase_status_group_reds[] = {0x2b, 0x06, 0x01, 0x04, 0x01, 0x89, 0x36, 0x04, 0x02, 0x01, 0x01, 0x04, 0x01, 0x02};

const Byte phase_status_group_yellows[] = {0x2b, 0x06, 0x01, 0
    x04, 0x01, 0x89, 0x36, 0x04, 0x02, 0x01, 0x01, 0x04, 0x01, 0
    x03};

const Byte phase_status_group_greens[] = {0x2b, 0x06, 0x01, 0x04
    , 0x01, 0x89, 0x36, 0x04, 0x02, 0x01, 0x01, 0x04, 0x01, 0x04
    };


//DATA STRUCTURES
    ..........................................................

// From Document NTCIP 9001 (Page 93)
//SNMP MESSAGE FIELDS, SetReqiest PDU, Varible Bindings,
    Bindings
typedef struct {
//SNMP MESSAGE FIELDS
  Byte seqTag;
  Byte seqLen;
  Byte versionTag;
  Byte versionLen;
  Byte versionVal;
  Byte communityNameTag;
  Byte communityNameLen;
  Byte communityNameVal[6];
//SET REQUEST PDU NUMBER
  Byte pduType;
  Byte pduLen;
  Byte requestIdTag;
  Byte requestIdlen;
  Byte requestIdVal;
  Byte errorStatusTag;
  Byte errorStatusLen;
  Byte errorStatusVal;
  Byte errorIndexTag;
  Byte errorIndexLen;
  Byte errorIndexVal;
//VARIABLE BINDINGS
  Byte seqVarBinding;
  Byte seqVarBindingLen;
//BINDING
  Byte seqBinding;
  Byte seqLenOrId;
  Byte varTag;
  Byte varLen;
  Byte varId[15];
  Byte varValueTag;
  Byte varValueLen;
  Byte varValueVal;
} SnmpMsgPublicOidSizeOXOF;
typedef struct {
//SNMP MESSAGE FIELDS
  Byte seqTag;
  Byte seqLen;
  Byte versionTag;
  Byte versionLen;
  Byte versionVal;
  Byte communityNameTag;
  Byte communityNameLen;
```

```
    Byte communityNameVal[6];
//SET REQUEST PDU NUMBER
    Byte pduType;
    Byte pduLen;
    Byte requestIdTag;
    Byte requestIdlen;
    Byte requestIdVal;
    Byte errorStatusTag;
    Byte errorStatusLen;
    Byte errorStatusVal;
    Byte errorIndexTag;
    Byte errorIndexLen;
    Byte errorIndexVal;
//VARIABLE BINDINGS
    Byte seqVarBinding;
    Byte seqVarBindingLen;
//BINDING
    Byte seqBinding;
    Byte seqLenOrId;
    Byte varTag;
    Byte varLen;
    Byte varId[15];
    Byte varValueTag;
    Byte varValueLen;
} SnmpMsgPublicOidGet;
union {
    SnmpMsgPublicOidGet snmpMsg;
    char buff[SNMP_MSG_SIZE_Get];
} SnmpMsgGet;
union {
    SnmpMsgPublicOidSizeOXOF snmpMsg;
    char buff[SNMP_MSG_POID15_SIZE];
} SnmpMsgPOXOF;
union {
    SnmpMsgPublicOidSizeOXOF snmpMsgResp;
    char buffer[SNMP_MSG_POID15_SIZE];
} SnmpMsgPOXOFresp;

//DYNAMIC_OBJECT_NUMBER DEFINATIONS -- currently using only one
    dynamic object
//*********************************
#define MAX_GREEN 0x28
#define MAX_PHASES   0x08

//DEFINATIONS FOR FAILURE FLAGS -- _cf_flags and _gapout_flags
/* No failure was detected for a given phase/FM */
#define DSC_CODE0    0x00

/* Failure detected for a given phase/PM */
#define DSC_CODE1          0x01

/* Feedback adjusted the TC parameter based on FM */
#define DSC_CODE2          0x02
```

A.3 Controller Access Utility

This utility file is mainly responsible for creating the sockets. It also provides a framework for accessing the traffic controller in order to send get and set requests.

```
/*
 * ControllerAccessUtility.h
 *
 *  Created on: May 18, 2015
 *      Author: root
 */

#ifndef CONTROLLERACCESSUTILITY_H_
#define CONTROLLERACCESSUTILITY_H_
#endif /* CONTROLLERACCESSUTILITY_H_ */

#include <sys/socket.h>

#ifndef DTC_GLOBAL_DEFS_H
#include "DTC_GLOBAL_DEFS.h"
#endif

#ifndef DYNAMIC_SIGNAL_CONTROL_C_H
#include "DYNAMIC_SIGNAL_CONTROL_S.h"
#endif

Byte tc_get_groups(const unsigned char * oid);
Byte tc_get(const unsigned char * oid, Byte phase);
Byte tc_set(const unsigned char * oid, Byte phase, Byte value);

int getValue();
void dsc_init(void);
void dsc_init_global_vars(void);

#define ACTIVE_PHASES 8 //must be sequential, eg in this case p1
    :8 are on

int fd, c, n, p;
struct _dsc_tcd_socket;
int slen;

Byte tc_get_groups(const unsigned char * oid) {
  Byte ret_val;
  ret_val = DSC_CODE1;
  if (_set_sock_status_flag == TC_SETRES_MSG_RECV &&
       _get_sock_status_flag == TC_GETRES_MSG_RECV) {
    memcpy(SnmpMsgGet.snmpMsg.varId, oid, (OID_SZ - 1));
    if (sendto(fd, SnmpMsgGet.buff, SNMP_MSG_SIZE_Get, 0,
        (struct sockaddr *) &_dsc_tcd_socket, slen) > 0) {
      ret_val = DSC_CODE2;
    }

    //Receive a reply from the server
    if (recv(fd, SnmpMsgPOXOFresp.buffer, SNMP_MSG_POID15_SIZE,
        0) < 0) {
      puts("recv failed");
      exit(1);
    }
  }
  return ret_val;
}

Byte tc_get(const unsigned char * oid, Byte phase) {
  Byte ret_val;
  ret_val = DSC_CODE1;
  if (_set_sock_status_flag == TC_SETRES_MSG_RECV &&
      _get_sock_status_flag == TC_GETRES_MSG_RECV) {
    memcpy(SnmpMsgGet.snmpMsg.varId, oid, (OID_SZ - 1));
    SnmpMsgGet.snmpMsg.varId[OID_SZ - 1] = phase + 1;
    if (sendto(fd, SnmpMsgGet.buff, SNMP_MSG_SIZE_Get, 0,
        (struct sockaddr *) &_dsc_tcd_socket, slen) > 0) {
      ret_val = DSC_CODE2;
    }

    //Receive a reply from the server
    if (recv(fd, SnmpMsgPOXOFresp.buffer, SNMP_MSG_POID15_SIZE,
        0) < 0) {
      puts("recv failed");
      exit(1);
    }
  }
  return ret_val;
}

/* START FUNCTION DESCRIPTION ────────────────────────────────────
 tc_set          <DYNAMIC_SIGNAL_CONTROL.LIB>
 SYNTAX:          Byte tc_set(const char * oid, Byte phase, Byte
     value)
 DESCRIPTION:   sends snmp set messages to the traffic controller
 PARAMETER1:     object id of the TC parameter to set
 PARAMETER2:     Phase number— 1:16
 PARAMETER3:     value to be written to the TC parameter
 RETURN VALUE: DSC_CODE1 for failure, or DSC_CODE2 for success
 END DESCRIPTION
     ──────────────────────────────────────────────────────────
     */
Byte tc_set(const unsigned char * oid, Byte phase, Byte value) {
  Byte ret_val;
  ret_val = DSC_CODE1;
  if (_set_sock_status_flag == TC_SETRES_MSG_RECV &&
      _get_sock_status_flag == TC_GETRES_MSG_RECV) {
    memcpy(SnmpMsgPOXOF.snmpMsg.varId, oid, (OID_SZ - 1));
    SnmpMsgPOXOF.snmpMsg.varId[OID_SZ - 1] = phase;
    SnmpMsgPOXOF.snmpMsg.varValueVal = value;

    if (sendto(fd, SnmpMsgPOXOF.buff, SNMP_MSG_POID15_SIZE, 0,
        (struct sockaddr *) &_dsc_tcd_socket, slen) > 0) {
      ret_val = DSC_CODE2;
    }
  }
  return ret_val;
}

int getValue() {
  return (int) SnmpMsgPOXOFresp.snmpMsgResp.varValueVal;
}

```

```
void dsc_init(void) {
  dsc_init_global_vars(); //Initializes all storage elements in
      this library

  if ((fd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
    printf("socket created\n");
  slen = sizeof(_dsc_tcd_socket);
  char *server = TC_IP; /* change this to use a different server
       */
  memset((char *) &_dsc_tcd_socket, 0, sizeof(_dsc_tcd_socket));
  _dsc_tcd_socket.sin_family = AF_INET;
  _dsc_tcd_socket.sin_port = htons(TC_PORT);
  if (inet_aton(server, &_dsc_tcd_socket.sin_addr) == 0) {
    fprintf(stderr, "inet_aton() failed\n");
    exit(1);
  }
}

void dsc_init_global_vars(void) {
  _get_sock_status_flag = TC_GETRES_MSG_RECV;
  _set_sock_status_flag = TC_SETRES_MSG_RECV;
  memcpy(SnmpMsgPOXOF.buff, _init_snmp_set_msg,
      SNMP_MSG_POID15_SIZE);
  memcpy(SnmpMsgGet.buff, _init_snmp_get_msg, SNMP_MSG_SIZE_Get)
      ;
}
```

A.4 Phase Information

This file is required in order to query the current status of traffic lights at any phases in the intersection. This utility has the capability to query any light (green, yellow, or red) in the intersection. However, ETSS queries only the status of the green light.

```
/*
 * PhaseInformation.h
 *
 *  Created on: May 18, 2015
 *      Author: Anup Chitrakar
 */

#ifndef PHASEINFORMATION_H_
#define PHASEINFORMATION_H_
#endif /* PHASEINFORMATION_H_ */

#ifndef DYNAMIC_SIGNAL_CONTROL_C_H
#include "DYNAMIC_SIGNAL_CONTROL_S.h"
#endif

#define TRUE 1
#define FALSE 0

#define SUCCESS 1
#define FAIL 0

typedef struct {
  int phaseNumber;  //1 to 8 (MAXPHASE)
  char phaseColor;  //R, Y, G
  int phaseStat;    //1 for on and 0 for off
} PhaseStatus;

int * decimalToBinary(int);

void printBinaryValue(int *x);

void getPhaseInformation(int *x, char color);

void printPhaseInformation(char light);

int isLightOn(int phaseNum, char color);

int queryForLight(int phase, char color);

int isGreenLightOn(int phase);

int setGreenPhaseExtension(Byte phase, Byte value);

PhaseStatus greenLightInformation[MAX_PHASES];
PhaseStatus yellowLightInformation[MAX_PHASES];
PhaseStatus redLightInformation[MAX_PHASES];

int queryForLight(int phase, char color) {
  if (color == 'G') {
    /*Check if Green light is on in a particular phase*/
    if (isLightOn(phase, 'G')) {
      return TRUE;
    } else {
      return FALSE;
    }
  } else if (color == 'Y') {
    /*Check if Yello light is on in a particular phase*/
    if (isLightOn(phase, 'Y')) {
      return TRUE;
    } else {
      return FALSE;
    }
  } else if (color == 'R') {
    /*Check if Red light is on in a particular phase*/
    if (isLightOn(phase, 'R')) {
      return TRUE;
    } else {
      return FALSE;
    }
  }
  return FALSE;
}

/**
 * This is a support function function for "int queryForLight(
     int phase, char color)".
 */
int isLightOn(int phaseNum, char color) {
  int i;
  for (i = 0; i < MAX_PHASES; i++) {
    if (color == 'G') {
      if (greenLightInformation[i].phaseNumber == phaseNum
          && greenLightInformation[i].phaseColor == color
          && greenLightInformation[i].phaseStat == 1) {
        return TRUE;
      }

    } else if (color == 'Y') {
      if (yellowLightInformation[i].phaseNumber == phaseNum
          && yellowLightInformation[i].phaseColor == color
          && yellowLightInformation[i].phaseStat == 1) {
        return TRUE;
      }
    } else if (color == 'R') {
      if (redLightInformation[i].phaseNumber == phaseNum
          && redLightInformation[i].phaseColor == color
          && redLightInformation[i].phaseStat == 1) {
        return TRUE;
      }
    }
  }
  return FALSE;
}

void printPhaseInformation(char color) {
  int i = 0;
  for (i = 0; i < MAX_PHASES; i++) {
    if (color == 'G') {
      printf("\nPhase %d: %d , Color: %c",
            greenLightInformation[i].phaseNumber,
            greenLightInformation[i].phaseStat,
        greenLightInformation[i].phaseColor);
    } else if (color == 'Y') {
      printf("\nPhase %d: %d , Color: %c",
        yellowLightInformation[i].phaseNumber,
        yellowLightInformation[i].phaseStat,
        yellowLightInformation[i].phaseColor);
    } else if (color == 'R') {
      printf("\nPhase %d: %d , Color: %c",
        redLightInformation[i].phaseNumber,
        redLightInformation[i].phaseStat,
        redLightInformation[i].phaseColor);
    } else {
      printf("\nInvalid Traffic Signal Color\n");
    }
  }
  printf("\n");
}

/**
 * This is a support function function for "int isLightOn(int phaseNum, char color)".
 * This function gathers information about all color (i.e., Green, Yello, and Red) for all phases.
 */
void getPhaseInformation(int *x, char color) {
  int i;

  int phaseNumber = 0;

  dsc_init();
  tc_get_groups(phase_stats_group_number);
  int phaseStatusGroupNumber = getValue();

  for (i = 0; i < MAX_PHASES; i++) {
    phaseNumber = (phaseStatusGroupNumber * 8) - i;
    if (color == 'G') {
      greenLightInformation[i].phaseNumber = phaseNumber;
      greenLightInformation[i].phaseColor = color;
      greenLightInformation[i].phaseStat = x[i];
    } else if (color == 'Y') {
      yellowLightInformation[i].phaseNumber = phaseNumber;
      yellowLightInformation[i].phaseColor = color;
      yellowLightInformation[i].phaseStat = x[i];
    } else if (color == 'R') {
      redLightInformation[i].phaseNumber = phaseNumber;
      redLightInformation[i].phaseColor = color;
      redLightInformation[i].phaseStat = x[i];
    }
  }
}

void printBinaryValue(int *x) {
  int i;
  for (i = 0; i < 8; i++) {
    printf("%d", x[i]);
  }
}

```

```
/**
 * This function convers a decimal integer into binary bits and
     stores those bits into an array.
 */
int * decimalToBinary(int num) {
  int i = 0;
  int c, k;
  static int status[8];
  static int error[8] = { -1 };
  if (num < 0 || num > 255) {
    printf("\nNot Valid Data\n");
    return error;
  }

  for (c = 7; c >= 0; c--) {
    k = num >> c;
    if (k & 1) {
      status[i] = 1;
    } else {
      status[i] = 0;
    }
    i++;
  }
  return status;
}

/**
 * Phase should be from 1 through MAX_PHASE (8/16), and the
     value should be from 0-255
 */
int setGreenPhaseExtension(Byte phase, Byte value) {
  if (phase < 0 || phase > MAX_PHASES || value < 0 || value >
      255) {
    return FAIL;
  } else {
    dsc_init();
    tc_set(_phase_passage_encoded_oid, phase, value);
    return SUCCESS;
  }
}

int isGreenLightOn(int phase) {
  printf("Initializing TCP/IP stack...\n");

  int *greenLightBinary = NULL;
  int *yellowLightBinary = NULL;

  int greenLightVal;
  int yellowLightVal;
  int redLightVal;

  dsc_init();
  tc_get_groups(phase_status_group_reds);
  redLightVal = getValue();

  dsc_init();
  tc_get_groups(phase_status_group_greens);
  greenLightVal = getValue();

  dsc_init();
  tc_get_groups(phase_status_group_yellows);
  yellowLightVal = getValue();

  if (yellowLightVal != 0) {
    greenLightBinary = decimalToBinary(0);
    getPhaseInformation(greenLightBinary, 'G');

    yellowLightBinary = decimalToBinary(yellowLightVal);
    getPhaseInformation(yellowLightBinary, 'Y');

    int phase;
    for (phase = 1; phase <= MAX_PHASES; phase++) {
      if (queryForLight(phase, 'Y') == TRUE) {
        printf("Phase %d :: Yellow IS ON\n", phase);
      } else {
        continue;
      }
    }

  } else {
    if (redLightVal > 0 && greenLightVal == 0) {
      int temp = 255 - redLightVal;
      greenLightBinary = decimalToBinary(temp);
      getPhaseInformation(greenLightBinary, 'G');
    } else {
      greenLightBinary = decimalToBinary(greenLightVal);
      getPhaseInformation(greenLightBinary, 'G');
    }
    if (queryForLight(phase, 'G') == TRUE) {
      return TRUE;
    } else {
      return FALSE;
    }
  }

  return 0;
}
```

A.5 Main Program

This is a sample program that verifies if the communication with a traffic controller was successful. In this main program the status of a green light in a particular phase is queried.

```
/*
 * client-send.c
 *
 *  Created on: Apr 1, 2015
 *      Author: Anup Chitrakar
 *
 *      Description:
 *      udp-send: a simple udp client
 *      send udp messages
 *      sends a sequence of messages (the # of messages is
    defined in MSGS)
 *      The messages are sent to a port defined in SERVICE_PORT
    available at DTC_GLOBAL_DEFS.h
 *
 *      usage:  udp-send
 *
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#include <unistd.h>

#include "ControllerAccessUtility.h"
#include "PhaseInformation.h"

int isGreenLightOn(int);

int main() {
  int phase = 4;
  if (isGreenLightOn(phase)) {
    printf("\nGreen Light in phase %d is ON\n", phase);
  } else {
    printf("\nGreen Light in phase %d is OFF\n", phase);
  }
  return 1;
}
```

Appendix B

Toolchain Setup

As a software developer, one needs to have a development platform ready before starting to write code. In our case, we will need a machine (Linux machine preferred), where the Microprocessor without Interlocked Pipeline Stages (MIPS) [39] SDK will be installed. Setting up the development environment depends on the machine configuration. Steps to setup the development environment in both 64 bit and 32 bit machines are explained below.

B.1 Linux 64 bit Version

Listed below are the procedures to install the mips sdk into 64 bit Linux machine:

1. Download toolchain[LocoMate `Toolchain_2013` [Posted Jan/31/2015]]from http://support.aradasystems.com/file.php?tab=files&file=3849.
   Note: For Username and password, Arada customer service center needs to be contacted.

2. Download mips sdk [Locomate-mips Source (Version 1.90) [Posted Jan/31/2015]] from http://support.aradasystems.com/file.php?tab=files&file=3850

3. Execute: sudo tar -Pjxvf `latest_2013_locomate-toolchain.tar.bz2`

4. Execute: sudo tar -Pjxvf locomate 1.90-mips.tar.bz2

5. Go to: cd /usr/src/locomate-release/mips/src/

6. Execute: make

If no error messages are displayed in console, then the 32 bit libraries were already installed. Otherwise one might end up with an error message saying, “mips-linux-gcc -I //usr/src/locomate-release/mips/src/../incs -I //usr/src/locomate-release/mips/src/../src -c -DLOCOMATE `-DSDK_-NEW` -o wsmpdemo.o wsmpdemo.c make: mips-linux-gcc: Command not found make: *** [wsmpdemo.o] Error 127”. In this case refer to the steps described in Subsection B.1.1.

B.1.1 Resolution

Install some 32 bit libraries.

Before starting to compile, one needs to make sure that the lib32z1, lib32ncurses5 and lib32z2-1.0 libraries are installed on your host system. Else install it by following commands on command line:

- sudo apt-get install lib32z1
- sudo apt-get install lib32ncurses5
- sudo apt-get install lib32bz2-1.0

B.2 Linux 32 bit Version

Listed below are the procedures to install the mips sdk into 32 bit Linux machine:

1. Download toolchain[LocoMate `Toolchain_2013` [Posted Jan/31/2015]]from http://support.aradasystems.com/file.php?tab=files&file=3849
2. Download mips sdk [Locomate-mips Source (Version 1.90) [Posted Jan/31/2015]] from http://support.aradasystems.com/file.php?tab=files&file=3850
3. Execute: sudo tar -Pjxvf `latest_2013_locomate-toolchain.tar.bz2`
4. Execute: sudo tar -Pjxvf locomate 1.90-mips.tar.bz2
5. Go to: cd /usr/src/locomate-release/mips/src/
6. Execute: sudo -s
7. Execute: make

B.2.1 Install WinSCP

To run WinSCP under Linux (Ubuntu 12.04), follow these steps:

1. Run sudo apt-get install wine (run this one time only, to get 'wine' in your system, if you haven't it)
2. Download http://winscp.net/download/winscp553.zip
3. Make a folder and put the content of zip file in this folder
4. Open a terminal

5. Type sudo su

6. Type wine WinSCP.exe Done! WinSCP will run like in Windows environment!

Alternatively, for transferring the executable from host machine to the remote locomate device, following scp command can be used:

sudo sshpass -p 'password' scp myRx root@192.168.1.40:/var/arada_locomate

Here, *root@192.168.1.40:/var/arada_locomate* is the destination and *myRx* is the name of the executable that needs to be transferred.

Appendix C

Setup Development Environment

Before setting up a development platform, following assumptions are made:

1. **Assumption 1** - *Using Linux Environment*: The developer is supposed to be installing a Linux platform in one's computer or laptop.

2. **Assumption 2** - *Eclipse IDE*: It is recommended to use an Integrated Development Environment (IDE) for writing and debugging code. Therefore, Eclipse IDE is preferred to be used. Eclipse is primarily used for developing Java applications, it may also be used to develop applications in other programming languages via the use of plugins. The C/C++ Development Tooling (CDT) provides a fully functional C and C++ Integrated Development Environment based on the Eclipse platform.

C.1 Import Arada Locomate source codes into Eclipse IDE

1. Open Eclipse using root.

2. File → New → Other → Makefile Project with Existing Code.

3. Click Next.

4. Put project Name: DSRCProject

5. For Existing Code Location, enter following path:

   /usr/src/locomate-release/mips/src

6. Project → properties → C/C++ General → Indexer

   Build configuration for the indexer → Use active build configuration

7. Project → Properties → C/C++ General → Paths and Symbols
   In **Includes Tab:**, Select GNU C and Click Add... button

   - /usr/src/locomate-release/mips/incs
   - /usr/include
   - /usr/include linux

- /opt/buildroot-2013.11/output/host/usr/mips-buildroot-linux-uclibc/sysroot/usr/include
- /opt/buildroot-2013.11/output/host/usr/mips-buildroot-linux-uclibc/sysroot/usr/include/linux

In **Libraries Tab:**, Click Add... button

- /usr/src/locomate-release/mips/lib

In **Libraries Paths:**, Click Add... button

- /usr/src/locomate-release/mips/lib

8. Build the project and there you go !!!

C.2 Import ASC-3 source codes into Eclipse IDE

1. Open Eclipse using root.
2. File → New → C Project.
3. Put Project Name: ASC-3.
4. Uncheck "Use default location".
5. Provide the source location, in my case it is:
   "/home/traffic/Anup/Research/TrafficController/trunk"
6. Click Next → Next → Finish.
7. Right click Project ASC-3 and go to Properties.
8. In C/C++ General, go to Paths and Symbols.
9. In Includes tab → GNU, click Add... button.

   /home/traffic/Anup/Research/TrafficController/trunk/incl
10. Click Ok and then a dialog box appears asking if you want to rebuild it now. Click Yes.

Appendix D

NTCIP Object Identifiers

Table D.1: List of NTCIP Global Objects

NTCIP 1201 OID	Object Name
1.3.6.1.4.1.1206.4.2.6.3	globalTimeManagement
1.3.6.1.4.1.1206.4.2.6.3.1	global.Time
1.3.6.1.4.1.1206.4.2.6.3.2	globalDaylightSaving
1.3.6.1.4.1.1206.4.2.6.3.3	timebase
1.3.6.1.4.1.1206.4.2.6.3.3.1	maxTimeBaseScheduleEntries
1.3.6.1.4.1.1206.4.2.6.3.3.2	timeBaseScheduleTable
1.3.6.1.4.1.1206.4.2.6.3.3.2.1	timeBaseScheduleEntry
1.3.6.1.4.1.1206.4.2.6.3.3.2.1.1	timeBaseScheduleNumber
1.3.6.1.4.1.1206.4.2.6.3.3.2.1.2	timeBaseScheduleMonth
1.3.6.1.4.1.1206.4.2.6.3.3.2.1.3	timeBaseScheduleDay
1.3.6.1.4.1.1206.4.2.6.3.3.2.1.4	timeBaseScheduleDate
1.3.6.1.4.1.1206.4.2.6.3.3.2.1.5	timeBaseScheduleDayPlan
1.3.6.1.4.1.1206.4.2.6.3.3.3	maxDayPlans
1.3.6.1.4.1.1206.4.2.6.3.3.4	maxDayPlanEvents
1.3.6.1.4.1.1206.4.2.6.3.3.5	timeBaseDayPlanTable
1.3.6.1.4.1.1206.4.2.6.3.3.5.1	timeBaseDayPlanEntry
1.3.6.1.4.1.1206.4.2.6.3.3.5.1.1	dayPlanNumber
1.3.6.1.4.1.1206.4.2.6.3.3.5.1.2	dayPlanEventNumber
1.3.6.1.4.1.1206.4.2.6.3.3.5.1.3	dayPlanHour
1.3.6.1.4.1.1206.4.2.6.3.3.5.1.4	dayPlanMinute
1.3.6.1.4.1.1206.4.2.6.3.3.5.1.5	dayPlanActionNumberOID
1.3.6.1.4.1.1206.4.2.6.3.3.6	dayPlanStatus

Table D.2: List of ASC Object Identifiers

NTCIP 1201 OID	Object Name
1.3.6.1.4.1.1206.4.2.1.1	phase
1.3.6.1.4.1.1206.4.2.1.1.1	maxPhases
1.3.6.1.4.1.1206.4.2.1.1.2	phaseTable
1.3.6.1.4.1.1206.4.2.1.1.2.1.1	phaseNumber
1.3.6.1.4.1.1206.4.2.1.1.2.1.2	phaseWalk
1.3.6.1.4.1.1206.4.2.1.1.2.1.3	phasePedestrianClear
1.3.6.1.4.1.1206.4.2.1.1.2.1.4	phaseMinimumGreen

continued ...

Table D.2: List of ASC Object Identifiers

. . . continued

NTCIP 1201 OID	**Object Name**
1.3.6.1.4.1.1206.4.2.1.1.2.1.5	phasePassage
1.3.6.1.4.1.1206.4.2.1.1.2.1.6	phaseMaximum1
1.3.6.1.4.1.1206.4.2.1.1.2.1.7	phaseMaximum2
1.3.6.1.4.1.1206.4.2.1.1.2.1.8	phaseYellowChange
1.3.6.1.4.1.1206.4.2.1.1.2.1.9	phaseRedClear
1.3.6.1.4.1.1206.4.2.1.1.2.1.10	phaseRedRevert
1.3.6.1.4.1.1206.4.2.1.1.2.1.11	phaseAddedInitial
1.3.6.1.4.1.1206.4.2.1.1.2.1.12	phaseMaximumInitial
1.3.6.1.4.1.1206.4.2.1.1.2.1.13	phaseTimeBeforeReduction
1.3.6.1.4.1.1206.4.2.1.1.2.1.14	phaseCarsBeforeReduction
1.3.6.1.4.1.1206.4.2.1.1.2.1.15	phaseTimeToReduce
1.3.6.1.4.1.1206.4.2.1.1.2.1.16	phaseReduceBy
1.3.6.1.4.1.1206.4.2.1.1.2.1.17	phaseMinimumGap
1.3.6.1.4.1.1206.4.2.1.1.2.1.18	phaseDynamicMaxLimit
1.3.6.1.4.1.1206.4.2.1.1.2.1.19	phaseDynamicMaxStep
1.3.6.1.4.1.1206.4.2.1.1.2.1.20	phaseStartup
1.3.6.1.4.1.1206.4.2.1.1.2.1.21	phaseOptions
1.3.6.1.4.1.1206.4.2.1.1.2.1.22	phaseRing
1.3.6.1.4.1.1206.4.2.1.1.2.1.23	phaseConcurrency
1.3.6.1.4.1.1206.4.2.1.1.3	maxPhaseGroups
1.3.6.1.4.1.1206.4.2.1.1.4	phaseStatusGroupTable
1.3.6.1.4.1.1206.4.2.1.1.4.1	phaseStatusGroupEntry
1.3.6.1.4.1.1206.4.2.1.1.4.1.1	phaseStatusGroupNumber
1.3.6.1.4.1.1206.4.2.1.1.4.1.2	phaseStatusGroupReds
1.3.6.1.4.1.1206.4.2.1.1.4.1.3	phaseStatusGroupYellows
1.3.6.1.4.1.1206.4.2.1.1.4.1.4	phaseStatusGroupGreens
1.3.6.1.4.1.1206.4.2.1.1.4.1.5	phaseStatusGroupDontWalks
1.3.6.1.4.1.1206.4.2.1.1.4.1.6	phaseStatusGroupPedClears
1.3.6.1.4.1.1206.4.2.1.1.4.1.7	phaseStatusGroupWalks
1.3.6.1.4.1.1206.4.2.1.1.4.1.8	phaseStatusGroupVehCalls
1.3.6.1.4.1.1206.4.2.1.1.4.1.9	phaseStatusGroupPedCalls
1.3.6.1.4.1.1206.4.2.1.1.4.1.10	phaseStatusGroupPhaseOns
1.3.6.1.4.1.1206.4.2.1.1.4.1.11	phaseStatusGroupPhaseNexts
1.3.6.1.4.1.1206.4.2.1.1.5	phaseControlGroupTable
1.3.6.1.4.1.1206.4.2.1.1.5.1	phaseControlGroupEntry
1.3.6.1.4.1.1206.4.2.1.1.5.1.1	phaseControlGroupNumber
1.3.6.1.4.1.1206.4.2.1.1.5.1.2	phaseControlGroupPhaseOmit

continued . . .

Table D.2: List of ASC Object Identifiers

. . . continued

NTCIP 1201 OID	**Object Name**
1.3.6.1.4.1.1206.4.2.1.1.5.1.3	phaseControlGroupPedOmit
1.3.6.1.4.1.1206.4.2.1.1.5.1.4	phaseControlGroupHold
1.3.6.1.4.1.1206.4.2.1.1.5.1.5	phaseControlGroupForceOff
1.3.6.1.4.1.1206.4.2.1.1.5.1.6	phaseControlGroupVehCall
1.3.6.1.4.1.1206.4.2.1.1.5.1.7	phaseControlGroupPedCall
1.3.6.1.4.1.1206.4.2.1.2	detector
1.3.6.1.4.1.1206.4.2.1.2.1	maxVehicleDetectors
1.3.6.1.4.1.1206.4.2.1.2.2	vehicleDetectorTable
1.3.6.1.4.1.1206.4.2.1.2.2.1	vehicleDetectorEntry
1.3.6.1.4.1.1206.4.2.1.2.2.1.1	vehicleDetectorNumber
1.3.6.1.4.1.1206.4.2.1.2.2.1.2	vehicleDetectorOptions
1.3.6.1.4.1.1206.4.2.1.2.2.1.4	vehicleDetectorCallPhase
1.3.6.1.4.1.1206.4.2.1.2.2.1.5	vehicleDetectorSwitchPhase
1.3.6.1.4.1.1206.4.2.1.2.2.1.6	vehicleDetectorDelay
1.3.6.1.4.1.1206.4.2.1.2.2.1.7	vehicleDetectorExtend
1.3.6.1.4.1.1206.4.2.1.2.2.1.8	vehicleDetectorQueueLimit
1.3.6.1.4.1.1206.4.2.1.2.2.1.9	vehicleDetectorNoActivity
1.3.6.1.4.1.1206.4.2.1.2.2.1.10	vehicleDetectorMaxPresence
1.3.6.1.4.1.1206.4.2.1.2.2.1.11	vehicleDetectorErraticCounts
1.3.6.1.4.1.1206.4.2.1.2.2.1.12	vehicleDetectorFailTime
1.3.6.1.4.1.1206.4.2.1.2.2.1.13	vehicleDetectorAlarms
1.3.6.1.4.1.1206.4.2.1.2.2.1.14	vehicleDetectorReportedAlarms
1.3.6.1.4.1.1206.4.2.1.2.2.1.15	vehicleDetectorReset
1.3.6.1.4.1.1206.4.2.1.2.3	maxVehicleDetectorStatusGroups
1.3.6.1.4.1.1206.4.2.1.2.4	vehicleDetectorStatusGroupTable
1.3.6.1.4.1.1206.4.2.1.2.4.1	vehicleDetectorStatusGroupEntry
1.3.6.1.4.1.1206.4.2.1.2.4.1.1	vehicleDetectorStatusGroupNumber
1.3.6.1.4.1.1206.4.2.1.2.4.1.2	vehicleDetectorStatusGroupActive
1.3.6.1.4.1.1206.4.2.1.2.4.1.3	vehicleDetectorStatusGroupAlarms
1.3.6.1.4.1.1206.4.2.1.2.5	volumeOccupancyReport
1.3.6.1.4.1.1206.4.2.1.2.5.1	volumeOccupancySequence
1.3.6.1.4.1.1206.4.2.1.2.5.2	volumeOccupancyPeriod
1.3.6.1.4.1.1206.4.2.1.2.5.3	activeVolumeOccupancyDetectors
1.3.6.1.4.1.1206.4.2.1.2.5.4	volumeOccupancyTable
1.3.6.1.4.1.1206.4.2.1.2.5.4.1	volumeOccupancyEntry
1.3.6.1.4.1.1206.4.2.1.2.5.4.1.1	detectorVolume
1.3.6.1.4.1.1206.4.2.1.2.5.4.1.2	detectorOccupancy

continued . . .

Table D.2: List of ASC Object Identifiers

. . . continued

NTCIP 1201 OID	**Object Name**
1.3.6.1.4.1.1206.4.2.1.2.6	maxPedestrianDetectors
1.3.6.1.4.1.1206.4.2.1.2.7	pedestrianDetectorTable
1.3.6.1.4.1.1206.4.2.1.2.7.1	pedestrianDetectorEntry
1.3.6.1.4.1.1206.4.2.1.2.7.1.1	pedestrianDetectorNumber
1.3.6.1.4.1.1206.4.2.1.2.7.1.2	pedestrianDetectorCallPhase
1.3.6.1.4.1.1206.4.2.1.2.7.1.3	pedestrianDetectorNoActivity
1.3.6.1.4.1.1206.4.2.1.2.7.1.4	pedestrianDetectorMaxPresence
1.3.6.1.4.1.1206.4.2.1.2.7.1.5	pedestrianDetectorErraticCounts
1.3.6.1.4.1.1206.4.2.1.2.7.1.6	pedestrianDetectorAlarms
1.3.6.1.4.1.1206.4.2.1.3	unit
1.3.6.1.4.1.1206.4.2.1.3.1	unitStartupFlash
1.3.6.1.4.1.1206.4.2.1.3.2	unitAutoPedestrianClear
1.3.6.1.4.1.1206.4.2.1.3.3	unitBackupTime
1.3.6.1.4.1.1206.4.2.1.3.4	unitRedRevert
1.3.6.1.4.1.1206.4.2.1.3.5	unitControlStatus
1.3.6.1.4.1.1206.4.2.1.3.6	unitFlashStatus
1.3.6.1.4.1.1206.4.2.1.3.7	unitAlarmStatus2
1.3.6.1.4.1.1206.4.2.1.3.8	unitAlarmStatus1
1.3.6.1.4.1.1206.4.2.1.3.9	shortAlarmStatus
1.3.6.1.4.1.1206.4.2.1.3.10	unitControl
1.3.6.1.4.1.1206.4.2.1.3.11	maxAlarmGroups
1.3.6.1.4.1.1206.4.2.1.3.12	alarmGroupTable
1.3.6.1.4.1.1206.4.2.1.3.12.1	alarmGroupEntry
1.3.6.1.4.1.1206.4.2.1.3.12.1.1	alarmGroupNumber
1.3.6.1.4.1.1206.4.2.1.3.12.1.2	alarmGroupState
1.3.6.1.4.1.1206.4.2.1.3.13	maxSpecialFunctionOutputs
1.3.6.1.4.1.1206.4.2.1.3.14	specialFunctionOutputTable
1.3.6.1.4.1.1206.4.2.1.3.14.1	specialFunctionOutputEntry
1.3.6.1.4.1.1206.4.2.1.3.14.1.1	specialFunctionOutputNumber
1.3.6.1.4.1.1206.4.2.1.3.14.1.3	specialFunctionOutputControl
1.3.6.1.4.1.1206.4.2.1.3.14.1.4	specialFunctionOutputStatus
1.3.6.1.4.1.1206.4.2.1.4	coord
1.3.6.1.4.1.1206.4.2.1.4.1	coordOperationalMode
1.3.6.1.4.1.1206.4.2.1.4.2	coordCorrectionMode
1.3.6.1.4.1.1206.4.2.1.4.3	coordMaximumMode
1.3.6.1.4.1.1206.4.2.1.4.4	coordForceMode
1.3.6.1.4.1.1206.4.2.1.4.5	maxPatterns

continued . . .

Table D.2: List of ASC Object Identifiers

. . . continued

NTCIP 1201 OID	**Object Name**
1.3.6.1.4.1.1206.4.2.1.4.6	patternTableType
1.3.6.1.4.1.1206.4.2.1.4.7	patternTable
1.3.6.1.4.1.1206.4.2.1.4.7.1	patternEntry
1.3.6.1.4.1.1206.4.2.1.4.7.1.1	patternNumber
1.3.6.1.4.1.1206.4.2.1.4.7.1.2	patternCycleTime
1.3.6.1.4.1.1206.4.2.1.4.7.1.3	patternOffsetTime
1.3.6.1.4.1.1206.4.2.1.4.7.1.4	patternSplitNumber
1.3.6.1.4.1.1206.4.2.1.4.7.1.5	patternSequenceNumber
1.3.6.1.4.1.1206.4.2.1.4.8	maxSplits
1.3.6.1.4.1.1206.4.2.1.4.9	splitTable
1.3.6.1.4.1.1206.4.2.1.4.9.1	splitEntry
1.3.6.1.4.1.1206.4.2.1.4.9.1.1	splitNumber
1.3.6.1.4.1.1206.4.2.1.4.9.1.2	splitPhase
1.3.6.1.4.1.1206.4.2.1.4.9.1.3	splitTime
1.3.6.1.4.1.1206.4.2.1.4.9.1.4	splitMode
1.3.6.1.4.1.1206.4.2.1.4.9.1.5	splitCoordPhase
1.3.6.1.4.1.1206.4.2.1.4.10	coordPatternStatus
1.3.6.1.4.1.1206.4.2.1.4.11	localFreeStatus
1.3.6.1.4.1.1206.4.2.1.4.12	coordCycleStatus
1.3.6.1.4.1.1206.4.2.1.4.13	coordSyncStatus
1.3.6.1.4.1.1206.4.2.1.4.14	systemPatternControl
1.3.6.1.4.1.1206.4.2.1.4.15	systemSyncControl
1.3.6.1.4.1.1206.4.2.1.5	timebaseAsc
1.3.6.1.4.1.1206.4.2.1.5.1	timeBaseAscPatternSync
1.3.6.1.4.1.1206.4.2.1.5.2	maxTimebaseAscActions
1.3.6.1.4.1.1206.4.2.1.5.3	timebaseActionTable
1.3.6.1.4.1.1206.4.2.1.5.3.1	timebaseActionEntry
1.3.6.1.4.1.1206.4.2.1.5.3.1.1	timebaseActionNumber
1.3.6.1.4.1.1206.4.2.1.5.3.1.2	timebaseAscPattern
1.3.6.1.4.1.1206.4.2.1.5.3.1.3	timebaseAscAuxillaryFunction
1.3.6.1.4.1.1206.4.2.1.5.3.1.4	timebaseAscSpecialFunction
1.3.6.1.4.1.1206.4.2.1.5.4	timebaseActionStatus
1.3.6.1.4.1.1206.4.2.1.6	preempt
1.3.6.1.4.1.1206.4.2.1.6.1	maxPreempts
1.3.6.1.4.1.1206.4.2.1.6.2	preemptTable
1.3.6.1.4.1.1206.4.2.1.6.2.1	preemptEntry
1.3.6.1.4.1.1206.4.2.1.6.2.1.1	preemptNumber

continued . . .

Table D.2: List of ASC Object Identifiers

. . . continued

NTCIP 1201 OID	Object Name
1.3.6.1.4.1.1206.4.2.1.6.2.1.2	preemptControl
1.3.6.1.4.1.1206.4.2.1.6.2.1.3	preemptLink
1.3.6.1.4.1.1206.4.2.1.6.2.1.4	preemptDelay
1.3.6.1.4.1.1206.4.2.1.6.2.1.5	preemptMinimumDuration
1.3.6.1.4.1.1206.4.2.1.6.2.1.6	preemptMinimumGreen
1.3.6.1.4.1.1206.4.2.1.6.2.1.7	preemptMinimumWalk
1.3.6.1.4.1.1206.4.2.1.6.2.1.8	preemptEnterPedClear
1.3.6.1.4.1.1206.4.2.1.6.2.1.9	preemptTrackGreen
1.3.6.1.4.1.1206.4.2.1.6.2.1.10	preemptDwellGreen
1.3.6.1.4.1.1206.4.2.1.6.2.1.11	preemptMaximumPresence
1.3.6.1.4.1.1206.4.2.1.6.2.1.12	preemptTrackPhase
1.3.6.1.4.1.1206.4.2.1.6.2.1.13	preemptDwellPhase
1.3.6.1.4.1.1206.4.2.1.6.2.1.14	preemptDwellPed
1.3.6.1.4.1.1206.4.2.1.6.2.1.15	preemptExitPhase
1.3.6.1.4.1.1206.4.2.1.6.2.1.16	preemptState
1.3.6.1.4.1.1206.4.2.1.6.2.1.17	preemptTrackOverlap
1.3.6.1.4.1.1206.4.2.1.6.2.1.18	preemptDwellOverlap
1.3.6.1.4.1.1206.4.2.1.6.2.1.19	preemptCyclingPhase
1.3.6.1.4.1.1206.4.2.1.6.2.1.20	preemptCyclingPed
1.3.6.1.4.1.1206.4.2.1.6.2.1.21	preemptCyclingOverlap
1.3.6.1.4.1.1206.4.2.1.7	ring
1.3.6.1.4.1.1206.4.2.1.7.1	maxRings
1.3.6.1.4.1.1206.4.2.1.7.2	maxSequences
1.3.6.1.4.1.1206.4.2.1.7.3	sequenceTable
1.3.6.1.4.1.1206.4.2.1.7.3.1	sequenceEntry
1.3.6.1.4.1.1206.4.2.1.7.3.1.1	sequenceNumber
1.3.6.1.4.1.1206.4.2.1.7.3.1.2	sequenceRingNumber
1.3.6.1.4.1.1206.4.2.1.7.3.1.3	sequenceData
1.3.6.1.4.1.1206.4.2.1.7.4	maxRingControlGroups
1.3.6.1.4.1.1206.4.2.1.7.5	ringControlGroupTable
1.3.6.1.4.1.1206.4.2.1.7.5.1	ringControlGroupEntry
1.3.6.1.4.1.1206.4.2.1.7.5.1.1	ringControlGroupNumber
1.3.6.1.4.1.1206.4.2.1.7.5.1.2	ringControlGroupStopTime
1.3.6.1.4.1.1206.4.2.1.7.5.1.3	ringControlGroupForceOff
1.3.6.1.4.1.1206.4.2.1.7.5.1.4	ringControlGroupMax2
1.3.6.1.4.1.1206.4.2.1.7.5.1.5	ringControlGroupMaxInhibit
1.3.6.1.4.1.1206.4.2.1.7.5.1.6	ringControlGroupPedRecycle

continued . . .

Table D.2: List of ASC Object Identifiers

. . . continued

NTCIP 1201 OID	**Object Name**
1.3.6.1.4.1.1206.4.2.1.7.5.1.7	ringControlGroupRedRest
1.3.6.1.4.1.1206.4.2.1.7.5.1.8	ringControlGroupOmitRedClear
1.3.6.1.4.1.1206.4.2.1.7.6	ringStatusTable
1.3.6.1.4.1.1206.4.2.1.7.6.1	ringStatusEntry
1.3.6.1.4.1.1206.4.2.1.7.6.1.1	ringStatus
1.3.6.1.4.1.1206.4.2.1.8	channel
1.3.6.1.4.1.1206.4.2.1.8.1	maxChannels
1.3.6.1.4.1.1206.4.2.1.8.2	channelTable
1.3.6.1.4.1.1206.4.2.1.8.2.1	channelEntry
1.3.6.1.4.1.1206.4.2.1.8.2.1.1	channelNumber
1.3.6.1.4.1.1206.4.2.1.8.2.1.2	channelControlSource
1.3.6.1.4.1.1206.4.2.1.8.2.1.3	channelControlType
1.3.6.1.4.1.1206.4.2.1.8.2.1.4	channelFlash
1.3.6.1.4.1.1206.4.2.1.8.2.1.5	channelDim
1.3.6.1.4.1.1206.4.2.1.8.3	maxChannelStatusGroups
1.3.6.1.4.1.1206.4.2.1.8.4	channelStatusGroupTable
1.3.6.1.4.1.1206.4.2.1.8.4.1	channelStatusGroupEntry
1.3.6.1.4.1.1206.4.2.1.8.4.1.1	channelStatusGroupNumber
1.3.6.1.4.1.1206.4.2.1.8.4.1.2	channelStatusGroupReds
1.3.6.1.4.1.1206.4.2.1.8.4.1.3	channelStatusGroupYellows
1.3.6.1.4.1.1206.4.2.1.8.4.1.4	channelStatusGroupGreens
1.3.6.1.4.1.1206.4.2.1.9	overlap
1.3.6.1.4.1.1206.4.2.1.9.1	maxOverlaps
1.3.6.1.4.1.1206.4.2.1.9.2	overlapTable
1.3.6.1.4.1.1206.4.2.1.9.2.1	overlapEntry
1.3.6.1.4.1.1206.4.2.1.9.2.1.1	overlapNumber
1.3.6.1.4.1.1206.4.2.1.9.2.1.2	overlapType
1.3.6.1.4.1.1206.4.2.1.9.2.1.3	overlapIncludedPhases
1.3.6.1.4.1.1206.4.2.1.9.2.1.4	overlapModifierPhases
1.3.6.1.4.1.1206.4.2.1.9.2.1.5	overlapTrailGreen
1.3.6.1.4.1.1206.4.2.1.9.2.1.6	overlapTrailYellow
1.3.6.1.4.1.1206.4.2.1.9.2.1.7	overlapTrailRed
1.3.6.1.4.1.1206.4.2.1.9.3	maxOverlapStatusGroups
1.3.6.1.4.1.1206.4.2.1.9.4	overlapStatusGroupTable
1.3.6.1.4.1.1206.4.2.1.9.4.1	overlapStatusGroupEntry
1.3.6.1.4.1.1206.4.2.1.9.4.1.1	overlapStatusGroupNumber
1.3.6.1.4.1.1206.4.2.1.9.4.1.2	overlapStatusGroupReds

continued . . .

Table D.2: List of ASC Object Identifiers

. . . continued

NTCIP 1201 OID	**Object Name**
1.3.6.1.4.1.1206.4.2.1.9.4.1.3	overlapStatusGroupYellows
1.3.6.1.4.1.1206.4.2.1.9.4.1.4	overlapStatusGroupGreens

Appendix E

Arada Locomate Application Parameter List

Table E.1: Common Options

Parameter	**Description**
-m	Mac Address [xx:xx:xx:xx:xx:xx]
-s	Service Channel
-b	TxPkt Channel
-w	Service Type [Provider/User]
-t	Message Type [BSM/PVD/RSA/ICA/SPAT/MAP/TIM]
-e	Security Type [Plain/Sign/Encrypt]
-D	Certificate Attach Interval in millisec should be in multiple of packet delay
-l	Output log filename, (specify path ending with / for pcap format)
-P	Prefex of certificate files)
-o	Tx/Rx Options [TXRX/NOTX/NORXALL/NORX/TXRX-UDP/NOTXRX]
-X	Logging Options [TXRXLOG/TXLOG/RXLOG/NOLOG]
-g	sign certificate type [certificate/digest_224/digest_256/certificate_chain]
-p	BSM Part II Packet interval (n BSM Part I messages)
-v	Path history number [2 represents BSM-PH-2, 5 represents BSM-PH-5]
-	Vehicle_Type (value as per DE_VehicleType)
-y	psid value (any decimal value)
-d	packet delay in millisec
-q	User Priority 0/1/2/3/4/5/6/7
-j	txpower in dBm
-M	Model Deployment Device ID
-T	Temporary ID control (1 = random, 0 = fixed upper two bytes)
-S	Safety Supplement (wsmp-s) ¡0:disable / 1:enable¿
-L	Vehicle Length in cm
-W	Vehicle Width in cm
-r	data rate 0.0, 3.0, 4.5, 6.0, 9.0, 12.0, 18.0, 24.0, 27.0, 36.0, 48.0,54.0mbps
-n	no argument, and selects no gps device available
-f	Type xml or csv for logging in XML or CSV format. Type pcaphdr for only pcap header logging and pcap for full packet logging
-F	frameType for TIM Packet 0-unknown(default) 1-advisory 2-roadSignage 3-commercialSignage

continued . . .

Table E.1: Common Options

. . . continued

Parameter	Description
-A	Active Message Status
-B	Port Address for RSU receive from UDP Server
-R	Repeat rate for WSA frame (Number of WSA per 5 seconds) Repeatrate is included in WSA-Header only if enabled from /proc/wsa_repeatrate_enable
-G	Repeat rate for TA frame (Number of TA per 5 seconds) TA is available only if TA channel [-c option] is given
-I	IP service Enable 1= enable 0 = disable
-O	Timeout for receiving udp data , in seconds

Table E.2: Provider Options

Parameter	Description
-z	Service Priority
-a	Service Channel Access [1:Alternating, 0:Continuous]
-c	Specify Channel Number to Transmit TA
-i	TA Channel Interval [1:cch int, 2:sch int]

Table E.3: User Options

Parameter	Description
-u	User Request Type [1:auto, 2:unconditional(not wait for WSA from provider), 3:none]
-x	Extended Access ¡0:alternate /1:continuous¿

Table E.4: Default Values

Parameter	Values
-m	Mac Address [00:00:00:00:00:00]
-s	Service Channel - 172
-b	TxPkt Channel - 172
-w	Service Type - User/Provider
-u	User Request Type - [1:auto]
-x	Extended Access - 0
-c	TA disabled - 0
-p	BSM Part II Interval - 10 packets

continued . . .

Table E.4: Default Values

. . . continued

Parameter	**Values**
-k	Vehicle Type - 0 (not available)
-v	PathHistory Number - 4 (PathHistory Set 4)
-t	Message Type - BSM
-e	Security Type - Plain
-l	Output log filename, - NULL
-o	Tx/Rx Options - TXRX
-X	Logging Options - NOLOG
-g	Sign Certificate Type - certificate
-y	psid value - 32
-d	packet delay in millisec - 100
-f	Format Type PCAP
-F	Frame Type 0
-r	data rate(mbps) - 3.0
-A	Active Message Status - 0 (Disable)
-C	Config File Name for active message - '/var/activemsg.conf'
-B	Port Address for RSU receive from UDP Server - 0
-R	Repeat rate for WSA frame - 50
-G	Repeat rate for TA frame - 0
-j	txpower(dBm) - 14
-q	User Priority 2
-S	Safety Supplement (wsmp-s) - 0 (disabled)
-E	Certificate change request flag -0)
-L	Vehicle Length(cm) - 0
-W	Vehicle Width(cm) - 0
-D	Certificate Attach Interval in millisec - 500
-M	Model Deployment Device ID = 1
-T	Temporary ID control = 1 (random 4 bytes)
-I	IP service Enable = 0
-O	Timeout for receiving udp data = 10 seconds