THE INTERACTIVE HARDWARE-IN-LOOP SIMULATION SYSTEM
FOR TRAFFIC CONTROL SYSTEM DEVELOPMENT

Li Sheng

Thesis

Approved: Accepted:

Advisor
Dr. Ping Yi

Dean of the College
Dr. George K. Haritos

Faculty Reader
Dr. Ernian Pan

Dean of Graduate School
Dr. George R. Newkome

Department Chair
Dr. Wieslaw Binienda

Date
ABSTRACT

With the development of modern computer technology, applications are developed to manage the transportation system intelligently and improve the efficiency of the system. Traffic controllers are the core components in the system to collect and process traffic data. However, the existing systems can only evaluate traffic control algorithms fixed in the controller, and they do not generally allow data communication between the controller and a simulated environment for the purpose of plan evaluation and system optimization.

The modern controllers can do more complicated work such as real-time control and data communication in support of decision analysis, but the development of such control systems becomes very difficult, especially over control algorithms which must be tested in the real world. It is therefore desirable to develop a flexible platform for the design and update of the control algorithm with data communication support.

This research develops and tests a new platform for real-time traffic control. The architecture includes both the software and hardware design. It consists of two components, the traffic simulation environment and the traffic controller. The traffic simulation environment is realized in the CORISM simulation software developed by Federal Highway Administration (FHWA). The traffic controller is developed with the PIC micro-controller, a single-chip computer manufactured by Microchip Incorporation.
A CORSIM Run-Time Extension program is developed to bridge the simulation environment and the controller and provides a communication data path between them. Using the designed platform, the developers can download their algorithm easily to the traffic controller and perform online debug for analysis and improvement. Five control algorithms are tested for the platform and discussed in this research as an example of application. The main benefits of the research includes the integration of traffic simulation environment and the traffic controller, flexible algorithms update, and the ability to collect multiple types of traffic data which cannot be obtained in the existing systems.
ACKNOWLEDGEMENTS

I would like to express my sincere thanks to my advisor, Dr. Ping Yi, for his advice and guidance throughout this research. His encouragement, support and understanding in these years will be a valuable memory in my life.

Also I would like to thank my committee members, Dr. Ernian Pan and Dr. Craig Menzemer for their time and support.

In addition, I would like to express my thanks to my wife, Zhi Kuang and my little son, Keyu. I really appreciate their love and support in my life.
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF TABLES</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>viii</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ix</td>
</tr>
</tbody>
</table>

CHAPTER

I. INTRODUCTION .................................................................................. 1

II. LITERATURE REVIEW ........................................................................ 5

III. INTERACTIVE HARDWARE-IN-LOOP SIMULATION SYSTEM DESIGN....... 8

   3.1 General Description.................................................................... 8

   3.2 Hardware Design......................................................................... 9

      3.2.1 PC side............................................................................... 9

      3.2.2 Microcontroller side......................................................... 9

      3.2.3 Communication method...................................................... 15

   3.3 Software Design......................................................................... 19

      3.3.1 General Description.......................................................... 19

      3.3.2 Personal Computer Side Software Design............................ 20

         3.3.2.1 TSIS Software Introduction.......................................... 21

         3.3.2.2 Application Software Architecture............................... 25

         3.3.2.3 Interface Module Programming................................... 25

            3.3.2.3.1 General Description of DLL................................. 25
3.3.2.3.2 Objects and associated member functions and variables……… 26

3.3.2.4 Communication Module……………………………………… 42

3.3.2.5 Software Integration………………………………………… 49

3.3.3 PIC Side Software Design……………………………………… 49

3.3.3.1 Software Architecture……………………………………… 49

3.3.3.2 Control Algorithm Design…………………………………… 50

3.3.3.3 Initialization Module and Communication Module Design……… 53

IV. SYSTEM TEST ……………………………………………………………… 57

4.1 General Description…………………………………………………. 57

4.2 Green Rest…………………………………………………………… 61

4.3 Gap Out ……………………………………………………………… 63

4.4 Queue & Gap Out………………………………………………… 66

4.5 Volume Density Control…………………………………………… 69

4.6 Dilemma Zone Control…………………………………………… 72

V. CONCLUSIONS AND FUTURE RESEARCH…………………………… 77

REFERENCES ……………………………………………………………. 78

APPENDICES ……………………………………………………………. 81

APPENDIX A. SOURCES CODE FOR PC SIDE……………………….. 82

APPENDIX B. SOURCES CODE FOR PIC18F452……………………… 154
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Classes</td>
<td>27</td>
</tr>
<tr>
<td>3.2</td>
<td>CNetwork Class</td>
<td>28</td>
</tr>
<tr>
<td>3.3</td>
<td>CNode Class</td>
<td>29</td>
</tr>
<tr>
<td>3.4</td>
<td>CLink Class</td>
<td>32</td>
</tr>
<tr>
<td>3.5</td>
<td>CLane Class</td>
<td>33</td>
</tr>
<tr>
<td>3.6</td>
<td>CDetector Class</td>
<td>34</td>
</tr>
<tr>
<td>3.7</td>
<td>CSignalState Class</td>
<td>35</td>
</tr>
<tr>
<td>3.8</td>
<td>CInteger Class</td>
<td>37</td>
</tr>
<tr>
<td>3.9</td>
<td>CInteger Class</td>
<td>37</td>
</tr>
<tr>
<td>3.10</td>
<td>Member Functions of Collection</td>
<td>39</td>
</tr>
<tr>
<td>3.11</td>
<td>WIN32 Communication Events</td>
<td>48</td>
</tr>
<tr>
<td>3.12</td>
<td>Register Configuration for PIC</td>
<td>53</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Typical Traffic Control System</td>
</tr>
<tr>
<td>2.1</td>
<td>Architecture for conventional HILS</td>
</tr>
<tr>
<td>3.1</td>
<td>Architecture for System Design</td>
</tr>
<tr>
<td>3.2</td>
<td>Architecture of PIC18F452</td>
</tr>
<tr>
<td>3.3</td>
<td>Components of PICDEM 2 PLUS</td>
</tr>
<tr>
<td>3.4</td>
<td>General ICD 2 Debug System</td>
</tr>
<tr>
<td>3.5</td>
<td>RS-232 voltage levels</td>
</tr>
<tr>
<td>3.6</td>
<td>RS-232 DTE to DCE connection</td>
</tr>
</tbody>
</table>
| 3.7   | Synchronous and Asynchronous transmission  
   (a) Synchronous Transmission (b) Asynchronous Transmission | 18 |
| 3.8   | Software Architecture | 19 |
| 3.9   | Architecture of CORSIM RTE Application | 23 |
| 3.10  | Architecture for PC side software | 25 |
| 3.11  | Link Configurations | 30 |
| 3.12  | Signal Sequence and Pointer Application | 31 |
| 3.13  | Signal Code for Input File | 36 |
| 3.14  | Flowchart for INIT Function | 40 |
| 3.15  | Flowchart for JMAIN Function | 41 |
3.16 Flowchart for JEXIT Function................................................................. 42
3.17 Flowchart for Communication Module................................................... 43
3.18 DCB Structure....................................................................................... 45
3.19 Architecture for PIC Side Software........................................................ 50
3.20 Flowchart of General Control Algorithm.................................................52
3.21 Flowchart for Communication...............................................................55
4.1 Diagrams for Basic Traffic Control..........................................................59
4.2 Traffic Network for test.............................................................................60
4.3 Detector Configuration for Green Rest................................................... 61
4.4 Flowchart for Green Rest.........................................................................62
4.5 Detector Configuration for Gap Out.........................................................64
4.6 Flowchart for Gap Out.............................................................................65
4.7 Detector Configuration for Queue&Gap Out.......................................... 67
4.8 Flowchart for Queue&Gap Out............................................................... 68
4.9 Detector Configuration for Volume Density Control...............................70
4.10 Flowchart for Volume Density Control..................................................71
4.11 Detector Configuration for Dilemma Zone Control.................................73
4.12 Flowchart for Dilemma Zone Control.....................................................74
CHAPTER I
INTRODUCTION

Traffic controller is the most important component in the traffic management system, especially for the Intelligent Transportation System (ITS), where the traffic controller works as the brain for the entire system.

Basically, controllers collect data from the detector system, which are preinstalled in the roadway, and then process these data with a specific algorithm to make the decision about when and how to switch from one phase to another phase. Both hardware and software are needed to make the controller running effectively. The software is related with the control strategy while the hardware is associated with the technology utilized to fulfill the strategy. So there are lots of categories for the traffic controller in practical application.

In terms of the strategy, there are fixed-time control, Actuated Control, and Adaptive Control, and Real-Time Adaptive Control. In terms of the technology, there are mechanical Controller, Electrical Mechanical Controller, and Microcomputer Controller. The associated hardware, Software and communication methods are different among the different controllers. No matter what kind of controller, it needs the traffic data from the real world to make decision and send the decision out to the traffic lights. Therefore, all follow the following schema in Figure 1.1.
With the development of traffic control application, the more complicated system is desired to work in the network level other than the single intersection control, which leads to the development of Intelligent Transportation System. The ITS is an advanced technology developed in recent years, which manages entire traffic system in a view of network-wide. The individual controller is identified as one node ion the network, where the grouped controllers are classified as one sub network. In this way, entire network can be divided into several levels. Each level performs different tasks. Usually, the lower levels focus on their local optimization of the traffic effectiveness, while the upper levels consider the system performance as the objective. Each controller is one fundamental element in the system. The controllers performing control function on specific traffic node or link are called Slave Controller, whereas the controllers cooperating the slave Controllers are called Master Controller, which focus on the system performance.
From above, we can see that traffic control system is a very complicated system. It is very difficult to develop a new system. So usually, the researchers use the simulation methods to evaluate their algorithms before they are put into practical application, which makes the development process more effective. Even in the practical work for the traffic engineers, they evaluate their timing plan and ITS schema to make sure that they work correctly in the field. Currently, generally speaking, there are two categories of simulation methods. One is the Pure Software Simulation (PSS); the other is the Hardware-In-Loop Simulation (HILS). The difference between them is that there is no hardware (controller) involved in the process in the PSS. Currently, the Traffic Software Integrated System (TSIS) and VISSIM are the most popular software used for traffic system simulation. In this research, TSIS software is applied to do the simulation [3]. When this software is used alone to perform simulation, it is called PSS. Although the user designed control algorithm can be evaluated by the PSS, this method is not related directly with the physical controllers, which may results in some problems when these algorithms are applied in the real world, because the true controller responds to the collected data differently by the different speed and some hardware associated problems cannot be found in the pure-virtual evaluation. Therefore, the physical controller should be included into the development procedure, which is called HILS. However, it is not only the conventional controller since it should be capable to connect with the simulation software to evaluate the embedded algorithm in the controller. In other words, except the general Traffic Controller functions, it should provide an interface to integrate itself with the existing evaluation software. One advantage of HILS is the separation of simulation environment and the controller. The control algorithm stands alone in the physical
controller, which eliminates the mutual affection when they are in the same computer. Other benefits will be introduced in the literature review section. It is because the HILS has some advantages over the pure software methods, more and more researchers use HILS in their research.
CHAPTER II
LITERATURE REVIEW

The HILS (Hardware-In-Loop Simulation) has been used by researchers for more than 10 years to integrate the traffic controller into the microscopic simulation process. In this period, there has been an evolution in the application methods of HILS. Perrin [6] developed a system to evaluate the SCOOT algorithm by connecting the controller with the CORSIM simulation system. Kwon [7] applied HILS to evaluate the adaptive control algorithm. Bullock [9, 10] designed a system to evaluate the intersection control system. Jeong [13] used HILS to evaluate the traffic network system performance. Byrne [15] implemented HILS to decide the transit priority. Engelbrecht [19] used HILS to set the controller parameters. All of these applications were based on the Controller Interface Device to connect the controller with the simulated environment. In this device, data are exchanged between the simulation and controller components through a shared memory. The common features for all the HILS are that they offer the advantages for some conditions where the PSS (Pure Software Simulation) cannot or less accurately evaluate the controller performance.

However, all the applications above only focus on the evaluation of the existing algorithm built in the controller; they cannot evaluate different configurations of the controller in terms of new control strategies and data feeds. The algorithm built in
controller cannot be updated and many useful types of data in the system cannot be collected.

Basically, the conventional Controller Interface Devices (CID) has the architecture shown in Figure 2.1. The PC component is the computer in which the simulation software is running. The Communication1 component is the data exchange path between simulation process and the Controller Interface Device (CID) component, while the Communication2 is for the data transfer between CID and physical controller.

![Figure 2.1. Architecture for conventional HILS](image)

The most important component in the HILS is the interface between the simulation software and the hardware, CID. The first non-commercial CID is developed in 1997 at the Louisiana States University for the Federal Highway Administration under a contract from ITT System & Science Corporation [9]. The Louisiana State University CID (LSU-CID) provides an interface to allow traffic engineers and analysts to replace the emulated logic of the CORSIM simulation model with a real controller based on the NEMA TS1 standard. The RS-232 serial communication works as the interface between the PC and

6
the real controller. The TransLink Research Center at the Texas Transportation Institute developed another non-commercial CID for use in its Roadside Equipment Laboratory (REL) [18]. The parallel communication interface is utilized to connect the PC and the controller, which provides faster speed for data communication. The most recently developed CID by the National Institute for Advanced Transportation Technology (NIATT) at the University of Idaho can be used for large scale HILS application, which is based upon the universal serial bus (USB) interface to send the detector information, the start time, and duration to the CIDs [10].

Each CID introduced above owns its respective advantage in certain area. The common problem for them is that they cannot transfer multiple types of data in the same time according to the user’s application, such as the speed, activation time, deactivation time, occupancy time, etc. Most of them can only provide the presence state of detectors for the controller. In other words, the data customization cannot be performed for those CIDs, which is a critical task for the intelligent traffic control system. Therefore, the more flexible system is desired for the development of traffic controller.
CHAPTER III
INTERACTIVE HARDWARE-IN-LOOP SIMULATION SYSTEM DESIGN

3.1 General Description

To make the HILS system more effective, the hardware is integrated in the system by a more flexible method. Both sides of system, the TSIS simulation environment and the Traffic controller can be assigned different algorithm, which makes the controller development system more flexible and powerful. The architecture of system is described in Figure 3.1.

![Figure 3.1 Architecture for System Design](image)

In this figure, the TSIS software is running in the Personal Computer (PC) side. The user-designed algorithm is built in the Dynamical Linking Library (DLL) program in c++ language, which can be associated with the specific network to perform the simulation. The DLL program will catch the detector data from the simulated network and send them to the PIC Microcontroller (PIC) side by the RS-232 Communication port, where the user-designed control algorithm is manipulated to process the received data and makes a decision for phase switch. The result is sent back to the PC side through RS-232.
Communication port and control the simulated intersection. The conventional CID is integrated into the physical controller, which eliminates the necessity of two microprocessors to handle the communication and control process. This will reduce the cost for the development.

3.2 Hardware Design

3.2.1 PC side

The capability to run TSIS software is the basic requirement for choosing the hardware in PC side [3]. Additionally, because there is a data exchange between the TSIS processes and the Control Algorithm processes in micro-controller side, which is called inter-processor communication, so a RS-232 port is necessary for the PC. We choose the following configurations for PC side:

- Intel Pentium 4 2.6GHz Processor
- 128 MB RAM
- 80 GB hard driver
- 2 RS-232 Serial Communication Ports

These configurations are enough according to the requirements of the manual [3].

3.2.2 Microcontroller side

PIC 18F452, which is manufactured by Microchip Corporation, is chosen as the microcontroller in the application. From this section, we call the Microcontroller as PIC for simplification. PIC possesses the following features [27]:

- Up to 10 MHz clock (PLL to 40 MHz)
• 4 independent timers
• 16 interrupt sources
• 25 mA sink/source on I/O pins
• Watchdog timer & brownout detector
• Low voltage operation
• I/O expansion via SPI port
• I²C Master and Slave mode
• Addressable USART module: RS-232 and RS-485
• Parallel Slave Port (PSP) module
• Math hardware support for add, subtract, and multiply
• Free tools from Microchip and others
• 40 pin DIP Layout
• Program Memory: 16k words
• RAM: 1536 Bytes

The most attractive feature is that the program memory is flash memory, which means you can erase it at any time to update your program only by the way in that you write the data into the hard driver. This feature improves the development efficiency a lot, which enables the user to do online debug. The other features make it possible to add more function into design when the controller is put into practical application.

The general architecture of this chip is shown in Figure 3.2. Basically, there are 3 components, the I/O ports, the Timer & Communication Component, and the CPU Component. I/O Component is composed by 5 groups of multiple function I/O pins, PORTA, PORTB, PORTC, PORTD and PORTE where each pin except RA6 and RB4
possesses at least two functions, which makes the entire architecture simpler. The Timer & Communication Component includes the 4 timers, their associated function modules, 10 Bits A/D Converter, and the communication modules. The CPU Component includes the Clock generation module, Calculation Modules and the Addressing Function Module. All the components work together to perform the assigned tasks.

Figure 3.2 Architecture of PIC18F452
The PICDEM 2 PLUS shown in Figure 3.3 is used as the target board to develop the control algorithm and communication program, which works as the traffic controller. One PIC microcontroller is included in this board. The clock for this board is set as 4 MHz.

The PICDEM 2 Plus board has the following hardware features:

- 18, 28 and 40-pin DIP sockets. (Although three sockets are provided, only one device may be used at a time.)
- On-board +5V regulator for direct input from 9V, 100 mA AC/DC wall adapter or 9V battery, or hooks for a +5V, 100 mA regulated DC supply.
- RS-232 socket and associated hardware for direct connection to an RS-232
interface.

- In-Circuit Debugger (ICD) connector.
- 5 KΩ pot for devices with analog inputs.
- Three push button switches for external stimulus and Reset.
- Green power-on indicator LED.
- Four red LEDs connected to PORTB.
- Jumper J6 to disconnect LEDs from PORTB.
- 4 MHz canned crystal oscillator.
- Unpopulated holes provided for crystal connection.
- 32.768 kHz crystal for Timer1 clock operation.
- Jumper J7 to disconnect on-board RC oscillator (approximately 2 MHz).
- 32K x 8 Serial EEPROM.
- LCD display.
- Piezo buzzer.
- Prototype area for user hardware.
- Microchip TC74 thermal sensor.

In this design, the 40-pin micro-controller is applied. RS-232 port is used to exchange data with the PC. The other module can be used for further applications.

After the micro-controller is selected, the corresponding development tools should be chosen to make it work. The In-circuit Debuggers, MPLAB ICD 2 online debugger is utilized as the development tool for this design. This all-in-one debugger/programmer solution is a low cost, real-time debugger and programmer for selected PIC micro-controllers. Using Microchip Technology's proprietary In-Circuit Debug functions,
programs can be downloaded, executed in real time and examined in detail using the debug functions of MPLAB. Watch variables and breakpoints can be set from symbolic labels in C or assembly source code, and single stepping can be done through C source line, assembly code level, or from a mixed C source and generated assembly level listing. MPLAB ICD 2 can also be used as a development programmer for supported PIC micro-controllers. The general development system is shown in Figure 3.4

Features of ICD 2

- USB (Full Speed 2 M bits/s) & RS 232 interface to HOST PC
- Real time background debugging
- MPLAB IDE compatible
- Built in over voltage/short circuit monitor
- Firmware upgradeable from PC
- Totally enclosed
- Supports low voltage to 2.0 volts. (2.0 to 6.0 range)
- Diagnostic LED’s (Power, Busy, Error)
- Reading/Writing memory space and EEDATA areas of target micro-controller
- Programs configuration bits
• Erase of program memory space with verification
• Peripheral freeze-on-halt stops timers at breakpoints

The USB is chosen for the connection between PC and ICD 2 for convenience. Then the corresponding development software should be installed in the Host PC. MPLAB Integrated Development Environment (IDE) is selected as the development environment, which is a free, integrated toolset for the development of embedded applications employing Microchip's PIC micro-controllers. MPLAB IDE runs as a 32-bit application on MS Windows, which is easy to use and includes a host of free software components for fast application development and super-charged debugging. MPLAB IDE also serves as a single, unified graphical user interface for additional Microchip and third party software and hardware development tools.

3.2.3 Communication method

RS-232 protocol is one of the most popular techniques for communication between the computers and external instruments [28,29], which utilizes serial communication protocol where one bit is sent along a line, at a time. This differs from parallel communications that sends one or more bytes, at a time. The main benefit of serial communication over parallel communication is that a single wire is needed to transmit and another to receive data. In addition, in terms of convenience, it is better to use serial communication because most of Personal Computers and the microcontrollers have the built in serial communication module. So the serial communication is the best choice for
this project. The 9-pin configuration is used. The electrical characteristics are introduced in Figure 3.5.

![Figure 3.5 RS-232 voltage levels](image)

The electrical characteristics of RS-232 define the minimum and maximum voltages of logic ‘1’ and ‘0’. Logic ‘1’ ranges from –3 V to –25 V, but will typically around –12 V. Logic ‘0’ ranges from 3 V to 25 V, but will typically around +12 V. Any voltages between –3 V and +3 V have an indeterminate logic state. If there are no pulses on line, the voltage level is equivalent to a high level, -12 V. A voltage level of 0 V at the receiver is interpreted as a line break or a short circuit.

In Figure 3.6, the basic architecture for RS-232 is described. In this case, DCE is the micro controller and DTE is the PC. The handshaking method is used in the communication. CTS, RTS, DTR, and DSR signal are used to synchronize the communication process. In a typical communication session, the "terminal," which is the PC in this case, raises the DTR line to indicate that it is ready to begin a transmission.
The DCE, the PIC microcontroller will raise the DSR line when it is turned on to indicate that it is ready. The computer will then raise the RTS line when it wants to transmit and the microcontroller will respond on the CTS line. The computer then begins to transmit.

To make the data exchange successful, the protocol cannot be neglected, which helps to ensure that every transmission reaches its destination and that each side can understand the messages sent to it. There are two kinds of protocols used for serial communication, synchronous and asynchronous protocols. The synchronous protocol needs the common clock in both sides to decide when to read and send each bit, so the clock line is used to send clock signal, while asynchronous doesn’t need the clock line because it is not necessary for common clock in both sides. For synchronous protocol, the

![Diagram of RS-232 DTE to DCE connection](image)

Figure 3.6 RS-232 DTE to DCE connection
transmitter sends bits on clock’s falling edge and receiver reads bits on clock’s rising edge. But for asynchronous protocol, the transmitter uses an internal clock to determine when to sends each bit and the receiver detects the falling edge of Start bit, and then uses its internal clock to read the following bits near their centers. The difference is shown in Figure 3.7.

(a). Synchronous Transmission

(b). Asynchronous Transmission

Figure 3.7 Synchronous and Asynchronous transmission
In this research, the asynchronous protocol is used. We apply 1 STOP BIT, 8 DATA BITS, and no PARITY BIT. The PARITY BIT is just used as the simple form of error checking. All these configurations are set in the Device Configuration Bits (DCB) structure of WIN32 API application. The software design of the communication will be introduced in the following section.

3.3 Software Design

3.3.1 General Description

![Software Architecture Diagram]

Figure 3.8 Software Architecture

Fundamentally, there are 4 components for the application software, the Interface (PC side), RS-232 Communication (PC side), RS-232 Communication (PIC side), and Control Algorithm (PIC side), which are shown in Figure 3.8. For the CORSIM Run Time Extension (RTE) application, we only show the CORSIM Server Component built in the TSIS CORSIM Simulation software, because it directly communicates with the user developed application RTE. The other components are not demonstrated here, because this section only focuses on the application architecture. All those components will be discussed in detail in the TSIS Software section.
The Dynamical Linking Library (DLL) programming method is used to program the RTE application. Both Interface and RS-232 Communication (PC side) Components are integrated into the RTE DLL in order to communicate with the TSIS and PIC processes in the same time. Interface Component is responsible to communicate with the TSIS CORSIM simulation software to collect simulated detector data and send the signal switch decision back to the simulated environment. RS-232. So the simulated software can evaluate the control algorithm embedded in PIC. On the other hand, the RS-232 Communication Component manages all the communication process between PC processes and the PIC processes and performs the timing synchronization. The RS-232 Communication (PIC side) Component performs the same task but only for PIC side. The communication is established through the work of above both components. The control logic is embedded in the PIC side as the Control Algorithm Component, which collects data from the communication component and processes these data by the built model. The decision is sent back to TSIS environment through the communication port to control the simulated network. Until now, the system architecture is presented completely. The following sections will give the detail on each component.

3.3.2 Personal Computer Side Software Design

Basically, there is some software necessarily installed before the application software is developed. In order to run the TSIS software, the WINDOWS 2000 operating system is installed in the computer, because our TSIS version is for WINDOWS system. The TSIS 5.1 and ITRAF 4.0 are installed in the host computer. The Microsoft Visual C++ 6.0 is utilized as the development environment for the C++ programming. In
addition to all above software installed in the host PC, it necessary to install the software to develop the PIC side application. In this case, we choose the MAPLAB IDE software and the HI-TECH PICC-18 C Compiler for embedded system by HI-TECH Software Company. The former one provides an integrated development environment for the embedded system application, the latter for the C language development of the embedded system. All the software and hardware are ready for the application development. The detail will be given in the following sections.

3.3.2.1 TSIS Software Introduction

TSIS software is widely used in the traffic simulation, which owns very powerful functions to support the diversities of simulated situations. Because in this research, we use TSIS and CORSIM to simulate all the traffic process, it is necessary to give a brief description of its architecture and functions.

The FHWA’s TSIS is an integrated development environment that enables users to conduct traffic operations analysis. This system is built through a component architecture and works as a toolbox that provides a method to define and manage traffic projects, define traffic networks and create inputs for traffic simulation analysis, execute traffic simulation models, and interpret the results of those model. Besides the built tools, the users can build their own tools into the TSIS environment, which is the way used in this research.

Basically, there are 5 components in the TSIS, which cooperate to fulfill all the functions.
• Tshell is the graphical user interface for the TSIS integrated development environment, which provides a Project view that enables users to manage TSIS projects and works as the container for the traffic analysis tools.

• CORSIM simulation consists of an integrated set of two microscopic simulation models (NETSIM and FRESIM) that represent the entire traffic environment as a function of time. NETSIM represents surface-street traffic and FRESIM represents freeway traffic. Microscopic simulations model the movements of individual vehicles, which include the influences of driver behavior. These models provide a simulated environment where the efficiency of traffic control algorithm can be evaluated by the generated output from the simulation.

• TRAFED is a graphical user interface-based editor that allows users to create and edit traffic networks and simulation input for the CORSIM model. But it is not used in this research because of its complication and unfriendly interface.

• TRAFVU is a state-of-the-art graphics post-processor for CORSIM microscopic traffic simulation system, which displays the traffic networks, animates simulated traffic flow operations, animates and displays simulation output measures of effectiveness, and displays user-specified input parameters for simulated network objects.

• TSIS Text Editor is a standard text editor specifically for the CORSIM TRF file format, by which the users can modify the input files in the TSIS environment based upon the specific network design and the input files can be shown very straightforward.
In this case, the most important component we care about is the CORSIM function that allows user to add user-designed algorithm into the original environment, which is called CORSIM Run Time Extension (RTE). For the real-time control application, RTE is utilized instead of the built CORSIM algorithm. It brings lots of flexibilities to perform specific functions and provides a more powerful function for the user [4]. The basic architecture for CORSIM RTE is shown in Figure 3.9.

Figure 3.9 Architecture of CORSIM RTE Application

In the figure, the block arrows represent Component Object Model (COM) interfaces and the thin arrows represent standard dynamic-link library (DLL) interfaces. The CORSIM Driver Component works as the TSIS tool running the CORSIM simulation and interfacing to the TShell user interface. This component enables the users to control CORSIM execution and output processing, and manages all user-supplied
RTEs. For each simulation time step, the CORSIM Server calls a series of functions within CORSIM to drive the simulation event loop. When an RTE is present and enabled, the CORSIM Server also calls exported functions of the RTE based on messages it receives from CORSIM. CORSIM sends RTE messages every time step at a specific point in its processing logic. The Server also calls the RTE’s initialization function during CORSIM initialization and the RTE’s exit function at the end of the simulation. This interface is illustrated in the figure by the path marked “1”. TSIS also provides an interface (CORWin) that enables an RTE to send messages to the CORSIM Server and to send text messages to be displayed by the CORSIM Driver. This interface is illustrated in the figure by the path marked “2”. In addition, the RTE can directly access the data structures in CORSIM because they have been exported as shared memory, which not only permit the RTE extract information from the simulation but also enables the RTE to control different aspects of the simulation. The shared memory interface is illustrated in the figure by the path marked “3”. The RTE may call exported CORSIM functions. This interface is illustrated in the figure by the path marked “4”.

For the general RTE application, there are 3 modules in the DLL program, Initialization Function, Main Function, and Exit Function, which provide the necessary entry points for the operation of CORSIM RTE application. The Initialization Function performs the initialization of the environmental variables for entire network, which is called first and only once by CORSIM. It is the same for the Exit Function only called after all the simulation process is finished. However, the Main Function is called every time step to update all the variables and send them back to the CORSIM environment
3.3.2.2 Application Software Architecture

In Figure 3.10, the architecture for PC side software includes 2 modules shown by solid line rectangular. The Interface Module is mainly responsible for bridging the Communication Module with the CORSIM Server, which transfers the CORSIM-based network and variables into the DLL-based network and variables so that the CORSIM variables can be processed in DLL and the variables returned from DLL can be applied into the CORSIM simulation, which is an integration process of the user-defined program and the existed CORSIM environment. The Communication Module sends the data from the Interface Module to the hardware controller and receives data from it to return them to the Interface Module. Therefore, the hardware controller is integrated into the CORSIM environment to replace the virtual controller.

![Figure 3.10 Architecture for PC side software](image)

3.3.2.3 Interface Module Programming

3.3.2.3.1 General Description of DLL

The Dynamical Linking Library (DLL) is an effective programming method when the program structure is complicated, where lots of functions are called by the main function to fulfill multiple tasks. Usually, the DLL files include the function files that can work individually. One benefit for DLL is that the DLL source files are not included in the program project when the project is compiled and linked. Instead, the DLL source
files are compiled and linked separately, which will generate the DLL file and library file. Only the library file for this DLL is included, which is a very small file including the information for DLL calling so that those functions in DLL are only called in the Run-Time instead of always staying in the memory. This way reduces the memory occupation of the executed program. Another advantage makes the program more flexible, or portable because the user can add some function modules to the existed software but not care about the structure inside it. If the existed software provides the interface files for DLL, you even can add the function modules without providing the library file for the compiling and linking of existed software, which means that it is not necessary to compile and link with the source files of existed software. This is the story for the DLL in this case, where the TSIS provides an interface for the DLL calling. So the user only need compile and link DLL individually with the provided interface libraries.

3.3.2.3.2 Objects and associated member functions and variables

The Object-Oriented C++ programming is used for the software design [28]. Basically, there are 8 classes to describe all the data structure, which is shown in Table 3.1.
Table 3.1 Classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNetwork</td>
<td>Include the functions to build network and process network attributes and the variables to describe the network</td>
</tr>
<tr>
<td>CNode</td>
<td>Include the functions to set the signal state and the variables to describe the node attributes</td>
</tr>
<tr>
<td>CLink</td>
<td>Include the functions to process the detector data and set the signal state and variables to describe the link attributes</td>
</tr>
<tr>
<td>CLane</td>
<td>Include the variables to describe the attributes and the detectors in this lane</td>
</tr>
<tr>
<td>CDetector</td>
<td>Include the variables to describe the detector information and attributes</td>
</tr>
<tr>
<td>CSignalState</td>
<td>Include the functions to set the signal state and the variables to describe this signal and its attributes</td>
</tr>
<tr>
<td>CInteger</td>
<td>Used to transfer the Integer Type to Binary Type in detector data process</td>
</tr>
<tr>
<td>CBinarySequence</td>
<td>Include variables to store the binary sequence in detector data process</td>
</tr>
</tbody>
</table>
In the Table 3.2 through Table 3.9, the detail for each class is given, which includes the important functions and variables for each class.

Table 3.2 CNetwork Class

<table>
<thead>
<tr>
<th>CNetwork</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Public Functions</strong></td>
</tr>
<tr>
<td>ReadTrafFile( void )</td>
</tr>
<tr>
<td>GetNodes( FILE* pFile )</td>
</tr>
<tr>
<td>FindNode( int id )</td>
</tr>
<tr>
<td>GetLinks( FILE* pFile)</td>
</tr>
<tr>
<td>FindLink( int up, int dn )</td>
</tr>
<tr>
<td>GetLinkCorsimId( int upnode, int dnnode )</td>
</tr>
<tr>
<td>UpdateNodeSignalStates( void )</td>
</tr>
<tr>
<td>GetDetectors(FILE* pFile)</td>
</tr>
<tr>
<td>GetDetectorCorsimId( CDetector* pDetector )</td>
</tr>
<tr>
<td>CreateLanes( FILE* pFile )</td>
</tr>
<tr>
<td>CreateSignalStates (void)</td>
</tr>
</tbody>
</table>

| **Private Functions** |
| ReadTRFLine( FILE* pFile, char* pszLine ) | Read into each line of TSIS input file |

| Attributes |
| m_LinkList | List of all links in this network |
| m_NodeList | List of all nodes in this network |

The main function of CNetwork class is to build the network from the input text file, including the links, lanes, nodes, detectors, and signal states. In addition, the relationship between the build network and corresponding CORSIM network is established to perform the data communication between them. The CNode, CLink and CLane objects are the components of CNetwork objects. So the CNetwork objects can call any functions.
in these components and implement all the attributes of them to fulfill the task for the signal control.

Table 3.3 CNode Class

<table>
<thead>
<tr>
<th>Attributes (variables)</th>
<th>CNode</th>
</tr>
</thead>
<tbody>
<tr>
<td>m_id</td>
<td>ID for this node</td>
</tr>
<tr>
<td>m_CORSIMInternalID</td>
<td>CORSIM ID for this node</td>
</tr>
<tr>
<td>m_xPos</td>
<td>X position for this node in the network</td>
</tr>
<tr>
<td>m_yPos</td>
<td>Y position for this node in the network</td>
</tr>
<tr>
<td>m_Link1</td>
<td>User defined approach 1 for this node</td>
</tr>
<tr>
<td>m_Link2</td>
<td>User defined approach 2 for this node</td>
</tr>
<tr>
<td>m_Link3</td>
<td>User defined approach 3 for this node</td>
</tr>
<tr>
<td>m_Link4</td>
<td>User defined approach 4 for this node</td>
</tr>
<tr>
<td>m_Link5</td>
<td>User defined approach 5 for this node</td>
</tr>
<tr>
<td>m_typeOfControl;</td>
<td>Type of control for this node (internal)</td>
</tr>
<tr>
<td>m_duration[12]</td>
<td>Duration of each phase, maximum 12 intervals</td>
</tr>
</tbody>
</table>

The CNode class is used to build the node in the program, which constructs the signal states for the node on each specific link. There are 5 links for one node defined in CORSIM software, which can be customized by the user. In this research, the m_Link1 to m_Link4 are defined as Figure 3.11. There is no m_Link5, because we only have 4 approaches.
CNode objects can call the functions associated with the CSignalState object to perform the signal update because CSignalState object is a component of CNode objects. The signal states can be updated by the user through setting the signal codes dynamically, where the codes are preset in a signal sequence, which is shown in Figure 3.12. The user change the signal codes by setting the pointer pointing to the position of this sequence, which is convenient for user only need to load the sequence one time and just change the pointer to perform any signal states transition. The only thing need to be addressed is that the enough green and red signal states should be loaded in case the pointer goes to the wrong place.
Figure 3.12 Signal Sequence and Pointer Application
### Table 3.4 CLink Class

<table>
<thead>
<tr>
<th><strong>Public Functions</strong></th>
<th><strong>Application</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ProcessDetectors( void );</td>
<td>Process detectors in this link</td>
</tr>
<tr>
<td>ConvertToBinary( int nValue );</td>
<td>Convert integer to binary</td>
</tr>
<tr>
<td>FindLane( int corsimId );</td>
<td>Find lane with specific corsimId</td>
</tr>
<tr>
<td>ComputeTravelTime( void );</td>
<td>Get travel time for this link</td>
</tr>
<tr>
<td>CreateSignalStates( void );</td>
<td>Create signal state for this link</td>
</tr>
<tr>
<td>SetSDCCode( void );</td>
<td>Set signal code</td>
</tr>
<tr>
<td>AdjustSignalState( void );</td>
<td>Adjust the signal state to allow for protected lefts for the green ball</td>
</tr>
</tbody>
</table>

**Attributes (variables)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int m_id</td>
<td>ID for this link</td>
</tr>
<tr>
<td>int m_CorsimId</td>
<td>CORSIM ID for this link</td>
</tr>
<tr>
<td>CNode* m_upnode</td>
<td>Up node for this link</td>
</tr>
<tr>
<td>CNode* m_dnnode</td>
<td>Down node for this link</td>
</tr>
<tr>
<td>CNode* m_thrunode</td>
<td>Through next node</td>
</tr>
<tr>
<td>CNode* m_leftnode</td>
<td>Left turn next node</td>
</tr>
<tr>
<td>CNode* m_rightnode</td>
<td>Right turn next node</td>
</tr>
<tr>
<td>CTypedPtrList&lt;CPtrList,CLane*&gt; m_listOfLanes</td>
<td>List of lanes for this link</td>
</tr>
<tr>
<td>CTypedPtrList&lt;CPtrList,CDetector*&gt; m_listOfDetectors</td>
<td>List of detectors for this link</td>
</tr>
<tr>
<td>CTypedPtrList&lt;CPtrList,CSignalState*&gt; m_signalStates</td>
<td>Sequence of signal state</td>
</tr>
<tr>
<td>int m_length</td>
<td>Length of this link</td>
</tr>
<tr>
<td>int m_numOfFullLanes</td>
<td>Number of full lanes</td>
</tr>
<tr>
<td>int m_numOfLeftTurnBays</td>
<td>Number of left turn bays</td>
</tr>
<tr>
<td>int m_numOfRightTurnBays</td>
<td>Number of right turn bays</td>
</tr>
<tr>
<td>int m_lengthOfLeftBay</td>
<td>Length of left bay</td>
</tr>
<tr>
<td>int m_lengthOfRightBay</td>
<td>Length of right bay</td>
</tr>
<tr>
<td>int m_freeFlowSpeed</td>
<td>Free speed for this link</td>
</tr>
<tr>
<td>float m_travelTime</td>
<td>Travel time for this link</td>
</tr>
<tr>
<td>int m_leftMovementPercent</td>
<td>Percentage of left turn</td>
</tr>
<tr>
<td>int m_thruMovementPercent</td>
<td>Percentage of through traffic</td>
</tr>
<tr>
<td>int m_rightMovementPercent</td>
<td>Percentage of right turn</td>
</tr>
<tr>
<td>CString m_channelCode[7]</td>
<td>Channel code for this link</td>
</tr>
<tr>
<td>int m_NumOfDetectors</td>
<td>Number of detectors</td>
</tr>
<tr>
<td>int m_code[12]</td>
<td>Signal code for each phase</td>
</tr>
<tr>
<td>int m_offset</td>
<td>Offset for this link regard to the signal sequence</td>
</tr>
<tr>
<td>POSITION m_pos</td>
<td>Pointer in the signal sequence</td>
</tr>
<tr>
<td>CLink* m_opposingLink</td>
<td>Pointer for the opposing link</td>
</tr>
<tr>
<td>int m_opposingID</td>
<td>ID for opposing link</td>
</tr>
</tbody>
</table>
The CLink class is responsible for building the links in the network and associated traffic structures. Each link owns respective signal states, lane and detector configurations so the function members in those CSignalState, CDetector, and CLane objects can be called by the CLink objects. The corresponding links, signal states and detectors in CORSIM software are connected with their counterparts in the program to establish the data exchange between them. The detector data is extracted from the CORSIM detectors through the detectors built in each lane of each link in the program.

Table 3.5 CLane Class

<table>
<thead>
<tr>
<th>Attributes (variables)</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>int m_id</td>
<td>ID for this lane</td>
</tr>
<tr>
<td>CString m_type</td>
<td>Full lane or bay</td>
</tr>
<tr>
<td>CLink* m_Link;</td>
<td>The Link this lane belong to</td>
</tr>
<tr>
<td>int m_length;</td>
<td>Length of this lane</td>
</tr>
<tr>
<td>int m_leftMovementPercent;</td>
<td>Percentage of left turn for this lane</td>
</tr>
<tr>
<td>int m_thruMovementPercent;</td>
<td>Percentage through traffic for this lane</td>
</tr>
<tr>
<td>int m_rightMovementPercent;</td>
<td>Percentage of right turn for this lane</td>
</tr>
<tr>
<td>CTypedPtrList&lt;CPtrList,CDetector*&gt; m_DetectorList;</td>
<td>Detectors in this lane</td>
</tr>
<tr>
<td>CTypedPtrList&lt;CPtrList,CDetector*&gt; m_StopBarDetectorList</td>
<td>Detectors in Stop Bar in this lane</td>
</tr>
</tbody>
</table>

CLane class works as the container for detectors, where each detector is specified according to its position in the lane. Besides detector information, the host link and the corresponding traffic information are described in the class. There is no member function for this class. All its attributes can be used by the class, where CLane object is a member variable.
Table 3.6 CDetector Class

<table>
<thead>
<tr>
<th>Attributes (variables)</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>int m_id</td>
<td>ID for this detector</td>
</tr>
<tr>
<td>int m_length</td>
<td>Length of this detector</td>
</tr>
<tr>
<td>CString m_type</td>
<td>Presence or pulse detector</td>
</tr>
<tr>
<td>CLink* m_Link</td>
<td>The link this detector belong to</td>
</tr>
<tr>
<td>int m_distanceFromDownstreamNode</td>
<td>Distance to the stop bar</td>
</tr>
<tr>
<td>CLane* m_Lane</td>
<td>The lane this detector belong to</td>
</tr>
<tr>
<td>int m_CorsimId</td>
<td>CORSIM ID for this detector</td>
</tr>
<tr>
<td>int m_count</td>
<td>Count of this detector</td>
</tr>
<tr>
<td>BOOL m_state</td>
<td>State of this detector: activated=TRUE or not=FALSE</td>
</tr>
<tr>
<td>float m_activationTime</td>
<td>Duration of activation for this detector</td>
</tr>
</tbody>
</table>

CDetector class collects all the information for the specific detector and its host lane object, where the detector is installed. All the information is included in Table 3.6. The detector data is extracted through the host lane for the specific detector and its host link, because there is no member function in CDetector and CLane objects. The detector only can be processed by the function ProcessDetectors in the CLink and the results are stored in the CDetector object.
Table 3.7 CSignalState Class

<table>
<thead>
<tr>
<th>Public Functions</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>void SetSDCODE( void )</td>
<td>Set the CORSIM signal state code except amber</td>
</tr>
<tr>
<td>int GetSDCODE( int actualCode )</td>
<td>Interpret the ITRAf signal code to CORSIM code</td>
</tr>
<tr>
<td>void SetAMBSPC( void )</td>
<td>Set the amber code</td>
</tr>
<tr>
<td>int GetAMBSPC1( int nextActualCode )</td>
<td>Get amber code from the signalCode and m_actualCode for previous state as green</td>
</tr>
<tr>
<td>int GetAMBSPC3( int nextActualCode )</td>
<td>Get amber code for previous state as green right turn</td>
</tr>
<tr>
<td>int GetAMBSPC4( int nextActualCode )</td>
<td>Get amber code for previous state as green left turn</td>
</tr>
<tr>
<td>int GetAMBSPC5( int nextActualCode )</td>
<td>Get amber code for previous state as stop sign</td>
</tr>
<tr>
<td>int GetAMBSPC6( int nextActualCode )</td>
<td>Get amber code for previous state as green diagonal</td>
</tr>
<tr>
<td>int GetAMBSPC7( int nextActualCode )</td>
<td>Get amber code for previous state as green through</td>
</tr>
<tr>
<td>int GetAMBSPC8( int nextActualCode )</td>
<td>Get amber code for previous state as green right and left turn</td>
</tr>
<tr>
<td>int GetAMBSPC9( int nextActualCode )</td>
<td>Get amber code for previous state as green through and right turn</td>
</tr>
</tbody>
</table>

**Attributes (variables)**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int signalCode</td>
<td>Code from input TRAF file card 36</td>
</tr>
<tr>
<td>int m_actualCode</td>
<td>Code used to actually set the AMBSPC</td>
</tr>
<tr>
<td>int SDCODE</td>
<td>Code that can be used in CORSIM</td>
</tr>
<tr>
<td>int AMBSPC</td>
<td>Code that can be used in CORSIM for amber state</td>
</tr>
<tr>
<td>CLink* m_Link</td>
<td>Link for this signal</td>
</tr>
</tbody>
</table>

The CSignalState class performs the functions for translating the signal code from the input file to the code that can be identified by the CORSIM software. The signalCode attribute is the code read from the 36 cards in the input TRAF file, which ranges from 1 – 9 shown in Figure 3.13, while m_actualCode is the code to get the correct amber state AMBSPC. Both are necessary for each state interval.
However, in our design, we bypass all the set signal from the original input TRAF file in the controlled intersection no matter it is set as fixed-time or actuated control. We use the member function CreateSignalStates of CNetwork class to build the signal
sequence and never change the sequence, only change the pointer “m_pos” pointing to the specific position in the signal sequence, which is mentioned in CNode class. This way makes the design more effective since it is unnecessary to create the signal sequence each time to update the signal state, which saves lots of computation time.

Table 3.8 CInteger Class

<table>
<thead>
<tr>
<th>Attributes (variables)</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>int data</td>
<td>Extracted detector state as an integer</td>
</tr>
</tbody>
</table>

CInteger class is used to extract the detector information. For each detector in CORSIM, the detector information is included in a data structure. There are 2 detector states, active and inactive, which is defined in CORSIM by individual time step, but is shown by an integer in Table 3.8.

Table 3.9 CBinarySequence Class

<table>
<thead>
<tr>
<th>Attributes (variables)</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTypedPtrList&lt;CPtrList, CInteger*&gt; m_sequence</td>
<td>The list for the detector state as a time series extracted from the CInteger data</td>
</tr>
</tbody>
</table>

However, in the application, only the binary data can define the 2 states, so the class CBinarySequence in Table 3.9 is utilized to translate the integer to binary data. One bit in the binary data corresponds to the state in one time step.

One special data structure named Collection is used in this design intensively, so its detail is given here. The Microsoft Foundation Class (MFC) Library provides the Collection class to manage groups of objects. There are two types of classes in implementation, one created from C++ templates and the other not created from
templates. Each type is characterized by its “shape” and by the types of its elements, where the shape refers to the way the objects are organized and stored by the collection. MFC provides 3 basic collection shapes: lists, arrays, and maps. In this research, the templates-based Collection classes are utilized through the shape of lists. The list class provides an ordered, nonindexed list of elements, implemented as a doubly linked list. The benefit for this application is that because a list has a “head” and “tail”, the adding or removing elements from the head and tail, or inserting or deleting elements in the middle, or retrieving elements is very effective. The basic syntax is given in the following expression:

```
CTypedPtrList<class BASE_CLASS, class TYPE> NAME
```

CTypedPtrList means that it is a template-based list Collection. The BASE_CLASS describes the list type used and TYPE for the objects managed by Collection. NAME is the variable name under this class type or object name of this class. The member functions of Collection are used to implement all these functions, which are listed in Table 3.10. When these member functions are applied, the POSITION type pointer should be defined to point to the specific position in the Collection, which works as the iteration variable.
Table 3.10 Member Functions of Collection

<table>
<thead>
<tr>
<th>Member Functions</th>
<th>Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>AddHead</td>
<td>Add element in the head</td>
</tr>
<tr>
<td>AddTail</td>
<td>Add element in the tail</td>
</tr>
<tr>
<td>GetAt</td>
<td>Go to the specific position</td>
</tr>
<tr>
<td>GetHead</td>
<td>Go to the head</td>
</tr>
<tr>
<td>GetNext</td>
<td>Go to the next element</td>
</tr>
<tr>
<td>GetPrev</td>
<td>Go to the previous element</td>
</tr>
<tr>
<td>GetTail</td>
<td>Go to the tail</td>
</tr>
<tr>
<td>RemoveHead</td>
<td>Remove the element in the head</td>
</tr>
<tr>
<td>RemoveTail</td>
<td>Remove the element in the tail</td>
</tr>
<tr>
<td>SetAt</td>
<td>Set new element in the specific position</td>
</tr>
</tbody>
</table>

In this design, the Collection is used to organize the links and nodes in one network, the lanes, detectors, and signal states in one link, the detectors in one lane, and the bits in one binary sequence for detector states. It is very effective when the user need to retrieve the data under the Collection just using the predefined member functions of Collection.

There are 3 main functions in Interface Module, which perform the interfacing task. The INIT function is described in Figure 3.14, which is called only once when the simulation starts. It builds the network from the input ITRAF file and constructs the initial situation for the traffic and signal states, where the signal sequence is established and never changed. The JMAIN function is called in each time step, which collects the
detector information and updates the signal states by moving the state sequence pointer. More importantly, it provides an interface for Communication Module to exchange data with external controller, namely sending detector data and receiving signal state decision. The general architecture for JMAIN is described in Figure 3.15. For different detector configuration, there is a little difference in the program, but the program architecture is the same.

![Figure 3.14 Flowchart for INIT Function](image-url)
In each time step, the CORSIM calls the JMAIN function in the DLL to implement the external control algorithm. The detector data is collected through searching the specific lane of one specific link for each controlled node. When the simulation is finished, the JEXIT function is called to clear all the memory and exit from the DLL.
program as shown in Figure 3.16. All the 3 functions reside in the unctrl.cpp source code in Appendix A.

![Flowchart for JEXIT Function](image)

Figure 3.16 Flowchart for JEXIT Function

### 3.3.2.4 Communication Module

WIN32 Application Program Interface (API) is a group of functions providing the interface for users to call the operating system services. Especially, for hardware control like the communication control and I/O control, APIs have to be called if users want to control them, because the WIN 32 operating system doesn’t allow the users to implement them directly for safe reason. In this Communication Module, since RS-232 port has to be implemented to transfer data, so the WIN32 APIs related to it are called in the application. The basic idea is described in Figure 3.17.
First the communication port is initialized according to the specific application, such as the baud rate and the hand shaking methods, which are included in the data structure DCB. Then the PC starts the communication after the data is ready. The EscapeCommFunction function is used to raise the DTR line in order to notify the external device that the PC communication port is ready to transmit and receive the data. In the following step, the data is sent to the external device. The PC is keeping checking the event trigger to see if there is data back from the external device. If data is coming, PC reads the buffer to get the data. In this case, two handshaking functions are applied, the EscapeCommFunction and WaitCommEvent, which synchronize entire

Figure 3.17 Flowchart for Communication Module
communication process to guarantee correct data receiving. The detail is described in the following.

In Windows 2000 system, the communication port is treated as a file, which can be created by the CreateFile function. In terms of communication, it means that the specific communication port is connected and selected. When you call CreateFile on any of communication ports, the system checks to make sure that the port is not already in use by some other process, and if it is, it returns the appropriate error code. If the port is available, one handle pointing to this port is returned to guarantee the exclusive access to it. The parameters and constants used in the call to CreateFile here are the same as those used when opening a file. The following parameters to CreateFile must be set as specified for communication port:

- Set the accessMode function to allow read and write access
- Set shareMode to 0 to indicate exclusive access
- Set the create parameter to OPEN_EXISTING, since the COM port already exists
- Set attributes to FILE_ATTRIBUTES_NORMAL
- Set templateFile to 0

After the connection is ready, communications parameters are set. Each port retains whatever settings it had from its last use, so it is important to set all relevant parameters to new values to make sure that you are using known settings. The GetCommState function is used to retrieve the existing settings, the structure DCB, which contains 28 fields and is described in Figure 3-18.
typedef struct _DCB {
    DWORD DCBlength; // size of (DCB)
    DWORD BaudRate; // current baud rate
    DWORD fBinary: 1; // binary mode, no EOF check
    DWORD fParity: 1; // enable parity checking
    DWORD fOutxCtsFlow: 1; // CTS output flow control
    DWORD fOutxDsrFlow: 1; // DSR output flow control
    DWORD fDtrControl: 2; // DTR flow control type
    DWORD fDsrSensitivity: 1; // DSR sensitivity
    DWORD fTXContinueOnXoff: 1; // XOFF continues Tx
    DWORD fOutX: 1; // XON/XOFF out flow control
    DWORD fInX: 1; // XON/XOFF in flow control
    DWORD fErrorChar: 1; // enable error replacement
    DWORD fNull: 1; // enable null stripping
    DWORD fRtsControl: 2; // RTS flow control
    DWORD fAbortOnError: 1; // abort rds/wrs on error
    DWORD fDummy2: 17; // reserved
    WORD wReserved; // not currently used
    WORD XonLim; // transmit XON threshold
    WORD XoffLim; // transmit XOFF threshold
    BYTE ByteSize; // number of bits/byte, 4-8
    BYTE Parity; // 0-4 = no, odd, even, mark, space
    BYTE StopBits; // 0,1,2 = 1, 1.5, 2
    char XonChar; // Tx and Rx XON character
    char XoffChar; // Tx and Rx XOFF character
    char ErrorChar; // error replacement character
    char EofChar; // end of input character
    char EvtChar; // received event character
} DCB;

Figure 3.18 DCB Structure

The new values for configuration are set according to application and sent to the communication port using SetCommState function, which is set as following:

- BaudRate = 9600
- ByteSize = 8;
- Parity = NOPARITY;
- StopBits = ONESTOPBIT
In addition to the port configuration in DCB structure, to guarantee that the reading and writing process work properly, the COMMTIMEOUTS structure is also needed to be set. These values are also retained from use to use, so it is important to set them anew if you are expecting certain behavior. These values give you very subtle control over the timeout behavior of your program during reads and writes, and guarantee that your program does not stall in the event of communication problems. For example, a ReadFile operation told to return with 100 bytes will stall until it finds 100 bytes in the input buffer. If there is a communication error, however, 100 bytes will never arrive, and the program gets stuck. Using timeouts, you can instruct the program to return immediately from any ReadFile operation regardless of the number of bytes found, or to return after a specified number of milliseconds.

The ReadIntervalTimeout value controls timeouts on the interval between characters. For example, you might set it to 1000 to indicate that a ReadFile operation should return if two characters are spaced out by more than one second. If you set ReadIntervalTimeout to 0, no interval timeouts are used. If you set it to MAXDWORD as application in this research, and if the other two read intervals are set to 0, then the system returns immediately from any ReadFile operation with whatever the input buffer contains.

The ReadTotalTimeoutMultiplier and ReadTotalTimeoutConstant values let you set a timeout based on the number of bytes specified in the ReadFile statement. If you ask to read 100 bytes, for example, the following equation determines the timeout value.

Total timeout value = 100 \times \text{ReadTotalTimeoutMultiplier}
After the specified number of milliseconds have elapsed, the ReadFile function returns, regardless of how many bytes it has actually read. The same process is used for the Write multiplier and constant in an outgoing direction. Once the timeout structure has been set up, it is passed to the system with the SetCommTimeouts function.

With all configuration set, the PC is ready to begin communicating with the external device. It sets the DTR line to indicate that it is ready using the EscapeCommFunction function. The SETDTR operation properly informs the external device that the PC is ready and is about to begin transmission. All of the other signals are handled automatically by the communication port and are not normally manipulated manually. In the next step, WriteFile command is used to send the data to the external device, which is wrapped in an array structure that is pointed by a pointer. To make the communication more effective, the Event Trigger method is applied, which provides a way to detect certain communications events to trigger the ReadFile function. These events can improve the responsiveness of the program, which are listed in Table 3.10.
EV_BREAK | Break was sent in the input
---|---
EV_CTS | The CTS line changed
EV_DSR | The DSR line changed
EV_ERR | An error was detected: CE_FRAME, CE_OVERRUN, and CE_RXPARITY are possible
EV_RING | The modem sensed a ring
EV_RLSD | The DCD line changed
EV_RXCHAR | A character was received
EV_RXFLAG | The special event character (as indicated by the EvtChar field in the DCB) was received
EV_TXEMPTY | The last character of the last WriteFile operation was sent

In this case, we utilize the EV_RXCHAR event to sense if there is data in the buffer. If data is in the buffer, the ReadFile function will read the data, if not, the program will wait until the data is coming. The Event Trigger method is implemented by first setting the specific event and then waiting for the occurrence of this event. The corresponding WIN32 APIs are the SetCommMask function and WaitCommEvent functions, which are detailed in the reference [28] for the set parameters. The SetCommMask function sets the specific event waiting for and the WaitCommEvent function will efficiently wait for the specified event to occur. Once the event occurs, the ReadFile function will read the data into the received data structure for further applications. At last, the EscapeCommFunction function is called again to clear the DTR line, which means that this communication process is finished, namely that the data is sent out and the results are received from the external device. Then the results are returned to the Interface Module.
for further application. The SerialComm function resides in the upctrl.cpp source code with the functions of Interface Module.

3.3.2.5 Software Integration

The Interface Module and Communication Module are integrated in the DLL program, which provides the calling entry point for the CORSIM software. As mentioned in Section 3.3.2.1, there are 3 functions in the Interface Module, where the Main Function works for the integration. After the data is obtained from CORSIM, it calls the Communication Module and sends the data to it to activate the communication process. However, the process doesn’t come back to the Main Function immediately until the data is received from the PIC side controller. The received data is brought by the Communication Module to the Main Function. Therefore, one-step simulation is accomplished.

3.3.3 PIC Side Software Design

3.3.3.1 Software Architecture

The software in micro-controller side is composed by 3 modules, the Initialization Module, the Control Algorithm Module, and the Communication Module. The architecture is shown in Figure 3.19.
The PIC is initialized first to set the corresponding registers for the specific functions. Then the program goes to the Communication Module to see if there is any received data in the buffer from the Communication Module of PC side. If no data, the program will stay in this module until the data comes in to trigger the process to go to Control Algorithm Module, where the data is processed and the result is returned to the Communication Module, which transfers the result to the Communication Module in PC side. This is an endless loop to keep the control process forever unless the PIC is turn off.

3.3.3.2 Control Algorithm Design

Control Algorithm Module receives detector data from Communication Module and processes those data according to the specific rule to make a decision for the signal state transition, which is sent back to the Communication Module. Regard to variety of control strategies, the Control Algorithm Module is different, which will be discussed in detail in the Experiment Section. The general architecture is shown in Figure 3.20.
simulation, the green is always set to start from Side Street, which is initialized in the
INIT function in the PC side. The returned signal code from this program controls the
transition of the signal state in the simulation. In this research, the signal code in this
program just provides 2 values, 1 for green of Main Street and 2 for green of Side Street.
However, user can customize the signal code depending on the specific situation. There
are 3 important variables in the program, mainStTime, sideStTime and clock. All the 3
variables are global, which means they keep consistent to all the functions in the program
and can be changed by all the functions. So all of them can be updated in each time step
and synchronize the simulation process. The input variables for this function are clock
and detector data, which are analyzed to get the correct signal code. The signal code is
initialized as “0”. Then the clock is compared with the time the Side Street should be
served by green. If it is smaller, the green can be kept in Side Street to satisfy the
minimum green by not updating signal code. If bigger, the signal code is or not updated
depending on the specific control strategy and the mainStTime is updated too. For each
time step, this process is repeated until the simulation is terminated.

Generally, there are several common issues for the control algorithm design. The
Clock is critical because all the control is time-based so it should be kept continuous for
one simulation process. In each time step, the detector information is obtained through
the Communication Module and the algorithm processes them to get the returned
decision variables. There is some difference between single detector information
(presence) and multiple detector information (presence, activation time, speed,
deactivation time, etc.) input. To consider both situations, one array is used as the input to
include all the detector information by a specific order so that they can be extracted correctly in the application.

Figure 3.20 Flowchart of General Control Algorithm

mainStTime-- time for green serving Main Street
sideStTime-- time for green serving Side Street
clock-- time step for the simulation
min2—minimum green for Side Street
3.3.3.3 Initialization Module and Communication Module Design

The initialization of registers corresponding to the RS-232 Serial Communication:

TXSTA=0x06;
RCSTA=0x90;
SPBRG=0x19;
TRISC=0x80;
TXEN=1;

The details for each bit of these registers are listed in Table 3.11.

Table 3.12 Register Configuration for PIC

<table>
<thead>
<tr>
<th>Register</th>
<th>Bit7</th>
<th>Bit6</th>
<th>Bit5</th>
<th>Bit4</th>
<th>Bit3</th>
<th>Bit2</th>
<th>Bit1</th>
<th>Bit0</th>
</tr>
</thead>
<tbody>
<tr>
<td>TXSTA</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>SPEN</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RCSTA</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Bit7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>RC5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TRISC</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
There are two ways for the design, polling method and interrupt method. Polling means that the program always work in the front side to check if there is a state change indicating the specific event happened, if no, the program is waiting in this point until this event occurs. The CPU cannot do anything except waiting here. For the Interrupt method, the interrupt program works in the background. If the specific event occurs, the interrupt program is trigger immediately to respond to this event. Both methods have respective advantages. Polling method is more straightforward but not so effective, which usually used in the small-scale program, while the Interrupt method is more effective but more complicated. For this design, only the Communication Component need wait for an event. For the transmit side, the CPU need know if the transmitted data is ready, while for receive side, the CPU need know if there is the received data in the buffer. So the Polling method is utilized to make the program more straightforward and entire program focuses on the control algorithm.

The meaning of each register value is described as following:

- TXSTA: High-speed asynchronous mode, Tx disabled, 8-bit transmit.
- RCSTA: Serial port enabled, 8-bit reception, continuous receive, and disable address detect.
- SPBRG: 9600 baud for 2 MHz XTAL.
- TRISC: Rx input and Tx output.

The SPBRG is calculated by following equation for the asynchronous high-speed mode:

\[
\text{Baud Rate} = \frac{\text{Fosc}}{16(X+1)}
\]
X: value in the SPBRG

Fosc: clock frequency

In this case, Fosc=4 MHz, after calculation, we get X=19h (hex)

The flowchart for communication module is introduced in Figure 3.21.
First the registers are initialized to prepare for the communication. Then the data-receive-buffer is checked to see if there is data, which is implemented by checking the RCIF flag for the communication. If RCIF is set as “1”, the data is in the buffer. So program can proceed to get and process the data. If RCIF is “0”, the program will stay here until it is changed to “1”. The detector information is extracted from the received data and sent to the control algorithm to be processed. Finally, The returned signal code from control algorithm is sent back to PC after the TXIF flag is set to “1”, which means that data is already in the transmit-buffer and waiting for transmission. The process is repeated again to keep the communication continuous.
4.1 General Description

Generally, what the user should do for the RTE application follows the procedures below. At first, a traffic network should be built, where the user designed traffic control algorithm is applied. In this research, the ITRAF 4.0 [5] is used instead of the built-in component TRAFED. ITRAF 4.0 is the software designed specifically for network construction in TSIS environment, which is more powerful and convenient than TRAFED. In ITRAF 4.0, the network is constructed by the link and node, where the link represents a segment of roadway and node represents a point that connects two or more links in a network. Specifically in this research, we only consider the node as an intersection controlled by the designed algorithm. There are three kinds of node, the External Node, the Interface Node, and the Internal Node. The External Node is one node from which traffic enters and exits the network. The Interface Node is the node that separates two simulation models, the NETSIM and FRESIM. The Internal Node is the node completely embedded in one simulation model. The control strategy can only be assigned to the Internal Node, i.e. only Internal Node is under the user’s control. In the RTE application, we will bypass the built-in control of specific nodes and apply our algorithm on them to get the evaluation result. For each link, there are lots of properties
need to be specified, such as the turning movement, the lane configuration, the alignment, and the volume, etc.

The second and most important task is programming the DLL, which can fulfill the designed functions of the specific algorithm and provide the interfaces for CORSIM. The 3rd task is configuration of the simulation tool in the TSIS environment, where the user designed DLL is integrated into the CORSIM simulation system in order that the CORSIM Server can call it to complete all the simulation process. Finally, one project has to be created to perform the simulation in the TSIS environment, which transfers the input file, the created ITRAF network file into the text file that the CORSIM can recognize and associate with the specific control algorithm.

Before we start the tests, some traffic engineering terminologies should be introduced [2] to better understand the traffic control algorithm.

- **Cycle**: A signal cycle is one complete rotation through all of the indications provided. In general, every legal vehicular movement receives a “green” indication once within each cycle, although there are some exceptions to this rule.
- **Cycle length**: the cycle length is the time (in seconds) that it takes a signal to complete one full cycle of indications.
- **Interval**: An interval is a period during which none of the lights at a signalized intersection changes. At any signalized intersection, there are many intervals included in the signal cycle, including: change interval (yellow), clearance interval (all red), green interval, and red interval.
• Phase: a phase is the green interval plus the following change interval and clearance interval.

• Max-out: when the maximum green for one phase is reached, this service for this phase is terminated.

• Gap-out: when the gap for one phase is greater than the predefined gap, the service for this phase is terminated.

• Vehicle call: when a vehicle comes, it triggers the detectors and is registered as one vehicle call.

Figure 4.1 shows the timing operation for a basic two-phase or two-traffic movement pre-timed controller unit.

![Figure 4.1 Diagrams for Basic Traffic Control](image-url)
There are 5 scenarios we will test to verify the stability of our software and hardware design, including Green Rest, Gap Out, Queue Control, Dilemma Zone Control, and Volume Density Control.

There are different detector configurations for different applications depending on the control strategy. For each situation, the detector processing method has some difference in the program. The basic schema is described in Figure 4.2.

![Figure 4.2 Traffic Network for test](image)
In this figure, circles represent the nodes in this network, while the hexatone represent the source nodes, where the traffic starts and disappears from. The control algorithm is only applied on the Node 1.

4.2 Green Rest

In this case, Green is always set in the Main Street unless there is a vehicle call from the Side Street and a Max Out in Main Street in the meantime. If both are satisfied, the green will be switched to Side Street and serve a minimum green. The configuration for detectors and algorithm flowchart are shown respectively in Figure 4.3 and Figure 4.4.

![Diagram](image)

Figure 4.3 Detector Configuration for Green Rest
Green starts from Side Street

\( \text{clock} > \text{sideStTime} + \text{min2} \)

Yes

Green to Main Street
\( \text{mainStTime} = \text{sideStTime} + \text{min2} \)

Iterate on node

Node = 1

Yes

Process \( m_{\text{link3}} \) and \( m_{\text{link4}} \) to get detector information

Find specific detectors

No

\( \text{clock} > \text{mainStTime} + \text{max1} \)

Yes

Detectors triggered

Stay in Main Street
\( \text{sideStTime} = \text{clock} \)

Green Switched to Side Street

Exit

Figure 4.4 Flowchart for Green Rest
4.3 Gap Out

In this case, the green starts from Side Street. After the minimum green, it is switched to Main Street. The green will go back to Side Street only when there is a gap in Main Street greater than the predefined gap and the main street finishes its minimum green. Both conditions should be satisfied. The detector deactivation time is updated each time when CORSIM calls the RTE program and is sent to the PIC microcomputer to be processed. The result back from PIC is the switch code for the point in the signal sequence that is built in the initialization of the RTE DLL program. Detector configuration and algorithm flowchart are presented respectively in Figure 4.5 and Figure 4.6
Figure 4.5 Detector Configuration for Gap Out
Green starts from Side Street

clock\(>\)sideStTime+min2

Yes

Green to Main Street
mainStTime=sideStTime+min2

No

Iterate on node

No

Node=1

Yes

Process m_link2 to get detector information

No

clock\(>\)mainStTime+min1

Yes

deactiveTime\(>\)Gap

No

Stay in Main Street

sideStTime=\text{clock}

Yes

Green Switched to Side Street

No

Exit

Figure 4.6 Flowchart for Gap Out
4.4 Queue & Gap Out

The control logic is almost the same as Gap Out. The difference is that there are two detectors located in different distance before the stop bar. The detector configuration is shown in Figure 4.7. Depending on the activation time of detector, we will decide either the maximum green or minimum green set in Side Street. After the max green of Main Street, system will check the third detector in Side Street. If its activation time is bigger than a predefined time interval A, it means that the queue is reaching this detector. A longest minimum green period will be served in Side Street. If second detector is reached, a second longest green is served. If just first detector is reached, a third longest green will be assigned to Side Street.

First the detector Det in Main street is checked to decide if there is a Gap Out after its minimum green min1 is finished. If there is no Gap Out, the green will run to maximum green. In the same time, the third detector Det3 in Side street is checked to find if there are vehicles waiting in the queue. If the call exists, the longest green interval min2[0] will be served for Side Street. If not, the Det2 is checked. If there is a call, min2[1] will be assigned. If not, finally, the Det1 (on stop bar) is checked to decide if there is a call. If the queue reaches Det1, min2[2] will be served. If there is no call from

| sideStTime: | start time for Side Street |
| mainStTime: | start time for Main Street |
| min1: | minimum green for Main Street |
| min2: | minimum green for Side Street |
| deactiveTime: | measured vehicle gap |
| Gap: | predefined gap |
| clock: | time step for simulation |

Figure 4.6 Flowchart for Gap Out (continued)
three detectors, the green stays in the main street until there is a call from Side Street. Once the Side Street finishes its green, the green will be switched to Main Street immediately. The algorithm flowchart is presented in Figure 4.8.

Figure 4.7 Detector Configuration for Queue&Gap Out
Green starts from Side Street

\[ \text{clock} > \text{sideStTime} + \text{min2} \]

Yes

Green to Main Street
\[ \text{mainStTime} = \text{sideStTime} + \text{min2} \]

Yes

Iterate on node

No

Node = 1

Yes

Process m_link2 and m_link3 to get detector information

\[ \text{clock} > \text{mainStTime} + \text{min1} \]

Yes

\[ (\text{m_link2})_{\text{deactiveTime}} > \text{Gap} \]

No

\[ (\text{m_link3-det3})_{\text{activeTime}} > \text{A} \]

No

\[ (\text{m_link3-det2})_{\text{activeTime}} > \text{A} \]

No

\[ (\text{m_link3-det1})_{\text{activeTime}} > \text{A} \]

Yes

Green min2[0] to Side Street
\[ \text{sideStTime} = \text{clock} \]

No

Yes

Yes

Yes

Exit

Figure 4.8 Flowchart for Queue&Gap Out

68
4.5 Volume Density Control

In this algorithm, the control logic is changed dynamically based upon the traffic volume. The detector configuration is the same as the Gap-out shown in Figure 4.9, but the application is different. We can call it dynamical Gap Out. After system starts work, the deactivation time for the detector is checked to trace the fluctuation of traffic volume density, in fact, which reflects the gap between the vehicles.

Volume-density operation can be considered to be a more advanced form of full-actuated control. It has the ability to calculate the duration of minimum green based on actual demand (calls on red) and the ability to reduce the maximum allowable time between calls from passage time down to minimum gap. Reducing the allowable time between calls below the passage time will improve efficiency by being better able to detect the end of queued flow. The algorithm is shown in Figure 4.10.
Figure 4.9 Detector Configuration for Volume Density Control
Figure 4.10 Flowchart for Volume Density Control
4.6 Dilemma Zone Control

The concept is that the system always intends to gap out after the initial green for the main street, but to guarantee that the last vehicle passed the 2nd detector passes the intersection safely, the green should be extended to give enough time for this vehicle. The detector configuration is presented in Figure 4.11. The detector information helps to make the decision how long the green should be extended and when Gap Out is allowed. If after the extension time, there is no vehicles coming, Gap Out will happen. It is like a pure Gap Out, but safer in the application, since an extension time is given to make sure the higher density volume traffic get safe and effective service. The algorithm is shown in Figure 4.12.

Figure 4.10 Flowchart for Volume Density Control (continued)
Figure 4.11 Detector Configuration for Dilemma Zone Control
Green starts from Side Street
g=0;

clock>sideStTime+min2

Yes

No

1

Green to Main Street
mainStTime=sideStTime+min2

Iterate on node

Node=1

Yes

No

2

3

clock>mainStTime+min1

Yes

No

Process m_link2 to get detector information by order

Get deactivationTime for far side detector to decide if gap out

Get count and speed for each detector

Get clock for the last vehicle passing each detector

Figure 4.12 Flowchart for Dilemma Zone Control
Time[0]=clock;
speed[0]=m_speed;
oldCount0=m_count;
endOfGreen0=endOfGreen0+Extension0

deactivationTime>MaxGap

oldCount0!=cn[0]
If new veh. coming

Yes
g=0

g++
detectorCount=cn[0]

No

Time[0]=clock;
speed[0]=m_speed;
oldCount0=m_count;
endOfGreen0=endOfGreen0+Extension0

oldCount1!=cn[1]
cn[1]>=detectorCount

Yes

Time[1]=clock;
speed[1]=m_speed;
oldCount1=m_count;
endOfGreen1=endOfGreen1+Extension1

detectorCount=cn[1]

cn[1]>detectorCount

Yes

No

oldCount2!=cn[2]
cn[2]>=detectorCount

Yes

Time[2]=clock;
speed[2]=m_speed;
oldCount2=m_count;
endOfGreen2=endOfGreen2+Extension2

Figure 4.12 Flowchart for Dilemma Zone Control (continued)
Figure 4.12 Flowchart for Dilemma Zone Control (continued)
CHAPTER V
CONCLUSION AND FUTURE RESEARCH

This Hardware-in-loop system is effective to evaluate the traffic controller and combine the traffic controller with the software simulation system. For the traffic controller development, different control algorithm can be updated online. After the simulation, the controller can be directly used to test its performance in the field once the input/output interfaces are added, because it is separated from the software simulation system. The most impressive advantage is that the multiple data communication can be fulfilled in this system, which makes the control system development more flexible, where the user can add the desired data in his controller and evaluate the performance.

The future work should be focused on the practical applications of the developed controller and the interface development with the existed traffic control and data acquisition system. In addition, the system integration of different algorithms is also an important aspect for future research, especially for the intelligent transportation system.
REFERENCES


The Loop Simulation System For Transportation Networks (CD-ROM), National

[22] Nichols A., D. Bullock, and M. Kyte. A Laboratory-Based Course On Real-Time
Conference.

[23] Li Z., M. Kyte, and B. Johnson. A Controller Interface Device Based Suitcase


through Hands-On Experiences: Lessons Learned from a Workshop Prototype. In
Transportation Research Record, Transportation Research Board, National

Simulation : What’s the Difference, (CD-ROM), National Research Council,

[27] PI18FXX2 DATA SHEET, Microchip Technology Inc., 2002

[28] William Buchanan, PC Interfacing, Communication and Windows Programming,
Addison-Wesley, 1999

[29] Jan Axelson, Serial Port Complete, Lakeview Research, 1998
APPENDIX A

SOURCES CODE FOR PC SIDE

The sources codes are only different in the main function JMAIN, which provides the entry point for the DLL calling. The other parts of program are the same in each case. So only the JMAIN is listed for each case, while the other parts are just listed once in the Green Rest case.

A.1. Code List for Green Rest

******************************************************************************
Module Name: binarySequence.cpp
Description: Implementation file for the CBinarySequence class.
******************************************************************************
#include "stdafx.h"
#include "binarySequence.h"
#include "integer.h"

CBinarySequence::CBinarySequence() : CObject()
    , m_sequence()
{
}

CBinarySequence::~CBinarySequence()
{
    CInteger* pI = NULL;
    POSITION pos = m_sequence.GetHeadPosition();
    while( pos != NULL )
    {
        pI = m_sequence.GetNext( pos );
        delete pI;
    }
}
m_sequence.RemoveAll();
}
/******************************************************************************
Module Name: detector.cpp
Description: Implementation file for the CDetector class.
******************************************************************************
#include "stdafx.h"
#include "detector.h"
#include "link.h"
#include "lane.h"

CDetector::CDetector() : CObject()
,
.m_id( 0 )
,
.m_length( 0 )
,
.m_type()
,
.m_Link( NULL )
,
.m_distanceFromDownstreamNode( 0 )
,
.m_Lane( NULL )
,
.m_CorsimId( 0 )
,
.m_count( 0 )
,
.m_state( FALSE )
,
.m_activationTime( 0.0f )
,
.m_deactivationTime( 0.0f )
{
}

CDetector::~CDetector()
{
}

/******************************************************************************
Module Name: integer.cpp
Description: Implementation file for the CInteger class. An extension to
standard integers that can be used in MFC collections.
******************************************************************************
#include "stdafx.h"
#include "integer.h"

CInteger::CInteger() : CObject()
,
data( 0 )
{
}

83
CInteger::~CInteger()
{
}

/*************************************************************************/
 Module Name: lane.cpp
 Description: Implementation file for the CLane class.
/*************************************************************************/
#include "stdafx.h"
#include "lane.h"
#include "link.h"

CLane::CLane() : CObject()
 , m_id( 0 )
 , m_type( "Full" )
 , m_Link( NULL )
 , m_length( 0 )
 , m_leftMovementPercent( 0 )
 , m_thruMovementPercent( 0 )
 , m_rightMovementPercent( 0 )
 , m_DetectorList()
 , m_StopBarDetectorList()
{
}

CLane::~CLane()
{
    m_DetectorList.RemoveAll();
}

/*************************************************************************/
 Module Name: link.cpp
 Description: Implementation file for the CLink class.
/*************************************************************************/
#include "stdafx.h"
#include "detector.h"
#include "link.h"
#include "lane.h"
#include "integer.h"
#include "binarySequence.h"
#include "node.h"
#include "signalState.h"
CLink::CLink() : CObject()
    , m_id( 0 )
    , m_CorsimId( 0 )
    , m_upnode( NULL )
    , m_dnnode( NULL )
    , m_thrunode( NULL )
    , m_leftnode( NULL )
    , m_rightnode( NULL )
    , m_listOfLanes() 
    , m_listOfDetectors() 
    , m_signalStates() 
    , m_length( 0 )
    , m_numOfFullLanes( 0 )
    , m_numOfLeftTurnBays( 0 )
    , m_numOfRightTurnBays( 0 )
    , m_lengthOfLeftBay( 0 )
    , m_lengthOfRightBay( 0 )
    , m_freeFlowSpeed( 0 )
    , m_travelTime( 0.0f)
    , m_leftMovementPercent( 0 )
    , m_thruMovementPercent( 0 )
    , m_rightMovementPercent( 0 )
    , m_NumOfDetectors( 0 )
    , m_offset( 0 )
    , m_pos( NULL )
    , m_opposingLink( NULL )
    , m_opposingID( 0 )
{
}

CLink::~CLink()
{
    POSITION pos = NULL;

    // Delete the lanes on this link.
    CLane* pLane = NULL;

pos = m_listOfLanes.GetHeadPosition();
while( pos != NULL )
{
    pLane = m_listOfLanes.GetNext( pos );
    delete pLane;
}
m_listOfLanes.RemoveAll();
// Delete the detectors on this link.
CDetector* pDetector = NULL;
pos = m_listOfDetectors.GetHeadPosition();
while( pos != NULL )
{
    pDetector = m_listOfDetectors.GetNext( pos );
    delete pDetector;
}
m_listOfDetectors.RemoveAll();
// Delete the signal states for this link.
CSignalState* pSignalState = NULL;
pos = m_signalStates.GetHeadPosition();
while( pos != NULL )
{
    pSignalState = m_signalStates.GetNext( pos );
    delete pSignalState;
}
m_signalStates.RemoveAll();
}

void CLink::ProcessDetectors()
{
    // This method processes the information from the detectors
    // on this link.
    int index = 0;
    int det = 0;
    int detinfo = 0;
    int type = 0;
    unsigned int speed = 0;
    unsigned int num = 0;
    int newstate = 0;
    int oldstate = 0;

    POSITION pos = NULL;
    CDetector* pDetector = NULL;

    POSITION posI = NULL;
    CInteger* pI = NULL;
CBinarySequence* sequence = NULL;

// Compute the current time.
int nTime = sclock + giEndOfInit;

// Loop through the detectors.
pos = m_listOfDetectors.GetHeadPosition();
while( pos != NULL )
{
    pDetector = m_listOfDetectors.GetNext( pos );
detinfo = dtmod[pDetector->m_CorsimId];
    // detinfo is bit packed:
    // the first 3 bits contain the type of detector.
    type = detinfo & 0x07;

    // bits 4 - 10 contain the vehicles speed, so
    // (AND) detinfo with 03F8 = 1111111000.
    speed = detinfo & 0x03F8;

    // shift out the lowest 3 bits to get the speed
    speed = speed >> 3;

    // bits 11 - 23 contain the vehicle count since start of simulation.
    num = detinfo & 0x7FFC00;
    num = num >> 10;

    det = deton[pDetector->m_CorsimId];
    // Note: ConvertToBinary allocates memory for sequence.
    sequence = ConvertToBinary( det );

    // Loop through the binary sequence starting at the end because
    // when the sequence was created it is in reverse order.
posI = sequence->m_sequence.GetTailPosition();
    oldstate = (int)pDetector->m_state;
    for( index=0; index<10; index++ )
    {
        pI = sequence->m_sequence.GetPrev( posI );
        newstate = pI->data;

        if( oldstate == 0 && newstate == 1 )
        {
            // activation
            pDetector->m_activationTime = 0.1f;
        }
    }
}
pDetector->m_deactivationTime = 0.0f;
pDetector->m_state = TRUE;
pDetector->m_flag=TRUE;
}
if ( oldstate == 1 ) && ( newstate == 1 )
{
pDetector->m_activationTime = pDetector->m_activationTime + 0.1f;
pDetector->m_deactivationTime = 0.0f;
pDetector->m_state = TRUE;
pDetector->m_flag=TRUE;
}
if( (oldstate == 1) && (newstate == 0) )
{
    // deactivation
    pDetector->m_activationTime = 0.0f;
pDetector->m_deactivationTime = 0.1f;
pDetector->m_state = FALSE;
}
if( (oldstate == 0) && (newstate == 0) )
{
pDetector->m_activationTime = 0.0f;
pDetector->m_deactivationTime =
    pDetector->m_deactivationTime + 0.1f;
pDetector->m_state = FALSE;
}
oldstate = newstate;
}

// Clean up
delete sequence;
pDetector->m_count= (int)num;

}
// assume the binary number is less than 2^10
nRemainder = nValue;
for( iPower=9; iPower>-1; iPower-- )
{
    pI = new CInteger;

    if( nRemainder >= (int)pow(2, iPower) )
        pI->data = 1;
    else
        pI->data = 0;

    pSequence->m_sequence.AddTail( pI );

    nRemainder = nRemainder - (pI->data) * (int)pow( 2, iPower );
}

    // The sequence will be reversed.
    return pSequence;

CLane* CLink::FindLane( int corsimId )
{
    // Finds the lane on the link based on the lane's ID from the TRAF file.
    BOOL found = FALSE;
    CLane* pLane = NULL;
    POSITION pos = m_listOfLanes.GetHeadPosition();
    while( (pos != NULL) && (!found) )
    {
        pLane = m_listOfLanes.GetNext( pos );
        found = pLane->m_id == corsimId;
    }

    return pLane;
}

float CLink::ComputeTravelTime()
{
    // Computes the travel time to traverse a link.
    float travelTime = 0.0f;

    travelTime = (float)m_length / (float)m_freeFlowSpeed;

    return travelTime;
}
void CLink::CreateSignalStates()
{
    // Creates the list of signal states for each of the
    // links not under CORSIM control.
    int i = 0;
    int j = 0;
    int cycleLength = 0;
    int offset = 0;
    int duration = 0;
    int code = 0;
    int prevcode = 0;

    POSITION pos = NULL;
    CSignalState* pSignalState = NULL;

    // Sum the durations to get the cycle length for the fixed time control.
    for( i=0; i<12; i++ )
    {
        cycleLength = cycleLength + m_dnnode->m_duration[i];
    }

    // Create the signal states for every second.
    for( i=0; i<cycleLength; i++ )
    {
        pSignalState = new CSignalState();
        pSignalState->m_Link = this;
        m_signalStates.AddTail( pSignalState );
    }

    // Use the offset to determine where to begin creating the signal states.
    // Move to that position in the signal state list.
    offset = m_offset;
    pos = m_signalStates.GetHeadPosition();
    for( i=0; i<offset; i++ )
    {
        pSignalState = m_signalStates.GetNext( pos );
    }

    // For each duration set signal code and actual code, if the signal
    // code is 0 for amber then the actual code will be the previous
    // signal code that was not amber.
    for( i=0; i<12; i++ )
    {
        duration = m_dnnode->m_duration[i];
        code = m_code[i];
for( j=0; j<duration; j++ )
{
    if( pSignalState != NULL )
    {
        pSignalState->signalCode = code;
        if( code == 0 )
        {
            pSignalState->m_actualCode = prevcode;
        }
        else
        {
            pSignalState->m_actualCode = code;
            prevcode = code;
        }
    }
}

// Advance to the next signal state in the list.
if( pos == NULL )
pos = m_signalStates.GetHeadPosition();
pSignalState = m_signalStates.GetNext( pos );

// For each signal state get the appropriate values for the SDCODE and AMBSPC.
pos = m_signalStates.GetHeadPosition();
while( pos != NULL )
{
    m_pos = pos;                                    //m_pos point to the signal state.
pSignalState = m_signalStates.GetNext( pos );
pSignalState->SetSDCODE();
pSignalState->SetAMBSPC();
}

void CLink::SetSDCCode()
{
    // Set the SDCODE and AMBSPC for this link
    int nTime = sclock + giEndOfInit;

    CSignalState* pSignalState = NULL;
    if( (nTime == 0) || (m_pos == NULL) )
    {
        // At the end of the signal state list; so go back
        // to the beginning of the list
    }
m_pos = m_signalStates.GetHeadPosition();
}  
if( m_pos != NULL )
{
  pSignalState = m_signalStates.GetNext( m_pos );  // if there are signal state
  remained in the series, run them in order.
  NETSIM_LINKS_mp_SDCODE[m_CorsimId] = pSignalState->SDCODE;
  NETSIM_LINKS_mp_AMBSPC[m_CorsimId] = pSignalState->AMBSPC;
}

void CLink::AdjustSignalState()
{
  // Adjust the signal state to allow for protected lefts for the green ball.
  POSITION pos = NULL;
  POSITION posOpposing = NULL;
  CSignalState* pSignalState = NULL;
  CSignalState* pOpposingSignalState = NULL;

  pos = m_signalStates.GetHeadPosition();
  if( m_opposingLink != NULL )
  {
    posOpposing = m_opposingLink->m_signalStates.GetHeadPosition();
    while( pos != NULL )
    {
      pSignalState = m_signalStates.GetNext( pos );
      pOpposingSignalState = m_opposingLink->m_signalStates.GetNext( posOpposing );
      if( (pSignalState->signalCode == 1) &&
          (pOpposingSignalState->signalCode == 2) )
      {
        pSignalState->SDCODE = 0;
      }
    }
  }
}

/**************************************************************************/
Module Name: network.cpp

Description: Implementation file for the CNetwork class.

**************************************************************************/
CNetwork::CNetwork() : CObject()
   , m_LinkList()
   , m_NetworkName( "" )
   , m_TrafInputFile( "" )
   , m_NodeList()
{
   // default constructor
}

CNetwork::CNetwork( const CString& strInputFileName )
   : CObject()
   , m_LinkList()
   , m_NetworkName( "" )
   , m_TrafInputFile( strInputFileName )
   , m_NodeList()
{
}

CNetwork::~CNetwork()
{
   POSITION pos = NULL;
   CNode* pNode = NULL;
   CLink* pLink = NULL;

   // Delete the link list.
   pos = m_LinkList.GetHeadPosition();
   while( pos != NULL )
   {
      pLink = m_LinkList.GetNext( pos );
      delete pLink;
   }
   m_LinkList.RemoveAll();
}
// Delete the node list.
pos = m_NodeList.GetHeadPosition();
while( pos != NULL )
{
    pNode = m_NodeList.GetNext( pos );
    delete pNode;
}
m_NodeList.RemoveAll();

void CNetwork::ReadTrafFile()
{
    // Open the TRAF file.
    FILE* pFileTRF = NULL;
    if( pFileTRF = fopen(m_TrafInputFile, "r") )
    {
        // Create the node list.
        GetNodes( pFileTRF );
        rewind( pFileTRF );

        // Create the link list.
        GetLinks( pFileTRF );
        rewind( pFileTRF );

        // Find the opposing link for each link.
        int up = 0;
        int down = 0;
        CLink* pLink = NULL;
        POSITION pos = m_LinkList.GetHeadPosition();
        while( pos != NULL )
        {
            pLink = m_LinkList.GetNext( pos );
            up = pLink->m_opposingID;
            down = pLink->m_dnnode->m_id;
            pLink->m_opposingLink = FindLink( up, down );
        }

        // Create the lanes on each link.
        CreateLanes( pFileTRF );
        rewind( pFileTRF );

        // Create the signal timings for the nodes to
        // be controlled by the algorithm.
    }
}
ReadCard35( pFileTRF );
rewind( pFileTRF );

ReadCard36( pFileTRF );
rewind( pFileTRF );

CreateSignalStates();

// Create the detectors that exist in the TRAF file.
GetDetectors( pFileTRF );
fclose( pFileTRF );

int CNetwork::ReadTRFLine( FILE* pFile, char* pszLine )
{
    // WARNING: To use this function properly, the caller must
    // ensure that pszLine is allocated with at least
    // 81 characters.

    int nCardType = -1;   // Card Type: -1 = failed to read line
    char lastchar;        // Used to store last character of the line
    char strDiscard[81];   // Used to discard characters beyond column 80

    // Clear the line (fill with nulls). This is necessary for the
    // card type extraction to work properly for lines that are shorter
    // than 80 characters.
    memset( pszLine, 0, 81 );

    // Read an entire line (up to 80 characters) until a newline character
    // is encountered. If fgets encounters a newline character, it places
    // the newline character into the buffer.
    if( fgets(pszLine, 81, pFile) != NULL )
    {
        // If a newline character was not encountered, continue to
        // read and discard bytes until a newline is encountered.
        lastchar = pszLine[strlen(pszLine)-1];
        while( lastchar != '\n' )
        {
            if( fgets(strDiscard, 81, pFile) != NULL )
                lastchar = strDiscard[strlen(strDiscard)-1];
            else
                break;
        }
    }
// Make sure the line is null terminated.
pszLine[80] = "0";

// Extract the card type from the line and return.
sscanf( pszLine + 77, "%d", &nCardType );

// Ensure the card type is in the proper range. If not,
// return an error code.
if( nCardType < 0 || 210 < nCardType )
    nCardType = -1;
else
{
    // Error reading the file.
    if( ferror(pFile) )
    {
        pszLine[0] = "0";
        nCardType = -1;
    }
}

return nCardType;
}

void CNetwork::GetNodes( FILE* pFile )
{
    // Get the node information from the TRAF file
    // and create the node list.
    char line[81] = { '0' };
    int nCardType = 0;

    CNode* pNode = NULL;
    POSITION pos = NULL;

    int nNodeID = 0;
    char NodeID[5] = { '0' };
    char ControlType[2] = { '0' };
    char xpos[7] = { '0' };
    char ypos[7] = { '0' };

    // Read data records from the TRAF file.
    while( !feof(pFile) )
    {
        // Read a line of the file.
        nCardType = ReadTRFLine( pFile, line );
// Parse the line based on its card type.
if( nCardType == 36 || nCardType == 43 )
{
    // First, extract the node ID.
    strncpy( NodeID, line, 4 );
    NodeID[4] = '\0';
    nNodeID = atoi( NodeID );

    // Second, determine if this node is under external control.
    strncpy( ControlType, line + 76, 1 );
    ControlType[1] = '\0';

    if( nNodeID < 9000 )
    {
        // Create a node object.
        pNode = new CNode();
        pNode->m_id = nNodeID;
        if( pNode->m_id > 8000 )
        {
            // Source node.
            pNode->m_typeOfControl = CString( "source" );
        }
        else
        {
            if( (ControlType[0] == '2') || (ControlType[0] == '3') )
            {
                // Node is controlled by algorithm external to CORSIM.
                pNode->m_typeOfControl = CString( "external" );
            }
            else
            {
                // Node is controlled by CORSIM.
                pNode->m_typeOfControl = CString( "corsim" );
            }
        }
        m_NodeList.AddTail( pNode );
    }
    else if( nCardType == 195 )
    {
        // First, get the node ID.
        strncpy( NodeID, line, 4 );
        NodeID[4] = '\0';
        nNodeID = atoi( NodeID );

        // Second, determine if this node is under external control.
        strncpy( ControlType, line + 76, 1 );
        ControlType[1] = '\0';

        if( nNodeID < 9000 )
        {
            // Create a node object.
            pNode = new CNode();
            pNode->m_id = nNodeID;
            if( pNode->m_id > 8000 )
            {
                // Source node.
                pNode->m_typeOfControl = CString( "source" );
            }
            else
            {
                if( (ControlType[0] == '2') || (ControlType[0] == '3') )
                {
                    // Node is controlled by algorithm external to CORSIM.
                    pNode->m_typeOfControl = CString( "external" );
                }
                else
                {
                    // Node is controlled by CORSIM.
                    pNode->m_typeOfControl = CString( "corsim" );
                }
            }
            m_NodeList.AddTail( pNode );
        }
    }
}
// Second, extract the node coordinates.
strncpy( xpos, line + 6, 6 );
xpos[6] = 0;

strncpy( ypos, line + 14, 6 );
ypos[6] = 0;

// Search the node list for the corresponding node object and
// set its node coordinates.
pos = m_NodeList.GetHeadPosition();
while( pos != NULL )
{
    pNode = m_NodeList.GetNext( pos );
    if( pNode->m_id == nNodeID )
    {
        pNode->m_xPos = atoi( xpos );
        pNode->m_yPos = atoi( ypos );
    }
}

// Assign the CORSIM internal ID to each node.
iNodeID = 0;
BOOL found;
pos = m_NodeList.GetHeadPosition();
while( pos != NULL )
{
    iNodeID = 0;
    found = FALSE;
pNode = m_NodeList.GetNext( pos );
    while( (iNodeID < IMXNOD) && (!found) )
    {
        if( pNode->m_id == nmap[iNodeID] )
        {
            found = TRUE;
            pNode->m_CORSIMInternalID = iNodeID;
        }
        iNodeID++;
    }
}
CNode* CNetwork::FindNode( int id )
{
    // Find the node (ID) in the node list.
    BOOL found = FALSE;
    CNode* pNode = NULL;
    POSITION pos = m_NodeList.GetHeadPosition();
    while( (pos != NULL) && (!found) )
    {
        pNode = m_NodeList.GetNext( pos );
        found = pNode->m_id == id;
    }

    if( found )
    {
        // Node was found.
        return pNode;
    }
    else
    {
        // Node was not found.
        return NULL;
    }
}

void CNetwork::GetLinks( FILE* pFile )
{
    // Get the information about the links in the
    // TRAF file and create the link list.
    char line[81] = { '\0' };
    int nCardType = 0;
    int index = 0;
    CLink* pLink = NULL;
    // Link data.
    char up[5] = { '\0' };
    char dn[5] = { '\0' };
    char th[5] = { '\0' };
    char le[5] = { '\0' };
    char ri[5] = { '\0' };
    char op[5] = { '\0' };
    char length[5] = { '\0' };
    char speed[5] = { '\0' };
    char lengthOfLeftBay[5] = { '\0' };
    char lengthOfRightBay[5] = { '\0' };
    char numOfFullLanes[5] = { '\0' };
    char numOfLeftBays[5] = { '\0' };
}
char numOfRightBays[5] = { '0' };
char ch[7][5] = { '0' };

// Search the TRAF file for link data records.
while( !feof(pFile) )
{
    // Read a line of the file.
    nCardType = ReadTRFLine( pFile, line );

    // Parse the type 11 (link data) cards.
    if( nCardType == 11 )
    {
        // Initialize all the character data strings.
        for( index=0; index<5; index++ )
        {
            up[index] = '\0';
            dn[index] = '\0';
            th[index] = '\0';
            le[index] = '\0';
            ri[index] = '\0';
            op[index] = '\0';
            length[index] = '\0';
            speed[index] = '\0';
            lengthOfLeftBay[index] = '\0';
            lengthOfRightBay[index] = '\0';
            numOfFullLanes[index] = '\0';
            numOfLeftBays[index] = '\0';
            numOfRightBays[index] = '\0';
            ch[0][index] = '\0';
            ch[1][index] = '\0';
            ch[2][index] = '\0';
            ch[3][index] = '\0';
            ch[4][index] = '\0';
            ch[5][index] = '\0';
            ch[6][index] = '\0';
        }

        // Parse the line, which contains:
        // the upstream node number,
        // the downstream node number,
        // the link length,
        // turn bay length,
        // etc. (see documentation describing CARD 11).
        for( index=0; index<4; index++ )
        {
            // Read a line of the file.
        }
    }
}
up[index] = line[index];
dn[index] = line[index+4];
length[index] = line[index+8];
lengthOfLeftBay[index] = line[index+12];
lengthOfRightBay[index] = line[index+16];
th[index] = line[index+40];
le[index] = line[index+36];
ri[index] = line[index+44];
op[index] = line[index+52];
speed[index] = line[index+64];
}

// Read the channelization codes for each lane.
for( index=0; index<7; index++ )
{
    ch[index][1] = line[index+29];
}

numOfFullLanes[0] = line[21];
umOfLeftBays[0] = line[23];
umOfRightBays[0] = line[25];

if( (atoi(dn) < 8000) && (atoi(up) < 8000) )
{
    // Link is not a source link so create it.
    pLink = new CLink();
    pLink->m_numOfFullLanes = atoi( numOfFullLanes );
    pLink->m_numOfLeftTurnBays = atoi( numOfLeftBays );
    pLink->m_lengthOfLeftBay = atoi( lengthOfLeftBay );
    pLink->m_numOfRightTurnBays = atoi( numOfRightBays );
    pLink->m_lengthOfRightBay = atoi( lengthOfRightBay );
    pLink->m_length = atoi( length );
    pLink->m_freeFlowSpeed = atoi( speed );
    if( pLink->m_freeFlowSpeed == 0 )
        pLink->m_freeFlowSpeed = 44;
    pLink->m_upnode = FindNode( atoi(up) );
    pLink->m_dnnode = FindNode( atoi(dn) );
    pLink->m_thrunode = FindNode( atoi(th) );
    pLink->m_leftnode = FindNode( atoi(le) );
    pLink->m_rightnode = FindNode( atoi(ri) );
    pLink->m_CorsimId = GetLinkCorsimId( pLink->m_upnode->m_id,
                                       pLink->m_dnnode->m_id );
    pLink->m_opposingID = atoi( op );
    if( (atoi(up) < 8000) && (atoi(dn) < 8000) )
        pLink->m_travelTime = pLink->ComputeTravelTime();
}
for (index=0; index<7; index++)
{
    pLink->m_channelCode[index] = CString(ch[index][1]);
}
m_LinkList.AddTail(pLink);
}
else if( nCardType > 11 )
{
    // Because CORSIM expects the cards to be in ascending order
    // by card type, we can jump out of the loop as soon as we
    // encounter a card type greater than 11.
    break;
}
}

CLink* CNetwork::FindLink( int up, int dn )
{
    CLink* pLink = NULL;

    // Finds the link between nodes (up, dn).
    BOOL found = FALSE;
    POSITION posLink = m_LinkList.GetHeadPosition();
    while( (posLink != NULL) && (!found) )
    {
        pLink = m_LinkList.GetNext( posLink );
        found = (pLink->m_upnode->m_id == up) &&
                (pLink->m_dnnode->m_id == dn);
    }

    if( found )
    {
        return pLink;
    }
    else
    {
        return NULL;
    }
}

int CNetwork::GetLinkCorsimId( int upnode, int dnnode )
{
    // Find the CORSIM link ID for the link (upnode, dnnode).
    int id = 0;
    int index = 0;
    int dnode = 0;
    int unode = 0;
// Search through all the links in CORSIM.
for( index=0; index<ttlnk; index++ )
{
    // For the ith link, get the downstream node and
    // upstream node as represented in CORSIM.
    dnode = NETSIM_LINKS_mp_DWNOD[index];
    unode = NETSIM_LINKS_mp_UPNOD[index];
    // For non-source nodes (<7000), use the nmap array to map
    // the node number in CORSIM back to the user defined node
    // number in the TRAF file. Use an offset of -1 (i.e., dnode-1),
    // because C arrays start at 0; FORTRAN arrays start at 1.
    if( dnode < 7000 ) dnode = nmap[dnode-1];
    if( unode < 7000 ) unode = nmap[unode-1];
    if( (dnode == dnnode) && (unode == upnode) )
    {
        id = index;
    }
}

return id;
}

void CNetwork::UpdateNodeSignalStates()
{
    // Update the signal states for all the
    // nodes not controlled by CORSIM.
    int nTime = sclock + giEndOfInit;

    CNode* pNode = NULL;
    POSITION pos = m_NodeList.GetHeadPosition();
    while( pos != NULL )
    {
        pNode = m_NodeList.GetNext( pos );
        if( pNode->m_typeOfControl == CString("external") )
        {
            // Node is not controlled by CORSIM.
            pNode->SetSignalState();
        }
    }
}

void CNetwork::GetDetectors( FILE* pFile )
{
    // Read the detectors information from the TRAF
    // input file and create the detector list.
char line[81] = { '\0' };
int nCardType = 0;

int index = 0;
BOOL found = FALSE;
CLink* pLink = NULL;
CLane* pLane = NULL;
POSITION posLink = NULL;
POSITION posLane = NULL;
CDetector* pDetector = NULL;

// Detector data.
char up[5] = { '\0' };
char dn[5] = { '\0' };
char lane[2] = { '\0' };
char distance[6] = { '\0' };
char type[2] = { '\0' };
char id[5] = { '\0' };
char length[5] = { '\0' };

// Search the TRAF file for detector data records.
while( !feof(pFile) )
{
    // Read a line of the file.
    nCardType = ReadTRFLine( pFile, line );

    // Parse the type 42 (detector data) cards.
    if( nCardType == 42 )
    {
        // Initialize the character data strings.
        for( index=0; index<5; index++ )
        {
            up[index] = '\0';
            dn[index] = '\0';
            id[index] = '\0';
            length[index] = '\0';
        }

        for( index=0; index<6; index++ )
        {
            distance[index] = '\0';
        }

        for( index=0; index<2; index++ )
        {
}}
lane[index] = '"0';
type[index] = '"0';
}

// Parse the line.
for( index=0; index<4; index++ )
{
    up[index] = line[index];
    dn[index] = line[index+4];
    id[index] = line[index+22];
    length[index] = line[index+28];
}

lane[0] = line[11];
type[0] = line[34];

for( index=0; index<5; index++ )
{
    distance[index] = line[index+15];
}

// Create the detector.
pDetector = new CDetector();
pDetector->m_count = 0;
pDetector->m_id = atoi( id );
pDetector->m_length = atoi( length );
pDetector->m_distanceFromDownstreamNode =
    (int)((float)atoi(distance)/(float)10.0);

// Search for the link the detector is on.
found = FALSE;
posLink = m_LinkList.GetHeadPosition();
while( (posLink != NULL) && (!found) )
{
    pLink = m_LinkList.GetNext( posLink );
    found = (pLink->m_upnode->m_id == atoi(up)) &&
        (pLink->m_dnnode->m_id == atoi(dn));
}
pDetector->m_Link = pLink;

// Search for lane the lane the detector is on.
if( pLink != NULL )
{
    found = FALSE;
posLane = pLink->m_listOfLanes.GetHeadPosition();
}
while( (posLane != NULL) && (!found) )
{
    pLane = pLink->m_listOfLanes.GetNext( posLane );
    found = pLane->m_id == atoi( lane );
}
pDetector->m_Lane = pLane;
GetDetectorCorsimId( pDetector );
pLink->m_listOfDetectors.AddTail( pDetector );

if( pLane != NULL )
    pLane->m_DetectorList.AddTail( pDetector );
pLink->m_NumOfDetectors++;
}
else if( nCardType > 42 )
{
    // Because CORSIM expects the cards to be in ascending order
    // by card type, we can jump out of the loop as soon as we
    // encounter a card type greater than 42.
    break;
}
}

void CNetwork::GetDetectorCorsimId( CDetector* pDetector )
{
    // Finds the CORSIM detector ID for pDetector.
    int index = 0;
    int id = 0;
    BOOL found = FALSE;

    int distance=pDetector->m_distanceFromDownstreamNode;
    CLink* pLink=pDetector->m_Link;
    CLane* pLane=pDetector->m_Lane;

    // Get the id of the first detector on the same link as pDetector.
    id = NETSIM_LINKS_mp_DTFLNK[pLink->m_CorsimId];
    // Compare the position and lane for the first detector
    // with that of pDetector.
    found = (pLane->m_id==dtlane[2*id-2]) &&
            (distance==dtpos[id-1]/10);
    index = 0;
    while( (!found) && (index<pLink->m_NumOfDetectors) )
    {
        // Not found so get the next detector on this link
void CNetwork::CreateLanes( FILE* pFile )
{
    // Create the lanes.
    char line[81] = { '\0' };
    int nCardType = 0;

    int index = 0;
    int lastid = 0;
    BOOL found = FALSE;
    POSITION pos = NULL;
    CLink* pLink = NULL;
    CLane* pLane = NULL;
    CString channelCode;

    // Lane data.
    char up[5] = { '\0' };
    char dn[5] = { '\0' };
    char left[5] = { '\0' };
    char thru[5] = { '\0' };
    char right[5] = { '\0' };

    // Search the TRAF file for link turn movement data records.
    while( !feof(pFile) )
    {
        // Read a line of the file.
        nCardType = ReadTRFLine( pFile, line );
        // Parse the type 21 (link turn movement data) cards.
        if( nCardType == 21 )
        {
            // Initialize th. character data strings.
            for( index=0; index<5; index++ )
            {
                id = dtnlnk[id-1];
                found = (pLane->m_id==dtlane[2*id-2]) &&
                        (distance==dtpos[id-1]/10);
                index++;
            }
            if( found )
            {
                // Detector found.
                pDetector->m_CorsimId = id - 1;
            }
        }
    }
}
up[index] = '\0';
dn[index] = '\0';
left[index] = '\0';
thru[index] = '\0';
right[index] = '\0';
}

for( index=0; index<4; index++ )
{
    up[index] = line[index];
dn[index] = line[index+4];
left[index] = line[index+8];
thru[index] = line[index+12];
right[index] = line[index+16];
}

// Search for the link between nodes up and down.
found = FALSE;
pos = m_LinkList.GetHeadPosition();
while( (pos != NULL) && (!found) )
{
    pLink = m_LinkList.GetNext( pos );
    found = (pLink->m_upnode->m_id==atoi(up)) &&
        (pLink->m_dnnode->m_id==atoi(dn));
}

if( found && pLink != NULL )
{
    // Assign the turning percentages for the link.
pLink->m_leftMovementPercent  = atoi( left );
pLink->m_thruMovementPercent  = atoi( thru );
pLink->m_rightMovementPercent = atoi( right );
}
else if( nCardType > 21 )
{
    // Because CORSIM expects the cards to be in ascending order
    // by card type, we can jump out of the loop as soon as we
    // encounter a card type greater than 21.
    break;
}

// For each link create the lanes for the link
float fLeft = 0.0f;
float fThru = 0.0f;
float fRight = 0.0f;
pos = m_LinkList.GetHeadPosition();
while( pos != NULL )
{
pLink = m_LinkList.GetNext( pos );

for( index=0; index<pLink->m_numOfFullLanes; index++ )
{
pLane = new CLane();
pLane->m_id = index + 1;
channelCode = pLink->m_channelCode[index];
if( channelCode == CString("T") )
{
    // Lane is thru only.
    pLane->m_leftMovementPercent = 0;
pLane->m_rightMovementPercent = 0;
pLane->m_thruMovementPercent = 100;
}
if( channelCode == CString("1") )
{
    // Lane is left turn only.
    pLane->m_leftMovementPercent = 100;
pLane->m_rightMovementPercent = 0;
pLane->m_thruMovementPercent = 0;
}
if( channelCode == CString("4") )
{
    // Lane is right turn only.
    pLane->m_leftMovementPercent = 0;
pLane->m_rightMovementPercent = 100;
pLane->m_thruMovementPercent = 0;
}
if( channelCode == CString("7") )
{
    // Lane is right and thru.
    fRight = (float)pLink->m_rightMovementPercent;
fThru = (float)pLink->m_thruMovementPercent;
pLane->m_leftMovementPercent = 0;
pLane->m_rightMovementPercent = (int)(100*(fRight/(fRight+fThru)));
pLane->m_thruMovementPercent = (int)(100*(fThru/(fThru+fRight)));
}
if( channelCode == CString("8") )
{
    // Lane is left and thru.
fThru = (float)pLink->m_thruMovementPercent;
fLeft = (float)pLink->m_leftMovementPercent;
pLane->m_leftMovementPercent = (int)(100*(fLeft/(fLeft+fThru)));
pLane->m_thruMovementPercent = (int)(100*(fThru/(fLeft+fThru)));
pLane->m_rightMovementPercent = 0;
}
if( channelCode == CString("9") )
{
    // Lane is left, thru, and right.
    fLeft = (float)pLink->m_leftMovementPercent;
    fThru = (float)pLink->m_thruMovementPercent;
    fRight = (float)pLink->m_rightMovementPercent;
    pLane->m_leftMovementPercent = (int)(100*(fLeft/(fLeft+fRight+fThru)));
    pLane->m_rightMovementPercent = (int)(100*(fRight/(fLeft+fRight+fThru)));
    pLane->m_thruMovementPercent = (int)(100*(fThru/(fLeft+fRight+fThru)));
}

pLane->m_Link = pLink;
pLane->m_type = CString( "Full" );
pLink->m_listOfLanes.AddTail( pLane );

// Compute lane number for the first right turn bay --
// see documentation for CARD type 11.
lastid = 7 - pLink->m_numOfRightTurnBays - pLink->m_numOfLeftTurnBays + 1;
for( index=0; index<pLink->m_numOfRightTurnBays; index++ )
{
    pLane = new CLane();
    pLane->m_id = lastid + index;
    pLane->m_leftMovementPercent = 0;
    pLane->m_Link = pLink;
    pLane->m_rightMovementPercent = 100;
    pLane->m_thruMovementPercent = 0;
    pLane->m_type = CString ("Bay");
    pLink->m_listOfLanes.AddTail( pLane );
}

// Compute the lane number for the first left turn bay --
// see documentation for CARD type 11.
lastid = 7 - pLink->m_numOfLeftTurnBays + 1;
for( index=0; index<pLink->m_numOfLeftTurnBays; index++ )
{
    pLane = new CLane();
    pLane->m_id = lastid + index;
    pLane->m_leftMovementPercent = 100;
pLane->m_Link = pLink;
pLane->m_rightMovementPercent = 0;
pLane->m_thruMovementPercent = 0;
pLane->m_type = CString( "Bay" );
pLink->m_listOfLanes.AddTail( pLane );
}
}
}

void CNetwork::ReadCard35( FILE * pFile )
{
    // Read card type 35 to get the duration for each signal interval
    // and the approach links for the nodes.
    char line[81] = { '\0' };
    int nCardType = 0;
    int index = 0;
    int up = 0;
    int down = 0;
    CNode* pNode = NULL;
    // Type 35 record data.
    char node[5] = { '\0' };
    char offset[5] = { '\0' };
    char upnode1[5] = { '\0' };
    char upnode2[5] = { '\0' };
    char upnode3[5] = { '\0' };
    char upnode4[5] = { '\0' };
    char upnode5[5] = { '\0' };
    char dur1[4] = { '\0' };
    char dur2[4] = { '\0' };
    char dur3[4] = { '\0' };
    char dur4[4] = { '\0' };
    char dur5[4] = { '\0' };
    char dur6[4] = { '\0' };
    char dur7[4] = { '\0' };
    char dur8[4] = { '\0' };
    char dur9[4] = { '\0' };
    char dur10[4] = { '\0' };
    char dur11[4] = { '\0' };
    char dur12[4] = { '\0' };

    // Search the TRAF file for type 35 data records.
    while( !feof(pFile) )
    {
        // Read a line of the file.
        nCardType = ReadTRFLine( pFile, line );
        
    }
}
// Parse the type 35 cards.
if( nCardType == 35 )
{
    // Initialize the character data strings.
    for( index=0; index<5; index++ )
    {
        node[index] = '\0';
        offset[index] = '\0';
        upnode1[index] = '\0';
        upnode2[index] = '\0';
        upnode3[index] = '\0';
        upnode4[index] = '\0';
        upnode5[index] = '\0';
    }
    for( index=0; index<4; index++ )
    {
        dur1[index] = '\0';
        dur2[index] = '\0';
        dur3[index] = '\0';
        dur4[index] = '\0';
        dur5[index] = '\0';
        dur6[index] = '\0';
        dur7[index] = '\0';
        dur8[index] = '\0';
        dur9[index] = '\0';
        dur10[index] = '\0';
        dur11[index] = '\0';
        dur12[index] = '\0';
    }
    // Parse the line.
    for( index=0; index<4; index++ )
    {
        // Get the node number, offset, and
        // upstream nodes for the 5 approaches.
        node[index] = line[index];
        offset[index] = line[index+4];
        upnode1[index] = line[index+8];
        upnode2[index] = line[index+12];
        upnode3[index] = line[index+16];
        upnode4[index] = line[index+20];
        upnode5[index] = line[index+24];
    }
    for( index=0; index<3; index++ )
// Get the durations the 12 signal intervals.
dur1[index] = line[index+29];
dur2[index] = line[index+33];
dur3[index] = line[index+37];
dur4[index] = line[index+41];
dur5[index] = line[index+46];
dur6[index] = line[index+49];
dur7[index] = line[index+53];
dur8[index] = line[index+57];
dur9[index] = line[index+61];
dur10[index] = line[index+65];
dur11[index] = line[index+69];
dur12[index] = line[index+73];
}

up = atoi( upnode1 );
down = atoi( node );
pNode = FindNode( down );

// Assign the 5 approach links for this node.
pNode->m_Link1 = FindLink( up, down );
if( pNode->m_Link1 )
    pNode->m_Link1->m_offset = atoi( offset );

up = atoi( upnode2 );
pNode->m_Link2 = FindLink( up, down );
if( pNode->m_Link2 )
    pNode->m_Link2->m_offset = atoi( offset );

up = atoi( upnode3 );
pNode->m_Link3 = FindLink( up, down );
if( pNode->m_Link3 )
    pNode->m_Link3->m_offset = atoi( offset );

up = atoi( upnode4 );
pNode->m_Link4 = FindLink( up, down );
if( pNode->m_Link4 )
    pNode->m_Link4->m_offset = atoi( offset );

up = atoi( upnode5 );
pNode->m_Link5 = FindLink( up, down );
if( pNode->m_Link5 )
    pNode->m_Link5->m_offset = atoi( offset );

    // Store the durations for the 12 signal states.
    pNode->m_duration[0]  = atoi( dur1 );
    pNode->m_duration[1]  = atoi( dur2 );
    pNode->m_duration[2]  = atoi( dur3 );
    pNode->m_duration[3]  = atoi( dur4 );
    pNode->m_duration[4]  = atoi( dur5 );
    pNode->m_duration[5]  = atoi( dur6 );
    pNode->m_duration[6]  = atoi( dur7 );
    pNode->m_duration[7]  = atoi( dur8 );
    pNode->m_duration[8]  = atoi( dur9 );
    pNode->m_duration[9]  = atoi( dur10 );
    pNode->m_duration[10] = atoi( dur11 );
    pNode->m_duration[11] = atoi( dur12 );
}
else if( nCardType > 35 )
{
    // Because CORSIM expects the cards to be in ascending order
    // by card type, we can jump out of the loop as soon as we
    // encounter a card type greater than 35.
    break;
}
}

void CNetwork::ReadCard36( FILE * pFile )
{
    // Read the 36 cards in the input TRAF file.
    char line[81] = { '\0' };
    int nCardType = 0;

    int index = 0;
    int iInterval = 0;
    int iApproach = 0;
    int down = 0;
    CNode* pNode = NULL;

    // Type 36 card data.
    char node[5] = { '\0' };
    char code[2] = { '\0' };
    int controlCode[12][5] = { '\0' }; // intervals, links
// Search the TRAF file for type 36 data records.
while( !feof(pFile) )
{
    // Read a line of the file.
    nCardType = ReadTRFLine( pFile, line );

    // Parse the type 36 cards.
    if( nCardType == 36 )
    {
        // Initialize the character data strings
        for( index=0; index<5; index++ )
        {
            node[index] = '\0';
        }
        for( index=0; index<2; index++ )
        {
            code[index] = '\0';
        }
        for( iInterval=0; iInterval<12; iInterval++ )
        {
            for( iApproach=0; iApproach<5; iApproach++ )
            {
                controlCode[iInterval][iApproach] = 0;
            }
        }

        // Parse the line
        for( index=0; index<4; index++ )
        {
            node[index] = line[index];
        }

        down = atoi( node );
        pNode = FindNode( down );

        if( pNode != NULL )
        {
            for( index=5; index<65; index++ )
            {
                code[0] = line[index];
                code[1] = '\0';
                iInterval = (int)((index-5)/5);
                iApproach = (index-5) - 5 * iInterval;
                controlCode[iInterval][iApproach] = atoi( code );
            }
        }
    }
}
// Store the control codes for each of the approach links
// for each of the 12 signal intervals.
for( iInterval=0; iInterval<12; iInterval++ )
{
    if( pNode->m_Link1 != NULL )
        pNode->m_Link1->m_code[iInterval] = controlCode[iInterval][0];
    if( pNode->m_Link2 != NULL )
        pNode->m_Link2->m_code[iInterval] = controlCode[iInterval][1];
    if( pNode->m_Link3 != NULL )
        pNode->m_Link3->m_code[iInterval] = controlCode[iInterval][2];
    if( pNode->m_Link4 != NULL )
        pNode->m_Link4->m_code[iInterval] = controlCode[iInterval][3];
    if( pNode->m_Link5 != NULL )
        pNode->m_Link5->m_code[iInterval] = controlCode[iInterval][4];
}
else if( nCardType > 36 )
{
    // Because CORSIM expects the cards to be in ascending order
    // by card type, we can jump out of the loop as soon as we
    // encounter a card type greater than 36.
    break;
}
}

void CNetwork::CreateSignalStates()
{
    // Create the list of signal states for each link.
    CLink* pLink = NULL;
    POSITION pos = m_LinkList.GetHeadPosition();
    while( pos != NULL )
    {
        pLink = m_LinkList.GetNext( pos );
        pLink->CreateSignalStates();
    }
    pos = m_LinkList.GetHeadPosition();
    while( pos != NULL )
    {


```cpp
{
    pLink = m_LinkList.GetNext( pos );
    pLink->AdjustSignalState();
}
}

*****************************************************************************
 Module Name: node.cpp

 Description: Implementation file for the CNode class.

*****************************************************************************
#include "stdafx.h"
#include "link.h"
#include "netsim.h"
#include "node.h"
#include "upcntrl.h"

CNode::CNode() : CObject()
    , m_id( 0 )
    , m_CORSIMInternalID( 0 )
    , m_xPos( 0 )
    , m_yPos( 0 )
    , m_Link1( NULL )
    , m_Link2( NULL )
    , m_Link3( NULL )
    , m_Link4( NULL )
    , m_Link5( NULL )
    , m_typeOfControl()
{
}

CNode::~CNode()
{
}

void CNode::SetSignalState()
{
    // Set the signal state for each node.
    int nTime = sclock + giEndOfInit;
    BOOL in = yinit;

    SetSDCCode();
}
void CNode::SetSDCCode()
```
{ // Set the SDCODE and AMBSPC for each of the 5 approach links.
    if( m_Link1 != NULL )
        m_Link1->SetSDCCode();

    if( m_Link2 != NULL )
        m_Link2->SetSDCCode();

    if( m_Link3 != NULL )
        m_Link3->SetSDCCode();

    if( m_Link4 != NULL )
        m_Link4->SetSDCCode();

    if( m_Link5 != NULL )
        m_Link5->SetSDCCode();
}

void CNode::Setm_pos(int offset)
{
    // set the m_pos as the defined value for each link of this node.
    int i=0;
    POSITION pos1 = m_Link1->m_signalStates.GetHeadPosition();
    for(i=0; i<offset; i++ )
    {
        CSignalState* pSignalState = m_Link1->m_signalStates.GetNext(pos1);
    }

    POSITION pos2 = m_Link2->m_signalStates.GetHeadPosition();
    for(i=0; i<offset; i++ )
    {
        CSignalState* pSignalState = m_Link2->m_signalStates.GetNext(pos2);
    }

    POSITION pos3 = m_Link3->m_signalStates.GetHeadPosition();
    for(i=0; i<offset; i++ )
    {
        CSignalState* pSignalState = m_Link3->m_signalStates.GetNext(pos3);
    }

    POSITION pos4 = m_Link4->m_signalStates.GetHeadPosition();
    for(i=0; i<offset; i++ )
    {
        CSignalState* pSignalState = m_Link4->m_signalStates.GetNext(pos4);
    }

    m_Link1->m_pos=pos1;
    m_Link2->m_pos=pos2;
m_Link3->m_pos=pos3;
m_Link4->m_pos=pos4;
}

/******************************************************************************
 Module Name: signalState.cpp
 Description: Implementation file for the CSignalState class.
 ******************************************************************************/
#include "stdafx.h"
#include "link.h"
#include "signalState.h"

CSignalState::CSignalState() : CObject()
    , signalCode( -10 )
    , m_actualCode( 0 )
    , SDCODE( 0 )
    , AMBSPC( 0 )
    , m_Link( NULL )
{
}

CSignalState::~CSignalState()
{
}

void CSignalState::SetSDCODE()
{
    SDCODE = GetSDCODE( m_actualCode );
}

int CSignalState::GetSDCODE( int actualCode )
{
    // The actualCode is the code read from the 36 cards in the input TRAF
    // file; it ranges from 1 - 9. See the documentation for Card type 36
    // in the TSIS users guide.
    int sdc = -1;

    switch( actualCode )
    {
    case 1:  // green ball
               if( m_Link->m_opposingLink == NULL )
               {
                   // without permitted left
                   sdc = 0;

               }
} else {
    // with permitted left
    sdc = 16;
}
break;

case 2: // red ball
    sdc = 15;
    break;

case 3: // green right turn only
    sdc = 14;
    break;

case 4: // green left turn only
    sdc = 7;
    break;

case 5: // stop sign
    sdc = 15;
    break;

case 6: // green diagonal only
    sdc = 11;
    break;

case 7: // green through only
    sdc = 13;
    break;

case 8: // green right and left turn only
    sdc = 2;
    break;

case 9: // green thru and right only
    sdc = 8;
    break;

default: // invalid code
    break;
}

return sdc;

void CSignalState::SetAMBSPC() {

    // Set the AMBSPC for the signal state. The AMBSPC for the current signal
    // state will depend on the next non-yellow or non-red signal state.

    POSITION pos = NULL;
    CSignalState* nextSignalState = NULL;
    int nextActualCode = 0;


// Find the next signal state with a nonzero signal code
// a zero signal code means amber.
pos = m_Link->m_pos;
if( pos == NULL )
{
    pos = m_Link->m_signalStates.GetHeadPosition();
    nextSignalState = m_Link->m_signalStates.GetNext( pos );
}
else

    nextSignalState = m_Link->m_signalStates.GetNext( pos );

}  

BOOL done = nextSignalState->signalCode != 0;
while( !done )
{
    // Signal code is 0 for yellow so get the next signal state.
    if( pos == NULL )
    {
        pos = m_Link->m_signalStates.GetHeadPosition();
        nextSignalState = m_Link->m_signalStates.GetNext( pos );
    }
    else

        nextSignalState = m_Link->m_signalStates.GetNext( pos );

    done = nextSignalState->signalCode != 0;
}

nextActualCode = nextSignalState->m_actualCode;

// Based on the current signal state and the next signal state,
// set the AMBSPC.
if( signalCode == 0 )
{
    switch( m_actualCode )
    {
        case 1:  // green ball
            AMBSPC = GetAMBSPC1( nextActualCode );
            break;
        case 2:  // <<REVIEW>> -- red ball?
            break;
        case 3:  // green right turn only
            AMBSPC = GetAMBSPC3( nextActualCode );
            break;
    }
break;  
case 4: // green left turn only  
AMBSPC = GetAMBSXC4( nextActualCode );  
break;  
case 5: // stop sign  
AMBSPC = GetAMBSXC5( nextActualCode );  
break;  
case 6: // green diagonal only  
AMBSPC = GetAMBSXC6( nextActualCode );  
break;  
case 7: // green thru only  
AMBSPC = GetAMBSXC7( nextActualCode );  
break;  
case 8: // green right and left turn only  
AMBSPC = GetAMBSXC8( nextActualCode );  
break;  
case 9: // green thru and right turn only  
AMBSPC = GetAMBSXC9( nextActualCode );  
break;  
default: // invalid code  
break;  
else  
{  
// red ball  
AMBSPC = 15; 
}  
}  

int CSignalState::GetAMBSXC1( int nextActualCode )  
{  
    // This routine is limited to use with the SampleRTE.trf file provided  
    // in this example. If other files are to be used this routine may  
    // need to be expanded to handle other values for the input argument  
    // nextActualCode.  
    // m_actualCode == 1  
    // green ball  
    int nextSDCODE = GetSDCODE( nextActualCode );  
    int ambspc = 15 - nextSDCODE;  

    return ambspc;  
}
int CSignalState::GetAMBSPC3( int nextActualCode )
{
    // This routine is limited to use with the SampleRTE.trf file provided
    // in this example. If other files are to be used this routine may
    // need to be expanded to handle other values for the input argument
    // nextActualCode.

    // m_actualCode == 3
    // green right turn only
    int ambspc = -1;

    int nextSDCODE = GetSDCODE( nextActualCode );
    int nextRightBit = nextSDCODE & 1;

    if( nextRightBit == 0 )
    {
        // next sdcodes allow for right turn
        ambspc = 15;
    }
    else if( nextRightBit > 0 )
    {
        // next sdcodes do not allow for right turn, so right turn bit
        // in ambspc must be set to 0
        ambspc = 14;
    }

    return ambspc;
}

int C SignalState::GetAMBSPC4( int nextActualCode )
{
    // This routine is limited to use with the SampleRTE.trf file provided
    // in this example. If other files are to be used this routine may
    // need to be expanded to handle other values for the input argument
    // nextActualCode.

    // m_actualCode == 4
    // green left turn only
    int ambspc = -1;

    int nextSDCODE = GetSDCODE( nextActualCode );

    int nextLeftBit1 = nextSDCODE & 16;
    int nextLeftBit0 = nextSDCODE & 8;

    return ambspc;
}
BOOL permittedLeftAllowed = (nextLeftBit1 > 0) && (nextLeftBit0 == 0);
BOOL protectedLeftAllowed = (nextLeftBit1 == 0) && (nextLeftBit0 == 0);

if( protectedLeftAllowed )
{
    // left turn is allowed by the next sdcode
    ambspc = 15;
}
else
{
    // left turn is not allowed or is only permissively
    // allowed by the next sdcode
    ambspc = 7;
}

return ambspc;
}

int CSignalState::GetAMBSPC5( int nextActualCode )
{
    // This routine is limited to use with the SampleRTE.trf file provided
    // in this example. If other files are to be used this routine may
    // need to be expanded to handle other values for the input argument
    // nextActualCode.

    // m_actualCode == 5 // <<REVIEW>>
    // stop sign
    int ambspc = -1;

    if( nextActualCode == 2 )
    {
        // stop sign to all red
        ambspc = 15;
    }

    return ambspc;
}

int CSignalState::GetAMBSPC6( int nextActualCode )
{
    // This routine is limited to use with the SampleRTE.trf file provided
    // in this example. If other files are to be used this routine may
    // need to be expanded to handle other values for the input argument
    // nextActualCode.

    // m_actualCode == 5 // <<REVIEW>>
    // stop sign
    int ambspc = -1;

    if( nextActualCode == 2 )
    {
        // stop sign to all red
        ambspc = 15;
    }

    return ambspc;
}
// m_actualCode == 6 // <<REVIEW>>
// green diagonal only
int ambspc = -1;

if( nextActualCode == 2 )
{
    // green diagonal only to all red
    ambspc = 11;
}

return ambspc;
}

int CSignalState::GetAMBSPC7( int nextActualCode )
{
    // This routine is limited to use with the SampleRTE.trf file provided
    // in this example. If other files are to be used this routine may
    // need to be expanded to handle other values for the input argument
    // nextActualCode.

    // m_actualCode == 7 // <<REVIEW>>
    // green thru only
    int ambspc = -1;

    int nextSDCODE = GetSDCODE( nextActualCode );

    int nextThruBit = nextSDCODE & 2;

    if( nextThruBit > 0 )
    {
        // thru is not allowed by the next sdcodes
        ambspc = 13;
    }
    else
    {
        // thru is allowed by the next sdcodes
        ambspc = 15;
    }

    return ambspc;
}

int CSignalState::GetAMBSPC8( int nextActualCode )
{
    // This routine is limited to use with the SampleRTE.trf file provided
// in this example. If other files are to be used this routine may
// need to be expanded to handle other values for the input argument
// nextActualCode.
// m_actualCode == 8  // <<REVIEW>>
// green right and left turn only
int ambspc = -1;
int nextSDCODE = GetSDCODE( nextActualCode );
int nextLeftBit0 = nextSDCODE & 8;
int nextLeftBit1 = nextSDCODE & 16;
int nextRightBit = nextSDCODE & 1;
BOOL permittedLeftAllowed = (nextLeftBit1 > 0) && (nextLeftBit0 == 0);
BOOL protectedLeftAllowed = (nextLeftBit1 == 0) && (nextLeftBit0 == 0);
ambspc = 15;
if( !permittedLeftAllowed )
{
    ambspc = ambspc & 7;
}
if( nextRightBit > 0 )
{
    ambspc = ambspc & 14;
}
return ambspc;
}

int CSignalState::GetAMBSPC9( int nextActualCode )
{
    // This routine is limited to use with the SampleRTE.trf file provided
    // in this example. If other files are to be used this routine may
    // need to be expanded to handle other values for the input argument
    // nextActualCode.
    // m_actualCode == 9
    // green right and thru only
    int ambspc = -1;
    int nextSDCODE = GetSDCODE( nextActualCode );
    int nextRightBit = nextSDCODE & 1;
    int nextThruBit = nextSDCODE & 2;
ambspc = 15;
if( nextRightBit > 0 )
{
    ambspc = ambspc & 14;
}
if( nextThruBit > 0 )
{
    ambspc = ambspc & 13;
}
return ambspc;
}

/****************************************************************************
 Module Name: upcntrl.cpp
 Description: Implementation file for the RTE interface functions.
****************************************************************************/
#include <stdio.h>
#include "stdafx.h"

// Include to provide access to the OutputString function.
#include "corwin.h"
#include "link.h"
#include "netsim.h"
#include "network.h"
#include "upcntrl.h"
#include <fstream.h>
#include "node.h"
#include "detector.h"
#include <windows.h>
#include <iostream.h>
#include <stdlib.h>
#include <stdio.h>
#define START_BYTE 48
#define MAX_BYTE   55

// Initialize global variables declared in upcntrl.h.
char gsOutput[132] = { '0' };
int giEndOfInit = 0;
int giPrevInit = 0;
int giPrevTime = 0;
int offsetNo;    //returned value from PIC to decide which offset should be chosen.
int offset1=0;
int offset2=50;
int offset3=55;
int offset4=105;
int initial1=1;   // number of times of in the signal sequence for Main Street green
int initial2=0;   // number of times of in the signal sequence for Side Street green

// Initialize a pointer to the network object used by the functions
// in this file.
CNetwork* pNetwork = NULL;
void ErrorHandler(char *message, DWORD error)
{
    // error process for the communication
    cout << message << endl;
    cout << "Error number = " << error << endl;
    ExitProcess(1);
}

int SerialComm(bool det)
{
    // communication function to send the detector data to traffic controller and
    // receive the signal states from controller
    HANDLE comHandle;
    BOOL success;
    DCB dcb;
    DWORD mask;
    int b;    // variable for BYTE changed to int.
    BYTE signalState;  // return the signal state by different offset point
    DWORD numWrite, numRead;
    COMMTIMEOUTS timeouts;
    BYTE detectorState;
    detectorState= (BYTE) det;   // force input BOOL det into BYTE type.

    // Open the comm port.
    comHandle = CreateFile("COM1",
        GENERIC_READ|GENERIC_WRITE,
        0, 0, OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL, 0);
    if (comHandle == INVALID_HANDLE_VALUE)
        ErrorHandler("In CreateFile",
            GetLastError());

    // Get the current settings of the COMM port
    success = GetCommState(comHandle, &dcb);
    if (!success)
        ErrorHandler("In GetCommState",
            GetLastError());

    // Modify the baud rate, etc.
    dcb.BaudRate = 9600;
    dcb.ByteSize = 8;
    dcb.Parity = NOPARITY;
    dcb.StopBits = ONESTOPBIT;
// Apply the new comm port settings
success = SetCommState(comHandle, &dcb);
if (!success)
    ErrorHandler("In SetCommState",
                GetLastError());

// Change the ReadIntervalTimeout so that
// ReadFile will return immediately.
timeouts.ReadIntervalTimeout = MAXDWORD;
timeouts.ReadTotalTimeoutMultiplier = 0;
timeouts.ReadTotalTimeoutConstant = 0;
timeouts.WriteTotalTimeoutMultiplier = 0;
timeouts.WriteTotalTimeoutConstant = 0;
SetCommTimeouts( comHandle, &timeouts );

// Set the Data Terminal Ready line
EscapeCommFunction(comHandle, SETDTR);

//write detector state to PIC
success = WriteFile(comHandle, &detectorState, 1,
        &numWrite, 0);
if (!success)
    ErrorHandler("In WriteFile", GetLastError());

//set the event to trigger the receiver
SetCommMask(comHandle,EV_RXCHAR);
//wait for occurance of event
WaitCommEvent(comHandle,&mask, 0);
//read data one by one from com1

success = ReadFile(comHandle, &signalState,
        100, &numRead, 0);
if (!success)
    ErrorHandler("In ReadFile", GetLastError());

//change the received data into int from ASSIC code.
b=(int) signalState;       //force BYTE type into int.
// Clear the DTR line
EscapeCommFunction(comHandle, CLRDTR);
CloseHandle(comHandle);
return b;
}

//*******************************************************************************
/ Implementation of the exported RTE interface initialization function.  
// CORSIM calls this function once at the beginning of the simulation.  
// **********************************************************************  
DLL_EXPORT void __stdcall INIT()  
{  
    // Set global variables.  
    giEndOfInit = 0;  
    giPrevInit = 0;  
    // Copy the CORSIM input file name from the imported string to a CString.  
    // Use the imported string length just in case the string is not null  
    // terminated.  
    CString strInputFileName( inputFileName, inputFileNameLength );  
    // Create the network object.  
    pNetwork = new CNetwork( strInputFileName );  
    // Read the traf file.  
    pNetwork->ReadTrafFile();  
    // process the controled node one by one  
    POSITION posN=pNetwork->m_NodeList.GetHeadPosition();  
    while (posN != NULL)  
    {  
        CNode* pNode=pNetwork->m_NodeList.GetNext(posN);  
        // get duration for this node  
        pNode->m_duration[0]=50;  
        pNode->m_duration[1]=2;  
        pNode->m_duration[2]=3;  
        pNode->m_duration[3]=50;  
        pNode->m_duration[4]=2;  
        pNode->m_duration[5]=3;  
        pNode->m_duration[6]=0;  
        pNode->m_duration[7]=0;  
        pNode->m_duration[8]=0;  
        pNode->m_duration[9]=0;  
        pNode->m_duration[10]=0;  
        pNode->m_duration[11]=0;  
        pNode->m_Link1=pNetwork->FindLink(4,1);  
        pNode->m_Link1->m_code[0]=2;  
        pNode->m_Link1->m_code[1]=2;  
        pNode->m_Link1->m_code[2]=2;  
        pNode->m_Link1->m_code[3]=1;  
        pNode->m_Link1->m_code[4]=0;  
        pNode->m_Link1->m_code[5]=2;  
    }  
}
pNode->m_Link2=pNetwork->FindLink(2,1);
pNode->m_Link2->m_code[0]=2;
pNode->m_Link2->m_code[1]=2;
pNode->m_Link2->m_code[2]=2;
pNode->m_Link2->m_code[3]=1;
pNode->m_Link2->m_code[4]=0;
pNode->m_Link2->m_code[5]=2;
pNode->m_Link2->m_code[6]=2;
pNode->m_Link3=pNetwork->FindLink(5,1);
pNode->m_Link3->m_code[0]=1;
pNode->m_Link3->m_code[1]=0;
pNode->m_Link3->m_code[2]=2;
pNode->m_Link3->m_code[3]=2;
pNode->m_Link3->m_code[4]=2;
pNode->m_Link3->m_code[5]=2;
pNode->m_Link4=pNetwork->FindLink(3,1);
pNode->m_Link4->m_code[0]=1;
pNode->m_Link4->m_code[1]=0;
pNode->m_Link4->m_code[2]=2;
pNode->m_Link4->m_code[3]=2;
pNode->m_Link4->m_code[4]=2;
pNode->m_Link4->m_code[5]=2;
}
pNetwork->CreateSignalStates();

sprintf( gsOutput, "\nRTE initialization complete\n" );
OutputString( gsOutput, strlen(gsOutput), SIM_COLOR_RGB, RTE_MESSAGE_RGB);
}

//******************************************************************************
// Implementation of the exported RTE interface main function.
// CORSIM calls this function at each time step in the simulation.
//******************************************************************************
DLL_EXPORT void __stdcall JMAIN()
{
    // Initialize the simulation time.
    int nTime = 0;

    // Initialization flag. CORSIM sets yinit to TRUE during initialization
    // and to FALSE after initialization has been completed.
    BOOL init = yinit;

    // The algorithm that controls the signal states at the intersections
    // assumes time is always increasing, but the CORSIM clock starts over
    // after initialization; so the time at which initialization is over must
    // be recorded.
if( (!init) && (giPrevInit) )
{
    // End of initialization.
    giEndOfInit = giPrevTime + 1;
}

// Adjust the time by adding the end of initialization.
    nTime = sclock + giEndOfInit;
// process the controled node one by one
POSITION posN=pNetwork->m_NodeList.GetHeadPosition();
while (posN != NULL)
{
    CNode* pNode=pNetwork->m_NodeList.GetNext(posN);
    if(pNode->m_id==1)                                 // if it is not the desired node skip it
    {
        posN=NULL;
        pNode->m_Link3->ProcessDetectors();    //Process the detector information
        pNode->m_Link4->ProcessDetectors();    //Process the detector information
    }
    //get detector in m_link3 of this node
    POSITION posD3=pNode->m_Link3->m_listOfDetectors.GetHeadPosition();
    CDetector* pDetector3=pNode->m_Link3->m_listOfDetectors.GetNext(posD3);
    //get detector in m_link4 of this node, because there is only one detector in this link
    POSITION posD4=pNode->mmlink4->m_listOfDetectors.GetHeadPosition();
    CDetector* pDetector4=pNode->m_Link4->m_listOfDetectors.GetNext(posD4);
    bool state1;
    if (pDetector3->m_flag==TRUE)
        state1=TRUE;
    else state1=FALSE;
    bool state2;
    if (pDetector4->m_flag==TRUE)
        state2=TRUE;
    else state2=FALSE;
    state2=state1||state2;
    offsetNo=SerialComm(state2);   //input the detector state to PIC controller
    switch(offsetNo){
        case 1: //If there is no vehicles coming from side street, Set main street
            // green.
            if (initial1<1)
                pNode->Setm_pos(offset2);
            else if(initial1>5)
                pNode->Setm_pos(offset3);
            initial1++;
            initial2=0;
            break;
case 2: //if there are vehicles coming from side street, serve side street for a
//minimum green
if (initial2<1)
pNode->Setm_pos(offset4);
else if(initial2>5)
pNode->Setm_pos(offset1);
initial1=0;
initial2++;
//reset flag for detector state in side street
pDetector3->m_flag=FALSE;
pDetector4->m_flag=FALSE;
break;
default: //invalid code
break;
}
}
}
pNetwork->UpdateNodeSignalStates();

// Record whether the simulation has reached equilibrium or not, so the
// time at which initialization can be recorded.
gIPrevInit = init;
gIPrevTime = nTime;
}

//**********************************************************************
// Implementation of the exported RTE interface exit function.
// CORSIM calls this function once at the end of the simulation.
//**********************************************************************
DLL_EXPORT void __stdcall JEXIT()
{
 // Clean up -- delete all objects that were created.
 sprintf( gsOutput, "RTE exit function - before delete\n" );
 OutputString( gsOutput, strlen(gsOutput), SIM_COLOR_RGB,
 RTE_MESSAGE_RGB );

delete pNetwork;

 sprintf( gsOutput, "RTE exit function - after delete\n" );
 OutputString( gsOutput, strlen(gsOutput), SIM_COLOR_RGB,
 RTE_MESSAGE_RGB );
}
**********************************************************************
Module Name: binarySequence.h
Description: Header file for the CBinarySequence class.

#ifndef CBinarySequence_H
#define CBinarySequence_H
class CInteger;

class CBinarySequence : public CObject
{
public:
    CBinarySequence();
    virtual ~CBinarySequence();

    // Data:
    CTypedPtrList<CPtrList, CInteger*> m_sequence;
};
#endif // CBinarySequence_H

Module Name: CORWin.h
Description: CORSIM-Windows Interface

#ifndef CORSIM_WINDOWS_INTERFACE
#define CORSIM_WINDOWS_INTERFACE

#ifdef CORWIN_IMPLEMENTATION
#define CORWINAPI __declspec(dllexport)
#else
#define CORWINAPI __declspec(dllimport)
#endif

/*/////////////////////////////////////////////////////////////////////////////
// Simulation Message Codes
////////////////////////////////////////////////////////////////////////////*/
#define SIM_YIELD            0x0000  /* yield time */
#define SIM_MESSAGE          0x0001  /* transfer text */
#define SIM_RTINIT           0x0002  /* call RT init */
#define SIM_RTUPDATE         0x0003  /* call RT main */
#define RT_MESSAGE           0x0004  /* transfer text */
#define SIM_GETKEYMESSAGE    0x0005  /* get user keyboard input */
#define SIM_ERRORMESSAGE     0x0006  /* transfer error message */
#define SIM_KILLPROC         0x0007  /* stop execution */
#define SIM_ADDVEHICLE       0x0008  /* add vehicle to message file */
#define SIM_LINKSTATEVALID   0x0009  /* add link to message file */
```c
#define SIM_STATUS 0x000a /* status of simulation */
#define SIM_ADDRAMPER 0x000b /* add ramp meter to message file */
#define SIM_LINKTIDVALID 0x000c /* current time interval data valid for link */
#define SIM_ADDINCIDENT 0x000d /* add incident to message file */
#define SIM_DELETEFILE 0x000e /* delete file */
#define SIM_SECTIONDATAREADY 0x000f /* section data valid */
#define WELCOME_MESSAGE 0x00ff /* message displayed on opening a project */
#define NEW_PROJECT_MESSAGE 0x00fe /* message displayed on creating a project */

/*************************************************************************************************
// Simulation Status Codes
/*************************************************************************************************
#define SIM_NORMAL 0x0000 /* OK */
#define SIM_ABORT 0x0001 /* aborting */
#define SIM_TIMESTEPCOMPLETE 0x0002 /* time step complete */
#define SIM_LINKCOMPLETE 0x0003 /* link data valid */
#define SIM_INVARIANTPROCESSCOMPLETE 0x0004 /* invariant data valid */
#define SIM_BEGININTERVAL 0x0005 /* start new time interval */
#define SIM_INTERVALCOMPLETE 0x0006 /* end current time interval */
#define SIM_BEGINTIMESTEP 0x0007 /* start new time step */
#define SIM_COMPLETE 0x0008 /* simulation is finished */
#define SIM_BEGINSIMULATION 0x0009 /* start of simulation data */
#define SIM_TIMEPERIOD_VALIDATION 0x000f /* time period validation complete */

/*************************************************************************************************
// Color Indices
/*************************************************************************************************
#define SIM_COLOR_RGB 0x0000 /* Use RGB value */
#define SIM_COLOR_INFO 0x0001 /* Normal */
#define SIM_COLOR_WARNING 0x0002 /* Warning */
#define SIM_COLOR_ERROR 0x0003 /* Error */

#ifdef __cplusplus
extern "C" {
#endif

/*************************************************************************************************
// Send Message
/*************************************************************************************************
CORWINAPI void __stdcall SendMsg(int msgID, int msgCode);

#ifdef __cplusplus
}
#endif

135
```
// Output String
//////////////////////////////////////////////////////////////////////////////*/
CORWINAPI void __stdcall OutputString(char* str, unsigned int size, int msgCode,
unsigned long color);
/*//////////////////////////////////////////////////////////////////////////////*/
// Message Box
//////////////////////////////////////////////////////////////////////////////*/
CORWINAPI void __stdcall MsgBox(char* message, unsigned int msize,
char* title, unsigned int tsize);
/*//////////////////////////////////////////////////////////////////////////////*/
// Request Keyboard Input
//////////////////////////////////////////////////////////////////////////////*/
CORWINAPI void __stdcall RequestKeyInput(char* prompt, unsigned int size);
#if defined(__cplusplus)
} /* extern C */
#endif
#if defined(_MSC_VER) && !defined(CORWIN_IMPLEMENTATION)
#pragma comment(lib, "CORWin")
#endif
#if defined(CORWIN_IMPLEMENTATION) || defined(NOT_RTE)
/*//////////////////////////////////////////////////////////////////////////////*/
// Message Identifiers
//////////////////////////////////////////////////////////////////////////////*/
#define WM_SIMMESSAGE (WM_USER + 100)
#define WM_OUTPUTMSG  (WM_USER + 102)
/*//////////////////////////////////////////////////////////////////////////////*/
// "This is a color code, not an RGB value" flag used by WM_OUTPUTMSG messages
//////////////////////////////////////////////////////////////////////////////*/
#define SIM_COLOR_CODE 0xff00
/*//////////////////////////////////////////////////////////////////////////////*/
// Set Window Handle
//////////////////////////////////////////////////////////////////////////////*/
#if defined(_MSC_VER) && !defined(CORWIN_IMPLEMENTATION)
extern "C"
#endif
CORWINAPI int __stdcall SetHWND(HWND hwnd);
#endif
136
class CDetector : public CObject
{
public:
    CDetector();
    virtual ~CDetector();

    int m_id;
    int m_length;
    CString m_type; // presence or pulse
    CLink* m_Link;
    int m_distanceFromDownstreamNode;
    CLane* m_Lane;
    int m_CorsimId;
    int m_count;
    BOOL m_state; // activated=TRUE or not=FALSE
    float m_activationTime;
    float m_deactivationTime;
    bool m_flag; // flag for detector state change.
};

#endif // CDetector_H
// Data.
    int data;
};

#endif // CInteger_H

/*******************************************************************************
 Module Name: lane.h
 Description: Header file for the CLane class.
*******************************************************************************/
#ifdef CLane_H
#define CLane_H
class CLane;
#endif // CLane_H

/*******************************************************************************
 Module Name: link.h
 Description: Header file for the CLink class.
*******************************************************************************/
#ifdef CLink_H
#define CLink_H
    class CNode;
    class CLane;
    class CDetector;
    class CBinarySequence;
#endif // CLink_H
class CSignalState;

class CLink : public CObject
{
public:
    CLink();
    virtual ~CLink();

    void ProcessDetectors( void );
    CBinarySequence* ConvertToBinary( int nValue );
    CLane* FindLane( int corsimId );
    float ComputeTravelTime( void );
    void CreateSignalStates( void );
    void SetSDCCode( void );
    void AdjustSignalState( void );

    int m_id;
    int m_CorsimId;
    CNode* m_upnode;
    CNode* m_dnnode;
    CNode* m_thrunode;
    CNode* m_leftnode;
    CNode* m_rightnode;
    CTypedPtrList<CPtrList,CLane*> m_listOfLanes;
    CTypedPtrList<CPtrList,CDetector*> m_listOfDetectors;
    CTypedPtrList<CPtrList,CSignalState*> m_signalStates;
    int m_length;
    int m_numOfFullLanes;
    int m_numOfLeftTurnBays;
    int m_numOfRightTurnBays;
    int m_lengthOfLeftBay;
    int m_lengthOfRightBay;
    int m_freeFlowSpeed;
    float m_travelTime;
    int m_leftMovementPercent;
    int m_thruMovementPercent;
    int m_rightMovementPercent;
    CString m_channelCode[7];
    int m_NumOfDetectors;
    int m_code[12];
    int m_offset;
    POSITION m_pos;
    CLink* m_opposingLink;
    int m_opposingID;
};
#ifndef _NETSIM_H_
#define _NETSIM_H_

// Define macros that make the import/export process easier.
#ifdef __cplusplus
// When compiling C++ code, the extern "C" is required to support the
// C-style interfaces between CORSIM and the Run-Time Extension.
#define DLL_IMPORT extern "C" __declspec( dllimport )
#define DLL_EXPORT extern "C" __declspec( dllexport )
#else
// When compiling C code, there is no need for the extern "C" keyword.
#define DLL_IMPORT extern __declspec( dllimport )
#define DLL_EXPORT __declspec( dllexport )
#endif // __cplusplus

// Define CORSIM array size parameters.
#define IMXNOD 8999
#define IMXDET 7000

// CORSIM input/output file specification.
DLL_IMPORT struct
{
    char linfname[512];
    char loutfname[512];
    int linflen;
    int loutflen;
} LIOFILES;

#define inputFileName LIOFILES.linfname
#define inputFileNameLength LIOFILES.linflen

DLL_IMPORT int NETSIM_LINKS_mp_IMXLNK;

// Import CORSIM dynamically_allocated arrays.
DLL_IMPORT int* NETSIM_LINKS_mp_SDCODE;
DLL_IMPORT int* NETSIM_LINKS_mp_DWNOD;
DLL_IMPORT int* NETSIM_LINKS_mp_UPNOD;

140
#include <stdio.h>
#include <stdlib.h>

int main() {
    int result = 0;
    return result;
}

#endif // _NETSIM_H_
/***************************************************************
 Module Name: network.h
 Description: Header file for the CNetwork class.
 ***************************************************************/*
#ifndef CNetwork_H
#define CNetwork_H

class CDetector;
class CLink;
class CNode;

class CNetwork : public CObject
{
 public:
 CNetwork();
 explicit CNetwork( const CString& strInputFileName );
 virtual ~CNetwork();

 inline CString GetTrafInputName( void ) const { return m_TrafInputFile; }
 inline void SetTrafInputName( const CString& strName ) { m_TrafInputFile = strName; }

 void ReadTrafFile( void );
 void GetNodes( FILE* pFile );
 CNode* FindNode( int id );
 void GetLinks( FILE* pFile );
 CLink* FindLink( int up, int dn );
 int GetLinkCorsimId( int upnode, int dnnode );
 void UpdateNodeSignalStates( void );
 void GetDetectors( FILE* pFile );
 void GetDetectorCorsimId( CDetector* pDetector );
 void CreateLanes( FILE* pFile );
 void ReadCard35( FILE* pFile );
 void ReadCard36( FILE* pFile );
 void CreateSignalStates( void );

 // Data.
 CTypedPtrList<CPtrList,CLink*> m_LinkList;
 CTypedPtrList<CPtrList,CNode*> m_NodeList;

 private:
 int ReadTRFLine( FILE* pFile, char* pszLine );

 // Data.
 CString m_NetworkName;
 CString m_TrafInputFile;
class CNode : public CObject
{
public:
    CNode();
    virtual ~CNode();

    void SetSDCCode( void );
    void SetSignalState( void );
    void Setm_pos(int offset);

    int    m_id;
    int    m_CORSIMInternalID;
    int    m_xPos;
    int    m_yPos;
    CLink* m_Link1;
    CLink* m_Link2;
    CLink* m_Link3;
    CLink* m_Link4;
    CLink* m_Link5;
    CString m_typeOfControl;
    int m_duration[12];
};

#include "node.h"
#include "signalState.h"
class CSignalState : public CObject
{
    public:
    CSignalState();
    virtual ~CSignalState();

    void SetSDCODE( void );
    int GetSDCODE( int actualCode );

    void SetAMBSPC( void );
    int GetAMBSPC1( int nextActualCode );
    int GetAMBSPC3( int nextActualCode );
    int GetAMBSPC4( int nextActualCode );
    int GetAMBSPC5( int nextActualCode );
    int GetAMBSPC6( int nextActualCode );
    int GetAMBSPC7( int nextActualCode );
    int GetAMBSPC8( int nextActualCode );
    int GetAMBSPC9( int nextActualCode );

    // Data - <<REVIEW>> should be private with accessor methods.
    int signalCode;       // code from the TRAF file card 36
    int m_actualCode;  // code used to actually set the SDCODE and
                       // AMBSPC
    int SDCODE;
    int AMBSPC;
    CLink* m_Link;
};

#endif // CSignalState_H

/*****************************************************************************/
/* Module Name: upcntrl.h */
/* Description: Header file for the RTE interface functions. */
/*****************************************************************************/
#ifndef_UPCNTRL_H_
#define_UPCNTRL_H_
// Define a color for display messages issued by the RTE.
#define RTE_MESSAGE_RGB 0x00116400L  // forest green

// Define a global character string used for displaying messages
// issued by the RTE.
extern char gsOutput[132];
// Declare global variables.
extern int giEndOfInit;
extern int giPrevInit;
extern int giPrevTime;

#endif // _UPCNTRL_H_

A.2 JMAIN function Sources Code for Gap-Out in upctrl.cpp

// **********************************************************************
// Implementation of the exported RTE interface main function.
// CORSIM calls this function at each time step in the simulation.
// **********************************************************************

DLL_EXPORT void __stdcall JMAIN()
{
    // Initialize the simulation time.
    int nTime = 0;
    // Initialization flag. CORSIM sets yinit to TRUE during initialization
    // and to FALSE after initialization has been completed.
    BOOL init = yinit;

    // The algorithm that controls the signal states at the intersections
    // assumes time is always increasing, but the CORSIM clock starts over
    // after initialization; so the time at which initialization is over must
    // be recorded.
    if( (!init) && (giPrevInit) )
    {
        // End of initialization.
        giEndOfInit = giPrevTime + 1;
    }

    // Adjust the time by adding the end of initialization.
    nTime = sclock + giEndOfInit;
    // process the controled node one by one
    POSITION posN=pNetwork->m_NodeList.GetHeadPosition();
    while (posN !=NULL)
    {
        CNode* pNode=pNetwork->m_NodeList.GetNext(posN);
        if(pNode->m_id==1)                               //if it is not the desired node skip it
        {
            posN=NULL;                                  //Once the node 1 is found, stop iteration
            pNode->m_Link2->ProcessDetectors();    //Process the detector information
        }
        
        //get detector in m_link4 of this node, because there is only one detector in this link
        POSITION posD2=pNode->m_Link2->m_listOfDetectors.GetHeadPosition();
        CDetector* pDetector2=pNode->m_Link2->m_listOfDetectors.GetNext(posD2);
    }
//input the detector state to PIC controller
offsetNo=SerialComm(pDetector2->m_deactivationTime);
switch(offsetNo){
    case 1: //If there is no vehicles coming from side street, Set main street green.
        if (initial1<1)
            pNode->Setm_pos(offset2);
        else if(initial1>5)
            pNode->Setm_pos(offset3);
            initial1++;  
            initial2=0;
        break;
    case 2: //if there are vehicles coming from side street, serve side street for a minimum green
        if (initial2<1)
            pNode->Setm_pos(offset4);
        else if(initial2>5)
            pNode->Setm_pos(offset1);
            initial1=0;
            initial2++;  
        break;
    default: //invalid code
        break;
}
}

pNetwork->UpdateNodeSignalStates();

// Record whether the simulation has reached equilibrium or not, so the // time at which initialization can be recorded.
giPrevInit = init;
giPrevTime = nTime;
}

A.3. JMAIN function Sources Code for Queue+Gap-Out in upctrl.cpp
// **********************************************************************
// Implementation of the exported RTE interface main function.
// CORSIM calls this function at each time step in the simulation.
// **********************************************************************
DLL_EXPORT void __stdcall JMAIN()
{
    // Initialize the simulation time.
int nTime = 0;
float *detData;  //detector data,detData[0] for 3rd detector,
                //detector on stop bar.
detData = new float[4];  // Initialization flag. CORSIM sets yinit to TRUE during initialization
                         // and to FALSE after initialization has been completed.
BOOL init = yinit;
// The algorithm that controls the signal states at the intersections
// assumes time is always increasing, but the CORSIM clock starts over
// after initialization; so the time at which initialization is over must
// be recorded.
if( (!init) && (giPrevInit) )
{
    // End of initialization.
    giEndOfInit = giPrevTime + 1;
}

// Adjust the time by adding the end of initialization.
nTime = sclock + giEndOfInit;
//process the controled node one by one
POSITION posN=pNetwork->m_NodeList.GetHeadPosition();
while (posN !=NULL) {
    CNode* pNode=pNetwork->m_NodeList.GetNext(posN);
    if(pNode->m_id==1)                  //if it is not the desired node skip it
    {
        posN=NULL;                       //Once the node 1 is found, stop iteration
        pNode->m_Link2->ProcessDetectors();  //Process the detector in main street
        pNode->m_Link3->ProcessDetectors();  //Process the detectors in side street
        CLink* pLink2 = NULL;
        CLink* pLink3 = NULL;
        pLink2=pNode->m_Link2;
        pLink3=pNode->m_Link3;
        //detectors for m_Link3
        POSITION posD3=pLink3->m_listOfDetectors.GetHeadPosition();
        while (posD3!=NULL) {
            CDetector* pDetector3=pLink3->m_listOfDetectors.GetNext(posD3);

            //get 3rd detector data
            //find the specific detectors in Link3
            if(pDetector3->m_distanceFromDownstreamNode==200)
                detData[0] = pDetector3->m_activationTime;
            else if(pDetector3->m_distanceFromDownstreamNode==100)
detData[1] = pDetector3->m_activationTime;
    //get 1st detector data
else if(pDetector3->m_distanceFromDownstreamNode==5)
detData[2] = pDetector3->m_activationTime;
}
    //detectors for m_Link2
POSITION posD2=pLink2->m_listOfDetectors.GetHeadPosition();           CDetector*  
pDetector2=pLink2->m_listOfDetectors.GetNext(posD2);
detData[3]=pDetector2->m_deactivationTime;
offsetNo=SerialComm(detData);   //input the detector state to PIC controller
switch(offsetNo){
    case 1:    //If there is no vehicles coming from side street, Set main street green.
        if (initial1<1)
            pNode->Setm_pos(offset2);
        else if(initial1>5)
            pNode->Setm_pos(offset3);
        initial1++;
        initial2=0;
        break;

    case 2:
        if (initial2<1)    //if there are vehicles coming from side street,
serve side street for a minimum green
            pNode->Setm_pos(offset4);
        else if(initial2>5)
            pNode->Setm_pos(offset1);
        initial1=0;
        initial2++;

        break;
    default: //invalid code
        break;
}
}
}
pNetwork->UpdateNodeSignalStates();
    // Record whether the simulation has reached equilibrium or not, so the
    // time at which initialization can be recorded.
giPrevInit = init;
giPrevTime = nTime;
DLL_EXPORT void __stdcall JMAIN()
{
    // Initialize the simulation time.
    int nTime = 0;
    // Initialization flag. CORSIM sets yinit to TRUE during initialization
    // and to FALSE after initialization has been completed.
    BOOL init = yinit;
    // The algorithm that controls the signal states at the intersections
    // assumes time is always increasing, but the CORSIM clock starts over
    // after initialization; so the time at which initialization is over must
    // be recorded.
    if( (!init) && (giPrevInit) )
    {
        // End of initialization.
        giEndOfInit = giPrevTime + 1;
    }
    // Adjust the time by adding the end of initialization.
    nTime = sclock + giEndOfInit;

    // process the controled node one by one
    POSITION posN=pNetwork->m_NodeList.GetHeadPosition();
    while (posN !=NULL)
    {
        CNode* pNode=pNetwork->m_NodeList.GetNext(posN);

        if(pNode->m_id==1)                               //if it is not the desired node skip it
        {
            mindeactivationTime=1000.0;    //reset mindeactivationTime as a big
                                                //number in each time step
            posN=NULL;
            pNode->m_Link2->ProcessDetectors();    //Process the detector
                                                //information
            //get detector in m_link2 of this node, because there is only one detector in
            //this link
            //it is not necessary to use iteration
            POSITION posD2=pNode->m_Link2->m_listOfDetectors.GetHeadPosition();
            while (posD2 !=NULL){

CDetector* pDetector2 = pNode->m_Link2->m_listOfDetectors.GetNext(posD2);

// find the minimum deactivation time for 3 detectors and compare it with the gap
if (mindeactivationTime > pDetector2->m_deactivationTime)
    mindeactivationTime = pDetector2->m_deactivationTime;
}

// end of detector information processing
offsetNo = SerialComm(mindeactivationTime);  // input the detector state to PIC controller
switch(offsetNo) {
    case 1:
        if (initial1 < 1)  // If there is no vehicles coming from side street, Set main street green.
            pNode->Setm_pos(offset2);
        else if (initial1 > 5)
            pNode->Setm_pos(offset3);
        initial1++;
        initial2 = 0;
        break;
    case 2:
        if (initial2 < 1)  // if there are vehicles coming from side street, serve side street for a minimum green
            pNode->Setm_pos(offset4);
        else if (initial2 > 5)
            pNode->Setm_pos(offset1);
        initial1 = 0;
        initial2++;
        break;
    default:  // invalid code
        break;
}
}
pNetwork->UpdateNodeSignalStates();
// Record whether the simulation has reached equilibrium or not, so the time at which initialization can be recorded.
giPrevInit = init;
giPrevTime = nTime;
E.5. JMAIN function Sources Code for DilemmaZone Control in upctrl.cpp

//************************************************************************
// Implementation of the exported RTE interface main function.
// CORSIM calls this function at each time step in the simulation.
//**********************************************************************

DLL_EXPORT void __stdcall JMAIN()
{
    // Initialize the simulation time.
    int nTime = 0;

    int *detData;
    detData = new int[9];

    // Initialization flag. CORSIM sets yinit to TRUE during initialization
    // and to FALSE after initialization has been completed.
    BOOL init = yinit;

    // The algorithm that controls the signal states at the intersections
    // assumes time is always increasing, but the CORSIM clock starts over
    // after initialization; so the time at which initialization is over must
    // be recorded.
    if( (!init) && (giPrevInit) )
    {
        // End of initialization.
        giEndOfInit = giPrevTime + 1;
    }

    // Adjust the time by adding the end of initialization.
    nTime = sclock + giEndOfInit;

    // Modified

    POSITION posN=pNetwork->m_NodeList.GetHeadPosition();
    //process the controled node one by one

    while (posN != NULL)
    {
        
    }
}

CNode* pNode=pNetwork->m_NodeList.GetNext(posN);

if(pNode->m_id==1)
{
    posN=NULL;       //after finding the node 1, stop looping
    pNode->m_Link2->ProcessDetectors();
}

POSITION posD2=pNode->m_Link2->m_listOfDetectors.GetHeadPosition();
while (posD2!=NULL)
{
    // get all detectors' information
    CDetector* pDetector=pNode->m_Link2->m_listOfDetectors.GetNext(posD2);
    CLane* pLane=pDetector->m_Lane;
    if(pDetector->m_distanceFromDownstreamNode==600&&pLane->m_id==2)
        {// 1st detector information
            // get deactivationTime for gap out (optional)
            detData[8]=pDetector->m_deactivationTime;
        }
    if(pDetector->m_distanceFromDownstreamNode==550&&pLane->m_id==2)
        {// 2nd detector
            detData[0]=pDetector->m_count;
            detData[4]=pDetector->m_speed;
        }
    if(pDetector->m_distanceFromDownstreamNode==400&&pLane->m_id==2)
        {// 3rd detector
            detData[1]=pDetector->m_count;
            detData[5]=pDetector->m_speed;
        }
    if(pDetector->m_distanceFromDownstreamNode==200&&pLane->m_id==2)
        {// 4th detector
            detData[2]=pDetector->m_count;
            detData[6]=pDetector->m_speed;
        }
    if(pDetector->m_distanceFromDownstreamNode==100&&pLane->m_id==2)
        {// 5th detector
            detData[3]=pDetector->m_count;
            detData[7]=pDetector->m_speed;
        }
}

// the Green need refresh each time step according to the time base change.
// probably the extension calculation is missed because the detector malfunction
offsetNo=SerialComm(detData);   // input the detector state to PIC controller
switch(offsetNo){
    case 1: // If there is no vehicles coming from side street, Set main street green.
        if (initial1<1)
            pNode->Setm_pos(offset2);
        else if(initial1>5)
            pNode->Setm_pos(offset3);
        initial1++;   
        initial2=0;   
        break;
    case 2: // if there are vehicles coming from side street, serve side street for a minimum green
        if (initial2<1)
            pNode->Setm_pos(offset4);
        else if(initial2>5)
            pNode->Setm_pos(offset1);
        initial1=0;
        initial2++;   
        break;
    default: // invalid code
        break;
    }
}

pNetwork->UpdateNodeSignalStates();

// Record whether the simulation has reached equilibrium or not, so the // time at which initialization can be recorded.
giPrevInit = init;
giPrevTime = nTime;
}
APPENDIX B

SOURCES CODE FOR PIC18F452

B1. //*****************************************************************
   Module Name: serial.c
   Description: Implementation file for serial communication.
   //*****************************************************************
#include <pic18.h>
#include "serial.h"

void InitUART(unsigned char baudRate, unsigned char ioMode){
   TXSTA=0x06;
   RCSTA=0x90;    //initialize registers
   SPBRG=0x19;
   TRISC=0x80;
   TXEN=1;
}

char getche(void){
   char TEMP;      //declare local variable to hold character
   TEMP=getch();    //get character and save in TEMP
   putch(TEMP);      //echo character to PC
   return TEMP;       //return the character in TEMP
}

char getch(void){
   int i;
   while(!RCIF){
      if(OERR){
         TXEN=0;
         TXEN=1;
         CREN=0;
         CREN=1;
      }
      if(FERR){
         char dummy;
         

154
{TXEN=0;
 TXEN=1;
}                          //wait until character is received
return RCREG;                 //return character

void putch(char transmit){

  while(!TXIF){
    if(OERR){
      if(OERR){
        TXEN=0;
        TXEN=1;
        CREN=0;
        CREN=1;
      }
      if(FERR){
        char dummy;
        TXEN=0;
        TXEN=1;
      }
    }                        //wait until TX register is empty
    //perform error checking

    TXREG=transmit;           //load character into register
    _60us();                 //provide 60us delay
  }

bit kbhit(void){
  _60us();
  return RCIF;              //return status of RCIF bit
}

void _60us(void){
  int count;
  for(count=600;count<1;count--){
    asm("nop");              //60 microsecond delay
  }
}

B2. //******************************************************************************
Module Name: serial.h
Description: header file for the communication.
/*************************************************************/
#ifndef SERIAL_H
#define SERIAL_H

#define BPS9600 (unsigned char)0
#define BPS19200 (unsigned char)1
#define BPS57600 (unsigned char)2

#define POLL (unsigned char)0
#define RONLY (unsigned char)1
#define TXONLY (unsigned char)2
#define TXRC (unsigned char)3

void InitUART(unsigned char, unsigned char);
char getche(void);
char getch(void);
void putch(char);
void _60us(void);
bit kbhit(void);

#endif

B3. Files for Gap Out Control

/********************************************************************************
 * Module Name: poll.c
 * Description: Implementation file for Gap Out control
 *********************************************************************************/

#include <pic18.h>
#include <stdio.h>
#include <conio.h>
#include "serial.h"

__CONFIG(1,OSCSDIS&HS);
__CONFIG(2,BORDIS&PWRTEN&WDTDIS);
__CONFIG(4,LVPDIS&DEBUGEN);

//these variables should be global
#define max1 (unsigned int)40          //max green for main street
#define max2 (unsigned int)20          //max green for side street
#define min1 (unsigned int)10
#define min2 (unsigned int)10
#define gap (float)2.0

unsigned char GapOutControl(unsigned int nTime, float DETdeactiveTime);
long int sideStTime=0;
long int mainStTime=0;
long int clock=0;                      //time in PIC side

void main( ){

unsigned char rec;
int i;
float DETdeactiveTime;
unsigned char signalState;

// initialize UART
    InitUART(BPS9600, POLL);

//endless loop
mainloop:
    //receiving data and put them into buffer
    rec=getch();
    DETdeactiveTime=(float) rec;
    clock=clock+1;
    //send data to PC after finishing processing data
    signalState=GapOutControl(clock, DETdeactiveTime);
    putch(signalState);
    goto mainloop;
}

unsigned char GapOutControl(unsigned int nTime, float DETdeactiveTime){
    int code=0;
    if(nTime>sideStTime+min2)   //max2 also is the start delay for main street green.
                                 //bypass the detector
    {
        mainStTime=sideStTime+min2; //the stop time of sidestreet green time is the start
time for main green
        if(nTime>mainStTime+min1)  //judge if the main street green time is greater than
min1
        {
            //if greater than min1, it can be gap out.
            //guarantee the minimum green is served for main street.
            if(DETdeactiveTime>gap)  //If deactive time is greater than gap, gap out
            {
                code=2;
                sideStTime=nTime;     //record the start time for side street
            }
            else //If there is no vehicles coming from side street, Set main street green.
            code=1;
        }
    }
}

code=1;
    }
else                          //if minimum green is not finished in side street, keep serving
    code=2;
    return code;
}.

B4. Files for Green Rest Control
//********************************************************************
Module Name: poll.c
Description: Implementation file for Green Rest control
//********************************************************************
#include <pic18.h>
#include <stdio.h>
#include <conio.h>
#include "serial.h"
__CONFIG(1,OSCSDIS&HS);
__CONFIG(2,BORDIS&PWRTEN&WDTDIS);
__CONFIG(4,LVPDIS&DEBUGEN);

unsigned char GreenRestControl(int nTime, int detState);

//these variables should be global
static unsigned int max1=40;        //max green for main street
static unsigned int max2=20;        //max green for side street
static unsigned int min1=10;
static unsigned int min2=15;
long int sideStTime=0;
long int mainStTime=0;
long int clock=0;                           //time in PIC side
void main( ){

    unsigned char rec;
    int i;
    unsigned char detState;
    unsigned char signalState;
    // initialize UART
    InitUART(BPS9600, POLL);
    //endless loop
    mainloop:
    //receiving data and put them into buffer
    rec=getch();
    detState=(int) rec;
clock=clock+1;
//send data to PC after finishing processing data
signalState=GreenRestControl(clock, detState);
putch(signalState);
goto mainloop;
}
unsigned char GreenRestControl(int nTime, int detState){
    int code=0;
    if(nTime>sideStTime+min2)       //max2 also is the start delay for main street green.
        //bypass the detector
        
        
    mainStTime=sideStTime+min2;// start time for main green
    if(nTime>mainStTime+max1)   // if the main street green time greater than max
        {
            if(detState==1)           //vehicles passing detector,change the signal
                {
                    code  =2;
                    sideStTime=nTime; //record the start time for side street
                }
        }else // no vehicles coming from side street, Set main street green.
            code=1;
    }
    else                //main street green less than max green,keep serving main street
        code=1;
    }
    else                          //minimum green is not finished in side street, keep serving
        code=2;     //return code;
}

B5. Files for Queue@GapOut Control
//********************************************************************
Module Name: poll.c
Description: Implementation file for Queue@GapOut control
//********************************************************************
#include <pic18.h>
#include <stdio.h>
#include "conio.h"
#include "serial.h"

_CONFIG(1,OSCSDIS&HS);
_CONFIG(2,BORDIS&PWRTEN&WDTDIS);
_CONFIG(4,LVPDIS&DEBUGEN);

unsigned char QueueGapOutControl(int nTime, float *detTime);
/these variables should be global
unsigned char max1=40;  //max green for main street
unsigned char max2=40;  //max green for side street
unsigned char min1=10;
unsigned char min2[3]={30,20,10};
unsigned char minNo=0;  //variable to choose which minimum green for side
//street depending on the queue length.
long int sideStTime=0;
long int mainStTime=0;
long int clock=0;       //time in PIC side

void main( ){

    unsigned char rec[4];
    int i;
    float detTime[4];   //deactivation time for detector in main street
    //and activation time for 3 detectors in side street
    unsigned char signalState;
    // initialize UART
    InitUART(BPS9600, POLL);
    //endless loop
    mainloop:
    //receiving data and put them into buffer
    for (i=0;i<4;i++){
        rec[i]=getch();
        detTime[i]=(float)rec[i]; //received detector time for both main and side street
    }
    clock=clock+1;
    //send data to PC after finishing processing data
    signalState=QueueGapOutControl(clock, detTime);
    putch(signalState);  //send signal state back
    goto mainloop;
}

unsigned char QueueGapOutControl(int nTime, float *detTime){
    int code=0;
    int i;
    if(nTime>sideStTime+min2[minNo])      //serve minimum green for side street
        mainStTime=sideStTime+min2[minNo];  // start time for main green
    if(nTime>mainStTime+min1)       //if the main street green time greater than min1
    }
if(detTime[3]>2) { //gap out for main street
    //iterate detector data by order to guarantee 3rd,2nd,1st detector order
    for (i=0;i<3;i++)
        {
            if(detTime[i]>0.5) // no vehicles passing detector
                {
                    minNo=i; //chosen minimum green
                    code=2; //switch to side street
                    sideStTime=nTime; //record the start time for side street
                    break;//if higher order detector is triggered,ignore following detectors
                }
        }
    if(code!=2) // no vehicles coming from side street, Set main street green.
        {
            code=1; //no gap out,keep green in main street
        }
} else //main street green less than min1,serve main street
    code=1;
else{
    code=2; //minimum green not finished in side street, keep serving
}
return code;

B6. Files for Volume Density Control

#include <pic18.h>
#include <stdio.h>
#include <conio.h>
#include "serial.h"

__CONFIG(1,OSCSDIS&HS);
__CONFIG(2,BORDIS&PWRTEN&WDTDIS);
__CONFIG(4,LVPDIS&DEBUGEN);

//these variables should be global
#define max1 (float)100.0 //max green for main street
#define max2 (float)20.0                                  //max green for side street
#define min1 (float)10.0
#define min2 (float)10.0
#define MaxGap (float)7.0
#define MinGap (float)1.0
#define TimeBeforeReduce (float)20.0
#define TimeToReduce (float) 10.0

unsigned char volumeDensityControl(unsigned int nTime, float DETdeactiveTime);
//parameters for VolumeDensity control
float TimeReduced=(MaxGap-MinGap)/TimeToReduce;      //reduced gap in each second.
float vehicleGap=MaxGap;                             //set start gap as maximum gap
long int sideStTime=0;
long int mainStTime=0;
long int clock=0;                                              //time in PIC side

void main( ){
  unsigned char rec;
  int i;
  float DETdeactiveTime;
  unsigned char signalState;
  // initialize UART
  InitUART(BPS9600, POLL);
  //endless loop
  mainloop:
  //receiving data and put them into buffer
  rec=getch();
  DETdeactiveTime=(float) rec;
  clock=clock+1;
  //send data to PC after finishing processing data
  signalState=volumeDensityControl(clock, DETdeactiveTime);
  putch(signalState);
  goto mainloop;
}

unsigned char volumeDensityControl(unsigned int nTime, float DETdeactiveTime){
  int code=0;
  if(nTime>sideStTime+min2)        //max2 also is the start delay for main street green.
  {
    mainStTime=sideStTime+(int)min2;      // start time for main green
    if (nTime>mainStTime+(int)TimeBeforeReduce)  //if the main street green time
      is greater than initial green
    {   //if greater than initial green, it can be gap out.
      //guarantee the minimum green is served for main street.
        if(nTime>mainStTime+(int)max1)
    }  
  }  
}
{  
    code=2;  
    vehicleGap=MaxGap;  //the vehicleGap is reset as MaxGap.  
    sideStTime=nTime;  //record the start time for side street  
}
else
{
    if(DETdeactiveTime>vehicleGap)  //If deactive time greater than gap, gap out
    {
        code=2;  
        vehicleGap=MaxGap;  //the vehicleGap is reset as MaxGap.  
        sideStTime=nTime;  //record the start time for side street  
    }
    else
    {  //reduce gap,green still in main street  
        vehicleGap=vehicleGap-TimeReduced;  
        if(vehicleGap<MinGap)  
            vehicleGap=MinGap;  
            code=1;  
    }
}
}
else  //if main street green less than max green,keep serving main street  
    code=1;  
}
else  //minimum green not finished in side street, keep serving  
    code=2;  
    return code;
}

B6. Files for Dilemma Zone Control
//********************************************************************
Module Name: poll.c
Description: Implementation file for Dilemma Zone control
//********************************************************************
#include <pic18.h>
#include <stdio.h>
#include <conio.h>
#include "serial.h"
__CONFIG(1,OSCSDIS&HS);
__CONFIG(2,BORDIS&PWRTEN&WDTDIS);
__CONFIG(4,LVPDIS&DEBUGEN);
//these variables should be global
#define max1 (float)100.0     //max green for main street
#define max2 (float)20.0     //max green for side street
#define min1 (float)15.0
#define min2 (float)10.0
#define MaxGap (float)7.0
#define MinGap (float)1.0
long int sideStTime=0;
long int mainStTime=0;
long int clock=0;               //time in PIC side
int ExtensionCalculation(int YellowAllRed, int currentTime, int speed, int distance);
int maximum(int *array, int n);
unsigned char dilemmaZoneControl(unsigned int nTime, int *det);

int YellowAllRed=5;
int width=24;
int l=20;
float deactTime=0;
int g=0;               //flag for set detectorTime
int detectorCt=0;
int endOfGreen[4]={0,0,0,0};
int distance[4]={550,400,200,100}; //trailing edge distance for each detector
int oldCount[4]={0,0,0,0};  //record the count for previous time step for 4 detectors
int Extension[4]={0,0,0,0}; //calculated extension time for each detector
int Green=0;

void main( ){
    unsigned char rec[9];
    int i;
    int det[9];
    unsigned char signalState;
    // initialize UART
    initUART(BPS9600, POLL);
    //endless loop
    mainloop:
    //receiving data and put them into buffer
    for (i=0;i<9;i++){
        rec[i]=getch();
        det[i]=(int)rec[i]; //received detector time for both main and side street
    }
    //detTime[0]:count for 4th detector in main street
    //detTime[1]:count for 3rd detector in main street
    //detTime[2]:count for 2nd detector in main street
    //detTime[3]:count for 1st detector in main street
    //detTime[4]:speed for 4th detector in main street
}
clock=clock+1;
//send data to PC after finishing processing data
signalState=dilemmaZoneControl(clock, det);
putch(signalState);
goto mainloop;
}

int ExtensionCalculation(int YellowAllRed, int currentTime, int speed,int distance)
{
    double GoZone=0.0;
    int newExt=0;
    int l=20;                //vehicle length.
    int width=24;            //half of width for intersection.
    if (speed==0)  //if speed is zero, just skip it and don't give extension.
        newExt=0;
    else{
        GoZone=speed*YellowAllRed*1.47-width-l;
        if (distance>GoZone)
            newExt=(int)(distance-GoZone)/speed;    //in dilemma zone
        else
            newExt=0;                               //in GoZone
    }
    return newExt;
}

//function to find maximum in an array
int maximum(int *array, int n){
    int i;
    int maximum=*array;
    for (i=1;i<n;i++)
    {
        if(*(array+i)>maximum)
            maximum=*(array+i);
    }
    return maximum;
}

unsigned char dilemmaZoneControl(unsigned int nTime, int *det){

    int code=0;
    if(nTime>sideStTime+(int)min2)
//green in main st
mainStTime=sideStTime+(int)min2;  // the start time for main green
code=1;                           //switch to main st
if(nTime>mainStTime+(int)min1)
  { //guarantee the min green for main st.
    if(nTime<mainStTime+(int)max1) //run to max green
      {
        if (*det+8)<MaxGap) //for deactivation time of 4th detector
          {
            if(oldCount[0]!=*det) //if gap out, skip the extension process for the detectors
              {
                if(g==0)
                  {
                    detectorCt=*det; //for 1st detector, only inialize it one time for each phase other
                                  // detectors in charge of updating
                    g++;                        //need to reinitilzed as 0 in side st green
                    oldCount[0]=*det;
                    Extension[0]=ExtensionCalculation(YellowAllRed,nTime,*det+4,distance[0]);
                    endOfGreen[0]=nTime+Extension[0];
                  }
        }
        oldCount[1]=*det+1;
        Extension[1]=ExtensionCalculation(YellowAllRed,nTime,*det+5,distance[1]);
        endOfGreen[1]=nTime+Extension[1];
      }
    if(oldCount[1]!=*(det+1)&&(det+1)>=detectorCt)
      {
        if(*(det+1)>=detectorCt) //new veh. coming, if malfunction for this detector.
          {
            detectorCt=*(det+1);  //cannot detect the new coming veh, update detectorCt
            oldCount[1]=*(det+1);
            Extension[1]=ExtensionCalculation(YellowAllRed,nTime,*det+6,distance[2]);
            endOfGreen[1]=nTime+Extension[1];
          }
    }
  }
//new veh. coming
if(oldCount[2]!=*(det+2)&&(det+2)>=detectorCt)
  {
    if(*(det+2)>=detectorCt)
      {
        detectorCt=*(det+2);
        oldCount[2]=*(det+2);
        Extension[2]=ExtensionCalculation(YellowAllRed,nTime,*det+7,distance[3]);
        endOfGreen[2]=nTime+Extension[2];
      }
  }
if(oldCount[3]!=*(det+3)&&(det+3)>=detectorCt)
  {
    //new veh. coming
if(*(det+3) >= detectorCt)
detectorCt=*(det+3);
detectorCt=detectorCt+1; //update count for following veh extension
oldCount[3]=*(det+3);
Extenslion[3]=ExtensionCalculation(YellowAllRed,nTime,*(det+7),distance[3]);
endOfGreen[3]=nTime+Extenstion[3];
}
}
Green=maximum(endOfGreen,4);
if(nTime > Green)
    { //switch to side st by gap out
        g=0;
        code=2;
        sideStTime=nTime;
    }
}
else
    { //max out
        code=2;
        g=0;
        sideStTime=nTime;
    }
}/min green for main st
}
else //if minimum green is not finished in side street, keep serving
    code=2;
return code;