12-31-2003

HDL implementation of a controller area network node

Anthony Richard Marino
Rowan University

HDL Implementation of a Controller Area Network Node
by

Anthony Richard Marino

A Thesis Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Department: Electrical and Computer Engineering
Major: Engineering (Electrical Engineering)

Approved:

Members of the Committee

_____
In Charge of Major Work

_____

_____
For the Major Department

_____

_____
For the College

Rowan University
Glassboro, New Jersey
2003

TABLE OF CONTENTS

List of Figures

List of Tables

Abstract

With increasing technology inside automobiles, especially law enforcement vehicles, there is a growing need for centralized wiring and device interconnection methods. Each new technology, such as GPS navigation units and in-vehicle video systems, includes a corresponding wiring harness that needs installation. The summation of all these hardwired technologies leads to large volumes of wire that negatively affect safety, power demands, and future design opportunities. A new trend among automotive manufacturers is to design and implement innovative networks that allow for shared communications between technologies.

This thesis presents the design, development, and implementation of an in-vehicle network that will centralize connections involved with the New Jersey State Police's pursuit light package currently installed in their troop cars. The intent is to show that if the wiring involved with such a system can be reduced, then other systems can follow suit. The network used is the controller area network (CAN) that was developed in the 1980s by Bosch. The work presented in this thesis is supported through the joint sponsorship of the New Jersey State Police (NJSP), New Jersey Department of Transportation (NJDOT), and the Federal Highways Administration (FHWA). Promising results have been obtained using this approach and should improve driver safety and assist NJSP with technology modernization and management within their fleet of troop cars.

Acknowledgements

I would like to take this opportunity to express my sincerest appreciation to my advisor, Dr. John Schmalzel. His relentless devotion to see me succeed has not gone unnoticed and I will always be thankful for that. He has been a mentor and a friend for the last five years and I look forward to keeping his company in the future.

As for my co-advisor, Dr. Shreekanth Mandayam, I would like to thank him for his encouragement, knowledge, and helpful feedback. Of course I can't forget to mention his ability to keep the mood light, even during my most trying times. Also I would like to thank Dr. Peter Mark Jansson for serving on my committee and providing me with useful feedback.

Tremendous gratitude to the joint partnership of the New Jersey State Police, New Jersey Department of Transportation, Federal Highway Administration, and Rowan University. This collection of agencies has given me an opportunity to work on a project that has eclipsed any lab experience and can someday prove useful in the real world.

The past five years have been demanding and rewarding, not just for me but for my family as well. I'd like to thank my family for supporting me in all my endeavors and allowing me to pursue my goals. Also I'd like to thank my girlfriend, Dana, who has been a steadying and calming force throughout this five year journey.

Finally, I would like to acknowledge my fellow colleagues who I have formed a strong friendship with during this experience. All of you have made this experience an unforgettable one.

Thank you to all who have made this possible.

Chapter 1: Introduction

Over the past ten years, electronic systems have steadily replaced or augmented many mechanical systems in automobiles. This is mainly the result of environmental concerns that require improved vehicle operating conditions. To help monitor the conditions of a vehicle there are numerous sensors and electronic control units (ECUs) that monitor various emissions related parameters. Effective control of vehicle efficiency requires ECUs to share information. For example, sensors and actuators sense the operation of specific components (e.g., oxygen sensor) and actuate others (e.g., fuel injectors) to maintain optimal engine control. In its simplest terms, today's vehicles contain hundreds of circuits and other electrical entities. In fact, it is estimated that today's cars can have up to 4 kilometers of wiring as compared to the 45 meters in cars manufactured in 1955 [1].

Aside from the in-vehicle telematics, which is the use of computers and telecommunications to enhance automotive environments, and electronics of an everyday passenger car, law enforcement vehicles require additional technologies. Needs differ with each law enforcement agency but usually include radar systems, in-vehicle video systems, and mobile computing devices. As can be imagined, each additional technology introduces a new set of wires and harnesses. Eventually the accumulation of these hardwired devices has implications on vehicle integrity and power demands. With each 50 kg of wiring, fuel consumption is increased by 0.2 liters per 100 km traveled [1]. Today's typical vehicle can contain between 37-91 kg of wiring as compared to 4 kg 50 years ago [2]. Automotive manufacturers have made strides to reduce wiring costs by creating in-vehicle networks to help centralize device connections and wiring to be able to transfer data from one electronic module to another over a serial bus.

1.1 In-Vehicle Networks

The ability to reduce and centralize wiring among the various electronic systems is essential for improving functionality and efficiency of a car, especially a law enforcement vehicle, with the focus on creating in-vehicle networks that facilitate information sharing between the distributed systems within the vehicle, and reducing point-to-point wiring. Much like a local area network (LAN) that connects computers together, an in-vehicle network connects the electronic systems of a car together. These networks are mostly based on serial protocols and are designed to replace point-to-point wiring. The goal of any in-vehicle protocol is to link and share information with the distributed electronic systems that make up the three networks (Class A, Class B, and Class C) of a vehicle.

1.1.1 In-Vehicle Network Classifications

The Society of Automotive Engineers (SAE) has established three basic categories for standards and protocols: Class A, Class B, and Class C [3]. These divisions are based on network speed and function as shown in Table 1.1. As communication

Table 1.1 – SAE classifications of in-vehicle networks from [3].

Class	Attributes
Class A	• Low speed (≤10 kb/s) • Convenience applications (entertainment, radio, etc.)
Class B	• Medium speed (10 kb/s to 125 kb/s) • Body electronics and diagnostics (vehicle speed, instrument cluster, emissions data, etc.)
Class C	• High speed (125 kb/s to 1 Mb/s or greater) • Vehicle dynamics (real time control, power train control, brake by wire, etc.)

performance increases, it is expected that in-vehicle networks will eclipse the 1 Mb/s barrier, and thus a Class D will be required. SAE has not yet defined standards for Class D networks; however, networks which exceed a data rate of 1 Mb/s are often referred to as Class D networks.

1.1.1.1 Class A Networks

Class A maintains the lowest data rate. For this classification the data rate peaks as high as 10 kb/s. The low speed protocols in this category typically use generic UARTs that are not standardized and are often proprietary. In general, Class A devices support convenience electronics, such as audio and comfort controls.

1.1.1.2 Class B Networks

Class B networks support data rates in the range of 10 – 100 kb/s. These networks typically promote non-real time control and general information sharing between nodes. The utilization of Class B can eliminate redundant sensors and other systems by providing a means to transfer data from node to node. SAE J1850 (section 1.2.1.1) is a typical Class B network used for diagnostic purposes, which can also be used for other electronics applications.

1.1.1.3 Class C Networks

Class C is the fastest of the three network categories. With performance capability up to 1 Mb/s, Class C facilitates distributed control associated with real-time control systems, such as power train control. The predominant Class C protocol is CAN (section 1.2.1.4).

1.2.1 Current In-Vehicle Networks

A few of the established and emerging in-vehicle networks are listed here:

1) On board diagnostics (OBD)

2) Local interconnect network (LIN)

3) Media-oriented systems transport (MOST)

4) Controller area network (CAN)

There is no single standard or one protocol that represents a solution to connect all the various applications presented by a vehicle. Therefore there are multitudes of networks with different features that are better suited for particular applications. For example, a low speed network may be best for non-critical applications, whereas a high-speed network is best for real time, critical applications. These points will be discussed in the following sections.

1.2.1.1 On Board Diagnostics (OBD)

On board diagnostic networks were originally developed in the 1980's to offer assistance in troubleshooting problems with the emissions-related components of computerized engine systems. The original OBD, a creation of the Environmental Protection Agency (EPA) has evolved into OBD II keeping pace with stricter emissions regulations. OBD II can identify problem(s) with the emissions control system before excessive emissions become a problem [4]. Upon identification, the driver is informed by means of an indicator light on the instrument panel prompting the owner to seek service. OBD II also stores codes corresponding to the faults found in the emissions system; these codes can be accessed through a diagnostic port via a standard protocol. Currently the standard protocol used by OBD II is SAE J1850 [5].

The SAE J1850 standard is a Class B protocol, which is a medium speed network and is used for general information communications. There are two forms of J1850. One is a high-speed version that operates at 41.6 kb/s using a two-wire differential approach. The alternative is a single-wire approach that operates at 10.4 kb/s.

Collision resolution allows multiple nodes to transmit at the same time without causing a failure. Similar to CAN, messages in the J1850 protocol are prioritized. This is accomplished by allowing an active symbol to win bus arbitration over a passive symbol. When a node transmits a passive symbol and in turn receives an active symbol, the transmitting node realizes that another node with higher priority is transmitting at the same time. Thus, the passive node loses arbitration. This process is illustrated in Figure 2.2.

1.2.1.2 Local Interconnect Network (LIN)

The LIN specification [6] is a single wire communication protocol derived from the ISO-9141 standard for automotive diagnostics. This protocol is based on the common SCI (UART) interface, which is available as low cost silicon modules on almost all microcontrollers. This leads to very-cost effective solutions.

Unlike the J1850 protocol, access in a LIN network is governed by a master node and consequently eliminates the need for collision management in slave nodes. Furthermore, the master-slave configuration also leaves the master node with the responsibility of initiating communication in the network. As a result, multiple nodes will not transmit simultaneously and thus alleviates the need for bus arbitration. The master sends out a message via the single wire bus and exactly one slave node will be activated upon the reception and ensuing filtering of this message. Although the master controls

communication, data can be sent from slave to slave by a corresponding message ID triggered by the master.

The development of the LIN standard was in response to the lack of a low-end, low-cost multiplexed network that would complement and not replace existing networks. With maximum transmission speeds of 20 kb/s, LIN is targeted for applications involving on-off devices, such as car seats, doors, and mirrors. LIN can also connect higher speed networks, like CAN, together.

1.2.1.3 Media-Oriented Systems Transport (MOST)

OBD and LIN are characterized by low speed data transfer rates utilized for simple applications. The motivation for the MOST specification [7] was based on the requirement for a low-cost, high-speed multimedia network for the proliferation of in-vehicle applications, such as navigation units and video players.

In contrast to OBD and LIN, which use copper medium, MOST technology uses a fiber optic transmission medium. The resulting benefits with optical medium are increased data rates and robustness. MOST can achieve data rates up to 24.8 Mb/s and can support up to 64 devices, depending on the bandwidth requirements of each device.

Communication in a MOST system is based on peer-to-peer communication in combination with broadcasting abilities. MOST is a synchronous network where a timing master supplies the clock and all other devices synchronize their operation to this clock. The technology is similar to the public switched telephone network. There are both data and control channels defined. The control channels are used to set up the data channels the sender and receiver are to use. Once the connection is established, data can flow

continuously with no further processing of packet information required. This is the optimum mechanism for delivering streaming data, such as video.

1.2.1.4 Controller Area Network (CAN)

The CAN protocol, defined by Bosch [8], is based on the identification of transmitted messages via a message identifier. Based on a received message identifier, all nodes check to see if the message is relevant. Therefore, messages can be accepted by one, many, or all nodes. The identifier of a message also determines the priority with regard to bus access. In other words, CAN provides schemes to provide priority-based message access to the bus.

Bus access in a CAN network is not controlled by a master but instead every node can transmit as soon as the bus is idle. In such a competitive use scheme, it is possible that several nodes will try to occupy the bus concurrently. In other protocols, such a situation would lead to destroyed messages. The CAN protocol has established an access method that guarantees that the message with the highest priority wins access and then successive messages follow without destruction.

Perhaps the two most appealing aspects of the CAN protocol are the data rate of up to 1 Mb/s and the robustness of the CAN network. There are two data lines defined by CAN, which transmit the same data. If one line is damaged or grounded, the network is not hindered but operates under degraded noise immunity. The CAN protocol also has the ability to limit bus access for defective nodes in the network. These metrics make CAN useful for control of real-time critical functions. The CAN protocol will be discussed extensively in Chapter 2.

1.3 Potential Law Enforcement Applications

In-vehicle networks are obviously not a new idea; however, they have not been fully exploited. One opportunity that has not been fully examined is the use of in-vehicle networks on platforms other than a standard passenger car. As already suggested, a law enforcement platform presents a suite of technology over and above the ones found in civilian vehicles. This suite of technology does not replace the factory-installed devices inside vehicles but instead builds upon them. In turn, this package of technology adds wiring in excess of standard vehicle wiring. The problems associated with this added vehicle wiring include reduced layout space, deterioration of serviceability and lower fuel efficiency, other performance issues, manufacturing difficulties, and the presence of new connectors, which leads to unreliable operation [9].

The target vehicle environment discussed in this thesis, the New Jersey State Police (NJSP) troop car, hosts a variety of technologies, which include a radar system, in-vehicle video system, two-way radio, and a mobile data computer (MDC) system. The human interface points for these applications are all located in the cockpit of the vehicle creating problems finding sufficient space for the system control points. The only practical area of the car that offers sufficient space for secure mounting of these devices is the trunk of the vehicle. Because of the separation between the interfaces in the cockpit and the corresponding processors and control units in the trunk, there is an excessive amount of wiring involved with the installation of this technology suite.

One example is the pursuit light package used by the NJSP, illustrated in Figure 1.1 and 1.2. This application includes a combination of LED, strobe, and halogen lights that are controlled by a switching unit inside the cockpit of the car with a remote power unit in the trunk of the vehicle. Overall, there are 14 lights in different positions. In the

front of the car, there is one strobe light in each headlight as well as two LEDs in the grille. There are also LED lights in the two side view mirrors. In the rear of the vehicle, there are two strobe lights and two halogen lights in the interior positioned on the rear

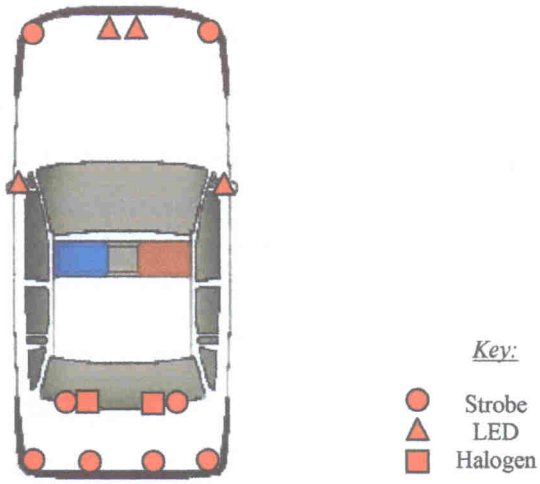

Figure 1.1 – Diagram of light placement on the NJSP troop car.

deck, and four strobe lights in the brake and reverse lights. As indicated, the interfaces in the cockpit of the car are separated from the control units in the trunk. For the light system, there is a dedicated cable that connects each light to the control unit in the trunk. The control unit then communicates with the cockpit user interface via another cable. The combination of these dedicated wire runs is an example of excessive wiring discussed earlier.

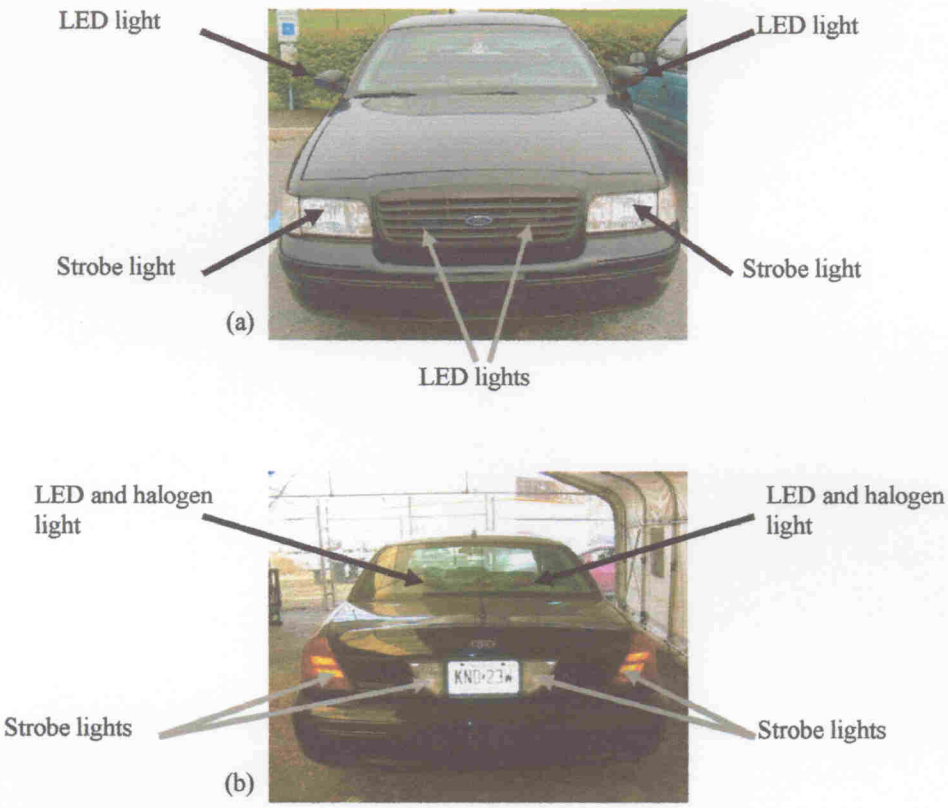

Figure 1.2 – Individual light locations of the New Jersey State Police Light package in the (a) front and (b) rear of the vehicle.

1.4 Selecting an In-Vehicle Network

The environment of a typical automobile presents multiple hazards for electronics. Electronics are subjected to vibration, extreme temperature swings, strong electromagnetic fields, voltage spikes, and many other potential risks. With this in mind, there are a number of special design considerations when designing an in-vehicle network [9,10]. Non-safety related communication networks should meet most or all of the considerations detailed in Table 1.2. Additional specifications are required for safety critical communications. Because CAN was initially designed for use in automobiles, the need for a fast, robust protocol was critical to prove useful in the automotive

environment. CAN nodes have the ability to detect errors as well as to confine errors based on their severity. Fault confinement essentially guarantees bandwidth for critical system information by limiting bus access by faulty nodes. The strengths of the Controller Area Network protocol made it the ideal candidate for automotive networks.

Table 1.2 – Vehicle network design considerations [9].

Design Consideration	Description
High integrity	Probability of undetected error must be insignificant. For CAN the probability of error is < message error rate · $4.7 \cdot 10^{-10}$.
Bounded determinism	Messages must be delivered with minimal latencies.
EMI compliance	Network must be able to tolerate EMI and limit the amount of interference radiated.
Low interconnection count	Connections should be limited to lower probability of faults.
Compact connectors	Smaller connectors lead to smaller nodes.
Low cost	The ability to keep the cost of a node to a minimum allows for greater acceptance and mass production.
Network composability	Systems can be developed independently and integrated flawlessly with a current network.
Fault tolerance	If a fault occurs, communications must be able to continue.

1.4.1 Selecting the Appropriate Network for the Application

The communication networks discussed to this point have been introduced for three main reasons [10]:

1.) To integrate subsystems with the goal of creating a more complex, overall system.

environment. CAN nodes have the ability to detect errors as well as to confine errors based on their severity. Fault confinement essentially guarantees bandwidth for critical system information by limiting bus access by faulty nodes. The strengths of the Controller Area Network protocol made it the ideal candidate for automotive networks.

Table 1.2 – Vehicle network design considerations [9].

Design Consideration	Description
High integrity	Probability of undetected error must be insignificant. For CAN the probability of error is < message error rate · $4.7 \cdot 10^{-10}$.
Bounded determinism	Messages must be delivered with minimal latencies.
EMI compliance	Network must be able to tolerate EMI and limit the amount of interference radiated.
Low interconnection count	Connections should be limited to lower probability of faults.
Compact connectors	Smaller connectors lead to smaller nodes.
Low cost	The ability to keep the cost of a node to a minimum allows for greater acceptance and mass production.
Network composability	Systems can be developed independently and integrated flawlessly with a current network.
Fault tolerance	If a fault occurs, communications must be able to continue.

1.4.1 Selecting the Appropriate Network for the Application

The communication networks discussed to this point have been introduced for three main reasons [10]:

1.) To integrate subsystems with the goal of creating a more complex, overall system.

Alternatively, low bandwidth networks are commonly used for simple comfort control applications, such as power windows and adjustable seats, where delays on the order of a second are acceptable. These comfort systems are highly cost sensitive and the associated networks hosting these applications are constructed to be cost efficient as well.

The target application discussed in this thesis, the NJSP pursuit light package, does not require a high bandwidth network since it is not a real-time, critical control application such as powertrain control nor is it a multimedia application, which both demand reliable and responsive networks. Although, the pursuit light package could operate on a low bandwidth network such as a single wire LIN implementation, greater flexibility and added functionality can be achieved with a network such as CAN. The advantages of CAN over J1850 (OBD), LIN and other low bandwidth networks is the fact that CAN offers the same functionality but at a higher bit rate. Despite the fact that CAN may be more expensive than these low bandwidth networks, the CAN protocol has the flexibility to operate as a class A or class B network much like OBD and LIN as well as a class C network.

Availability of a network implementation also plays a vital role in the selection of a network. An emerging network such as MOST does not have the market popularity of an established, standardized network such as CAN. With support from many of the leading semiconductor companies the availability of CAN implementations are numerous. CAN is currently the most widely used vehicular network, with more than 100 million CAN nodes sold in 2000 alone [11].

Although CAN is not the cheapest network it is also not the most expensive network to implement. CAN stand alone controllers cost around $3 and CAN controllers

combined with a microcontroller can cost around $7 [11]. The economics of CAN combined with its availability; flexibility and robust communications profile make it a wise choice for an in-vehicle network at this time for the New Jersey State Police Troopcar. However, in-car electronics will continue to progress towards more multimedia applications and other complex, distributed electronic controls and the need for a better and faster network may present itself in the future. In such an event, other networks can be implemented for dissimilar functions. For instance, a MOST network can be established for the control of multimedia applications while the existing CAN network handles such applications as the comfort electronics, engine management, and other control applications. The use of bridges and gateway devices will support the interconnections between multi-network environments.

1.5 Current CAN Implementations

Since the beginning of the 1980s, many automotive manufacturers have become actively involved in research and development of Class C and Class A networks. The concerted efforts were focused on the communication between real-time, critical controls and comfort and convenience controls. Many different protocols were developed during this time, but it was CAN that became the predominant protocol for automotive applications. In 1992, CAN was first applied in Mercedes S-class cars for communication between engine controller, gear box controller, dashboard, and distributed air conditioning control [12]. Many automotive manufacturers followed the lead of Mercedes a short time later.

Today CAN has been designed into and applied to many areas related to automotive and industrial controls, such as airplanes, trains, passenger cars, trucks, buses,

textile machinery, medical systems, and robots to name a few [13-19]. While the use of CAN in the automotive industry is already established, CAN is likely poised to make another major impact on the automotive market with the growing interest in electric and hybrid vehicles [20]. CAN is an ideal network for communications in an electric vehicle. However, with the creation of electric vehicles and drive-by-wire applications, there are demands for CAN updates designed for safety-critical, real-time automotive systems [21-22].

1.6 Statement of the Problem

A modern law enforcement vehicle contains numerous systems and electronic devices in addition to stock vehicle electronics. Many law enforcement agencies utilize mobile PCs and other computing devices to retrieve driving records, arrest records, and other sources of information to aid the officer. However, these computers usually operate independently from the networks inside the vehicle, thereby missing an opportunity to increase the functionality of the vehicle. The problem is further compounded by the fact that different law enforcement agencies use a variety of vehicle makes and model years, as well as a variety of computing devices. The continued development of standardized in-vehicle networks will facilitate future CAN enabled devices to simply plug and play into an existing CAN network within an automobile equipped with a CAN bus. Furthermore, simply replacing point-to-point wiring with a robust serial network could reduce installation costs, minimize wiring weight, and still provide greater functionality.

The objective of the research work presented in this thesis is to develop a CAN network within a model law enforcement vehicle that allows multiple applications, such as the pursuit light package, to interface with the network, while adding functionality and

reducing wiring. The selected network must demonstrate the ability to decrease the amount of point-to-point wiring while maintaining or improving the established integrity of the application, and simultaneously offer increased functionality to the entire vehicle environment.

1.7 Scope and Organization of Thesis

This thesis presents a novel approach for implementing an in-vehicle serial network for controlling law enforcement applications. The results of the research work presented show a reduction of point-to-point wiring in the NJSP troop car, while adding increased functionality to the entire environment and demonstrating the advantages that can be gained from the inclusion of reconfigurable logic elements. The proposed network has been evaluated with the use of field programmable gate arrays (FPGA) to demonstrate the ability to control an application such as the NJSP pursuit light package.

This thesis is organized as follows: Chapter 1 (this introduction) is followed by background material on data communications and the Controller Area Network in Chapter 2. The approach proposed for implementing CAN as the in-vehicle network is presented in Chapter 3. Results from the implementation are then given in Chapter 4. Chapter 5 follows with a summary of accomplishments and conclusions, along with a discussion for future directions related to this research.

Chapter 2: Background

While there are a large variety of protocols suited for automotive implementation, in general, the protocols share many fundamental principles. The following discussion highlights the fundamentals of data communications that provide the basis for understanding the Controller Area Network.

In its simplest form, data communication takes place between two devices that are directly connected by a point-to-point medium [23]. It is often impractical, as is the case in an automobile, for many devices to be directly connected by point-to-point wiring. An alternative topology is a bus-organized system as the solution for connecting a set of devices using a shared communications network. This concept is illustrated in Figure 2.1.

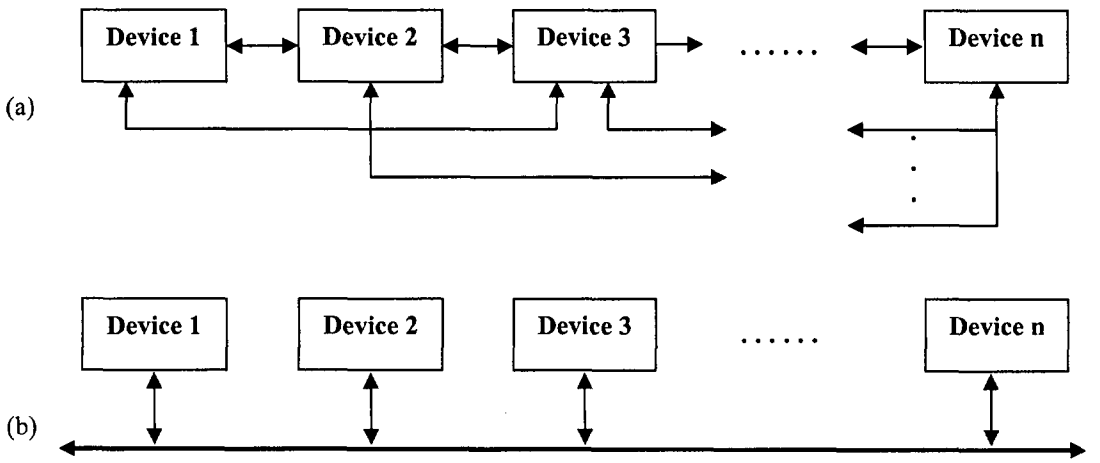

Figure 2.1 – Example of data transfer using (a) point-to-point transmission medium and device interconnections using a (b) bus-organized communications network.

2.1 Data Communications Reference Model

There are two key concepts that need to be realized for devices in a network to communicate between each other properly: protocol and protocol architecture. In general, a device or node in a network has the capability of both sending and receiving information. Therefore, it follows that for two such devices to communicate properly between each other they need to "speak the same language." To ensure this, there are conventions that must be met by all devices. The conventions, known as a *protocol*, encompass three key elements [23]:

1) Syntax
2) Semantics
3) Timing

These three elements govern such things as data format, error handling, sequencing, control of the appropriate data to be transmitted, the correct form for data transmission, and transmission timing.

Most network applications follow a layered approach to system implementation. The primary reference for most protocol specifications is the open systems interconnection (OSI) model developed by the International Standards Organization (ISO). According to the OSI model, data communication systems can be described as a seven-layer hierarchical model. The seven layers are:

1) Application
2) Presentation
3) Session
4) Transport
5) Network

6) Data Link

7) Physical

Figure 2.2 illustrates the seven layers of the OSI model and provides a brief definition of the function performed at each layer.

Application Provides access to the OSI environment for users and provides distributed information services.
Presentation Provides independence to the application processes from differences in data representation.
Session Provides the control structure for communication between applications; establishes, manages, and terminates connections between cooperating applications.
Transport Provides reliable, transparent transfer of data between end points; provides end-to-end error recovery and flow control.
Network Provides upper layers with independence from the data transmission and switching technologies used to connect systems; responsible for establishing, maintaining, and terminating connections.
Data Link Provides for the reliable transfer of information across the physical link; sends blocks of data with the necessary synchronization, error control, and flow control.
Physical Concerned with transmission of the bit stream over physical medium; deals with the mechanical, electrical, functional, and procedural characteristics to access the physical medium.

Figure 2.2 – The seven OSI layers [23].

However, since the model was intended to provide a framework for any type of communication system, some systems do not use all the layers. In many cases, only three layers (physical layer, data link layer, and application layer) are relevant. The CAN protocol implements most of the lower two layers of the OSI model.

2.2 CAN Architecture

To achieve design transparency and implementation compatibility, the CAN specification [8] subdivides CAN into layers according to the ISO/OSI reference model:

1) Data Link

2) Physical

3) Application

The following sections outline the basic concepts of these three layers and further discuss the overall architecture of CAN [8, 24-25].

2.2.1 CAN Data Link Layer

In general, the data link layer has two main functions. The first function is construction of data frames, which also control error detection code as well as containing data. The transmitted control information enables the receiver to determine if a frame is relevant and whether or not there has been an error in transmission. If a frame has been corrupted then the frames are rejected and retransmitted.

The second main function controls bus arbitration to resolve conflicts when multiple nodes access the bus simultaneously. The following sections describe the principle of bus arbitration, the format of a CAN frame, error detection, and error confinement.

2.2.1.1 Bus Arbitration

The CAN protocol uses a collision avoidance scheme, carrier-sense multiple access with collision avoidance (CSMA/CA). Although several nodes may start

2.) To manage the increasing number of electronic components.

3.) To reduce the costs associated with increased wiring of electronic components.

To reach these goals numerous networks were introduced with the same driving motivation but inevitably led to networks with varying characteristics, thus reinforcing the idea that there is no single network protocol that is available to connect the variety of systems within an automobile.

Since there is no common protocol the selection of a protocol is dependent upon the specific application, availability of parts, and economics, as well as meeting the requirements discussed in the previous section. Table 1.3 demonstrates the selection of networks discussed in the previous sections. Among these networks there is a range of functional and economic advantages and disadvantages. Networks of high bandwidth are used for vehicle multimedia applications, where cost is less important.

Table 1.3 – Selection of automotive networks.

Protocol	Application	Media	Media Access	Error Detection	Data Field Length	Max. Bit Rate
CAN	Control	2 wire	CSMA/CA	CRC	0-64 bit	1 Mb/s
J1850 PWM	Control	2 wire	CSMA/CR	CRC	8-64 bit	41.6 kb/s
J1850 VPW	Control	1 wire	CSMA/CR	CRC	8-64 bit	10.4 kb/s
LIN	Control	1 wire	Master/slave	CRC	0-64 bit	20 kb/s
MOST	Multimedia	Fiber optic	Master/slave	CRC	0-480 bit	25 Mb/s

This results in the node with the lower priority retiring from bus arbitration and becoming a receiver of the transmitted message. Thus, the node with highest priority gains access to the bus. Once the bus becomes free, the subordinate node(s) engage arbitration again for the completion of their remaining message(s).

For comparison purposes, the popular protocol Ethernet [26] follows the CSMA/CD access method. When an Ethernet node wants to begin transmission, it monitors the channel to see if any other node is currently transmitting. The channel is considered free when after 96 bit times there are no messages occupying that channel. However, if the channel is busy when the node wishes to transmit, the node will continue monitoring the channel until it is free. While the node is transmitting, the node continues to monitor the channel for collisions. If a collision occurs during transmission, the transmission is immediately concluded and a jam signal is sent over the channel. The jam signal ensures that all other nodes will realize that there has been a collision. Following collision, the transmitting node endures a waiting time before attempting to re-transmit from the beginning.

2.2.1.2 Frame Formats

There are four types of frames specified by the CAN protocol:

1) Data frame

2) Remote frame

3) Error frame

4) Overload frame

The CAN specification details two different, compatible frame formats. There is a "base format" with an 11-bit frame identifier, and an "extended format" with a 29-bit frame identifier.

2.2.1.2.1 Data Frame

A CAN data frame (Figure 2.4) is composed of seven different bit fields: start-of-frame (SOF), arbitration, control, data, CRC, acknowledgement, and end-of-frame (EOF). It is important to note that the data field can be of length zero.

The SOF bit is represented by a single dominant bit and marks the beginning of data frames and remote frames. A node in a CAN network is only allowed to begin arbitration if the bus is idle or free. All nodes then synchronize to the leading edge of the SOF created by the first node to start bus arbitration.

The arbitration field consists of the identifier field and the remote transmission request bit (RTR), which is transmitted as a dominant (0) bit in a data frame. A frame is identified and its priority defined by the frame identifier. The length of the frame identifier in the base format is 11 bits, which means that 2048 (2^{11}) different messages are discernible. The extended format can distinguish 512 million (2^{29}) messages. These bits are transmitted in the order ID [10:0], with the most significant bit being ID10.

The CAN specification defines the control field as consisting of six bits. The first bit, identifier extension bit (IDE), distinguishes between base and extended formats. For the base format, the IDE bit is transmitted as a dominant level. The second bit, r0, is reserved for future expansion of the CAN protocol and until defined, is transmitted dominantly. The last four bits in the control field, referred to as the data length code,

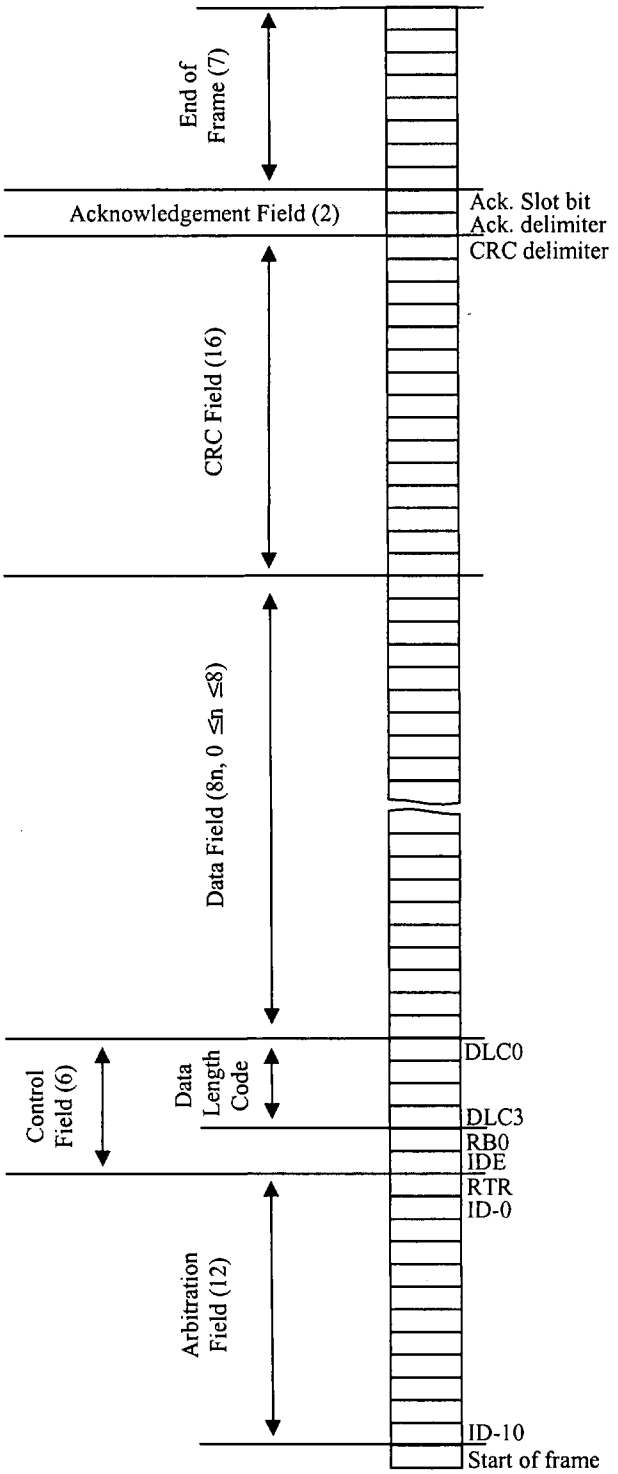

Figure 2.4 – Standard (base format) CAN data frame (number of bits = 44 + 8n), from [27].

determine the number of bytes transmitted in the succeeding data field. For example, a value of 0 corresponds to 0 data bytes and a value of 8 corresponds to 8 data bytes. The straight binary coding of the number of data bytes by the data length code is illustrated in Table 2.1.

Table 2.1 – Coding of the number of data bytes by the data length code from [8].

Number of Data Bytes	Data Length Code			
	DLC3	DLC2	DLC1	DLC0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0

Upon the completed transmission of the data field, the cyclic redundancy check (CRC) field enables a receiving node to verify the integrity of the received data. CRC is one of the most common and powerful, error-detecting codes. In general, CRC is described as follows [23] and summarized in Fig. 2.5. Given a k-bit block of bits or messages, a transmitter generates an n-bit sequence, known as a frame check sequence (FCS). The resulting frame, which is $k + n$ bits long, is exactly divisible by some

CRC polynomial process :

Define:

$T(X) = (k+n)-$bit frame

$M(X) = k-$bit message

$P(X) = (n+1)-$bit divisor

$F(X) = n$-bit FCS

T is the concatenation of M and F, where M is shifted to the left n bits.

$$T(X) = X^n M(X) + F(X)$$

T should be exactly divisible by P. Divide $X^n M(X)$ by $P(X)$.

$$\frac{X^n M(X)}{P(X)} = Q(X) + \frac{R(X)}{P(X)}$$

There is a quotient and a remainder, the remainder becomes the FCS.

$$T(X) = X^n M(X) + R(X)$$

Now divide T by P.

$$\frac{T(X)}{P(X)} = \frac{X^n M(X) + R(X)}{P(X)}$$

$$\frac{T(X)}{P(X)} = Q(X) + \frac{R(X)}{P(X)} + \frac{R(X)}{P(X)}$$

However, since arithmetic operations are modulo-2:

$$R(X) + R(X) = 0$$

Therefore,

$$\frac{T(X)}{P(X)} = Q(X)$$

There is no remainder and therefore, T is exactly divisible by P.

Figure 2.5 – CRC polynomial procedure from [23].

predetermined number. After receiving the message, a receiver will then divide the received frame by the predetermined number. If there is no remainder after the division, the receiver will conclude that there was no error.

The CRC procedure can be implemented in three different ways: modulo-2 arithmetic, polynomials and random logic. When the CRC procedure is represented as a polynomial, all values of the polynomial are expressed as a dummy variable X with binary coefficients. The coefficients correspond to the bits in the binary message. Thus, for a binary message $M=110011$, the polynomial form is $M(X)= X^5+X^4+X^1+1$. Figure 2.5 demonstrates the CRC process expressed as polynomials.

According to the CAN protocol the CRC field consists of a 15-bit check sequence plus a delimiter bit. The FCS is derived from a cyclic redundancy code, based upon the Bose-Chaudhuri-Hocquenghem (BCH) code [28], which is best suited for frames that contain less than 127 bits. To perform the CRC calculation, the frame header and the data to be transmitted are represented as the frame polynomial, which is then divided by a defined generator polynomial. The remainder of the modulo-2 division is the CRC sequence sent over the bus as part of the frame. A receiving node in the network receives the frame polynomial, which is also divided by the same generator polynomial. When the frame is transmitted error-free, the calculated check sequence of the receiver is the same as the sequence contained in the received frame.

The coefficients of the frame polynomial are formed by every bit value from the SOF to the final data bit, extended by 15 least significant coefficients of value 0. This polynomial is then divided by the generator polynomial,

$$x^{15} + x^{14} + x^{10} + x^8 + x^7 + x^4 + x^3 + 1.$$

This division can be implemented by means of a random logic circuit, more specifically a 15-bit linear feedback shift register [8].

The CRC delimiter bit is followed by the 2-bit acknowledgement field, which consists of an acknowledgement slot and an acknowledgement delimiter. A transmitting node transmits the two acknowledgement bits high and then awaits reception acknowledgement of the transmitted frame by at least one receiver. When a validation is received, the receiving node acknowledges the transmitter by writing zeros in the acknowledgement slot.

The completion of the data frame is signaled by a sequence of seven recessive bits, referred to as the end-of-frame flag (EOF), combined with the recessive acknowledgement delimiter bit. Together, these eight bits indicate an error free transmitted frame.

2.2.1.2.2 Remote Frame

Any node can initiate transmission of a specific frame from another node by means of the remote frame. The frame requested by a node is specified by the identifier transmitted with the remote frame. The difference between the remote frame and the data frame formats is the RTR bit level. RTR is transmitted at logic high for remote frame; for the data frame, RTR is low. The difference in transmission levels ensures that when a remote frame arbitrates simultaneously with a data frame with the same identifier, the data frame wins arbitration. Another difference between a data frame and a remote frame is the fact that the data field of a remote frame is always empty. Instead of the data length code determining the number of data bytes in the data field of the remote frame, it corresponds to the data length code of the requested frame.

2.2.1.2.3 Error Frame

In CAN, any sequence of more than five consecutive bits of equal value within a frame is considered an error frame. An error frame consists of two fields, error flag, and error delimiter. The error frame indicates the detection of any error during transmission of a data frame or a remote frame.

The first field contains the superposition of error flags transmitted by one or several nodes. Error flags can take on two different forms: active or passive error flags. The active error flag consists of six consecutive logic 0 bits, whereas the passive error flag consists of six consecutive logic 1 bits. An active error node detecting an error condition signals the anomaly with an active flag. The flag's form violates the bit-stuffing rule and consequently all other nodes detect an error condition and begin sending their own error flag. Therefore, the sequence of logic 0 bits monitored on the bus results from the superposition of the different error flags from various nodes. This sequence ranges from a minimum of six bits to a maximum of 12.

A passive error flag initiated by a transmitting node will result in errors at receiving nodes when the flag starts within a bit stuffed frame. The result is a bit stuff error. However, if the flag starts during arbitration or less than 6 bits before the end of the CRC sequence that ends in recessive bits, an error will not be recognized by the receiver. A receiver attempting to send a passive error flag cannot overcome any current activity on the bus. As a result, error passive receivers have to wait for six equal bits after detecting an error condition to complete their error flag.

After transmission of an error flag, each node sends recessive bits and monitors the bus level until it detects a recessive bit. Once a recessive bit is detected, the node then

starts transmitting seven more recessive bits. This error delimiter field determines if the node was the first to detect an error.

2.2.1.2.4 Overload Frame

An overload frame can be considered as a special form of an error frame. The overload frame is composed of an overload flag field and an overload delimiter. The overload frame is used to request a delay of the next data or remote frame by the receiver or to signal certain error conditions related to the intermission field. The intermission field is a 3-bit field of recessive bits, which represents a minimum distance of three bits between frames.

2.2.1.3 Error Detection and Handling

The overall high requirements for security and integrity of data transmission call for an average of less than one error for every thousand messages (10^{+3}). To achieve this CAN has a variety of error detection mechanisms built in [29] including:

1) Bit check (monitoring)
2) Frame check
3) CRC check
4) Acknowledgement check
5) Bit stuff rule check

If any one of these mechanisms is violated, then an error flag is created.

During the monitoring phase, every node that transmits a signal monitors the corresponding bus level. If the bus level does not match the transmitted level, a bit error

is detected. Bit checking thus proves to be very effective in detecting global errors as well as local errors at the transmitting node.

All frames in the CAN protocol contain fixed bit fields, SOF, RTR, IDE, DLC, EOF, and delimiters. These fixed fields must be consistent with the protocol; a non-compliant format field generates a format error and the receiver rejects the frame.

CAN implements the principle of cyclic redundancy checking (CRC) (see section 2.2.1.2.2) to enable a receiving node to verify the integrity of a received frame. If the CRC sequence received from the transmitting node is not identical to the CRC sequence calculated by the receiving node, then there is a resulting CRC error.

After completing transmission, a transmitting node expects that at least one receiver will acknowledge reception of a valid frame. However, if no receiving nodes acknowledge by forcing logic 0 in the acknowledgement slot, the transmitting node produces an acknowledgement error.

The final error handling mechanism reinforces detected errors. Every node that detects the violation of the bit-stuffing rule or simply detects an error flag automatically produces an error frame. The resulting error frame generated for any of the above errors will be sent to all nodes; upon reception all nodes will discard the received bits of the corrupted message. Once informed of the error, the transmitter of the corrupted message will retransmit the original message when the bus becomes free.

Although these mechanisms provide for a high degree of error detection, there are still possibilities for some errors to go undetected. The probability for errors to be undetectable at receiving nodes according to the CAN specification is less than $4.7 \cdot 10^{-14}$. This means that on average, about one corrupted message is expected to go undetected

for every 20 trillion transmitted messages. However, further analysis shows that the residual error for a 10-node network reaches a maximum value of $7.2 \cdot 10^{-12}$ [29].

2.2.2 CAN Physical Layer

The physical layer of the CAN protocol defines how the frames or signals defined by the data link layer are actually transmitted. The physical layer defines bit representation, bit timing, bit coding and decoding, and the specification of the physical medium. The common CAN physical layer concept is specified by the ISO 11898 standard, parts 1 and 2 [30-31] as well as the CAN specification [8].

2.2.2.1 Bit Timing

The bit time, t_b, in CAN is the time duration of one bit, which is the reciprocal of the nominal bit rate. This bit time is partitioned into four segments (Fig. 2.6):

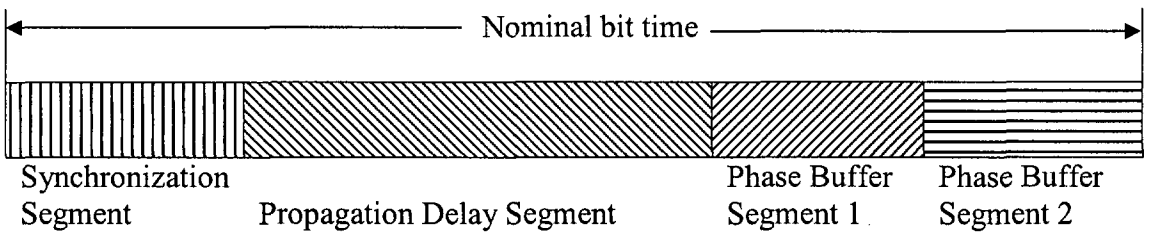

Figure 2.6 – CAN bit timing [32].

synchronization segment, propagation delay segment, phase buffer segment 1, and phase buffer segment 2. Each segment consists of a specific amount of time quantum (t_q). Time quantum is defined as the product of the value of the baud rate prescaler (BRP), which can include integer values from 1 to 32, and the reciprocal of the system clock (f_{sys}):

$$t_q = \frac{BRP}{f_{sys}} \tag{2.1}$$

The synchronization segment is the component of the bit time where signal edges occur and is used to synchronize nodes on the CAN bus. Physical delay times within the network are resolved by the propagation delay segment. Since a CAN node is required to receive data from another node during its own transmission, any node synchronized to the bus will be out of phase with the transmitting node of the bit stream seen on the bus. Therefore, the propagation delay segment must be twice as long as the sum of the maximum signal propagation delays between two nodes.

Phase buffer segment 1 and 2 are used to compensate for edge phase errors, defined to be the distance between an edge occurring outside the synchronization segment and the synchronization segment. The buffer segments are separated by the sample point, which is the time that the bus level is read and interpreted as a bit value. Depending on the edge phase error, the phase buffer segments can be altered to effectively change the sample point and thus restore the sampling point relative to the current bit position. For example, if the transmitter is operating at a lower clock speed than the receiver, the signal edge will arrive after the synchronization segment. The receiver compensates by shifting the sampling point. This is done by extending the phase buffer segment 1 by an amount equal to the error, as illustrated in Figure 2.7. The same approach applies to the condition where the transmitter is faster than the receiver.

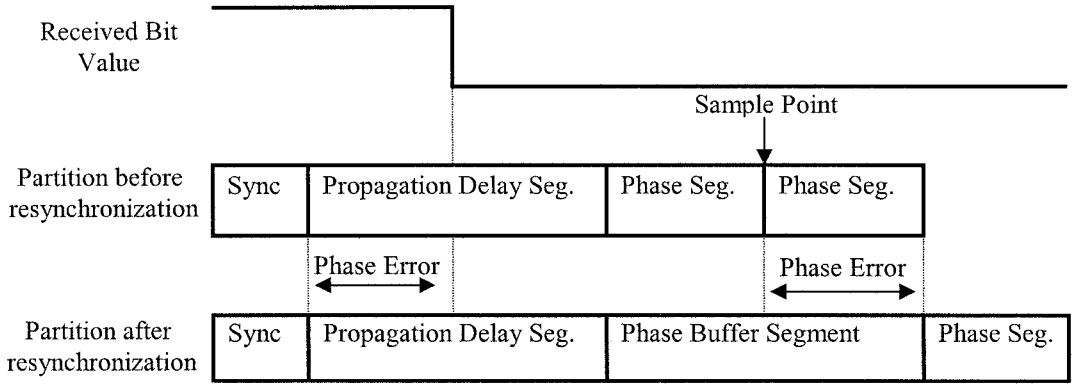

Figure 2.7 – Principle of resynchronization on a late edge.

The alterations made to the phase buffer segments are only temporary. At the next bit time, the segments will return to their original values as long as there are no phase errors detected in that bit time.

2.2.2.2 Physical Signaling

In the CAN protocol, the bits of a frame are physically represented according to the Non-Return-to-Zero (NRZ) code (Figure 2.8). Since a transmission medium normally cannot accept transmitted symbols in their "natural" form; symbols are transmitted using

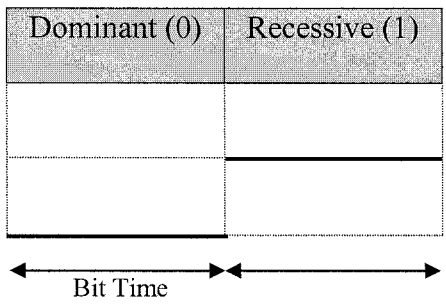

Figure 2.8 – NRZ bit representation.

line codes instead. Symbols in CAN are transmitted using the baseband line coding technique NRZ [33]. NRZ defines two bit levels, dominant (0) and recessive (1). The generated bit level remains constant over the entire bit time interval. This can result in a relatively long stream of data bits of equal value and thus there may not be any edges available for synchronization. Furthermore, recall that more than five consecutive bits of the same polarity (111111 or 000000) is used to represent an error. To eliminate synchronization issues and error frame conflicts, CAN implements a bit stream coding method known as bit stuffing.

In general, a transmitter implementing bit stuffing inserts an additional bit (stuff bit) of complementary value after a prescribed amount of consecutive bits (stuff width) of the same level (Figure 2.9). On the receiver side, the procedure is reversed and the inserted bits are removed. CAN uses a stuff width of five bits; this ensures that a sequence of six bits of equal polarity is a violation of the bit stuffing rule and will be

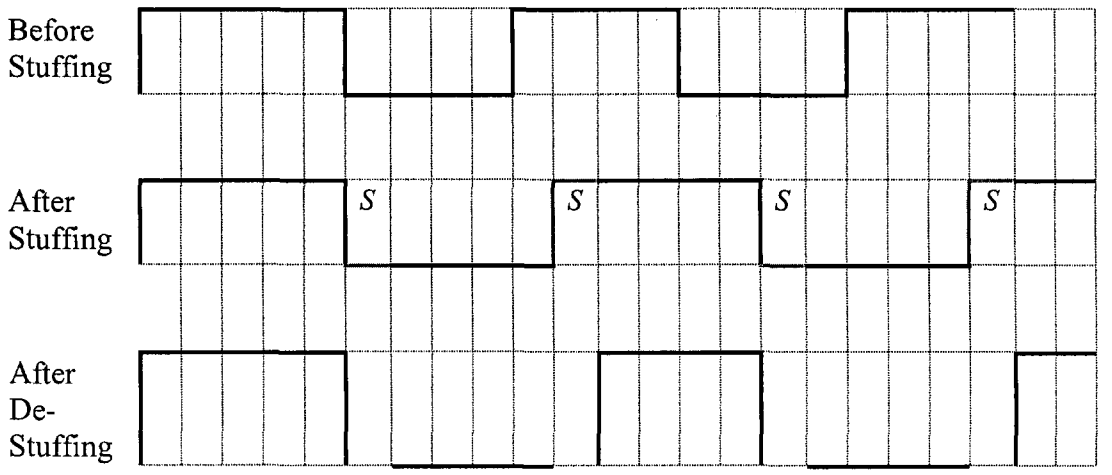

Figure 2.9 – Principle of bit stuffing with a stuff width of 5 bits, where *S* is the stuff bit.

interpreted as an error frame. Bit stuffing allows receiving nodes to synchronize on recessive to dominant transitions.

Bit stuffing can increase the actual number of transmitted bits beyond the size of the original message, which can result in a transmission delay. Before bit stuffing, the total number of bits in a data frame including a three-bit intermission field is:

$$N_{bits} = 8n + 47, \tag{2.2}$$

where n is the number of data bytes in the data field. The CRC delimiter, acknowledgement field, and end of frame are not coded using the bit stuff method, which leaves 34 of the 47 control bits subject to bit stuffing. In a worst-case scenario, the total number of bits after bit stuffing cannot be greater than [34]:

$$N_{stuffed} = 8n + 47 + \left\lfloor \frac{34 + 8n - 1}{4} \right\rfloor. \tag{2.3}$$

This model can be used to determine how long it will take to send a frame. Let τ equal the amount of time it takes to transmit a bit over a bus operating at its full 1 Mb/s potential. For this case, τ = 1μs. Therefore, the worst-case transmission time, C_i, of a frame is:

$$C_i = N_{stuffed} \cdot \tau = \left(8n + 47 + \left\lfloor \frac{34 + 8n - 1}{4} \right\rfloor \right) \tau. \tag{2.4}$$

For a data field consisting of eight data bytes, n = *8,* then C_i = 135μs. In other words, for the largest frame it takes 135 bit times to complete the transmission.

2.2.2.3 Physical Transmission Medium

A two-wire differential bus (Figure 2.10) is a typical transmission medium used to implement CAN networks. The advantage of using the two-wire differential approach is

that it is robust to certain faults. If the wires are twisted, it is resistant to electromagnetic interference. Good transmission line practice means that the bus lines must also be terminated at each end by a matched resistor to minimize signal reflections. If the lines are not terminated properly, the signal lines may also become disturbed by induced radio frequency signals.

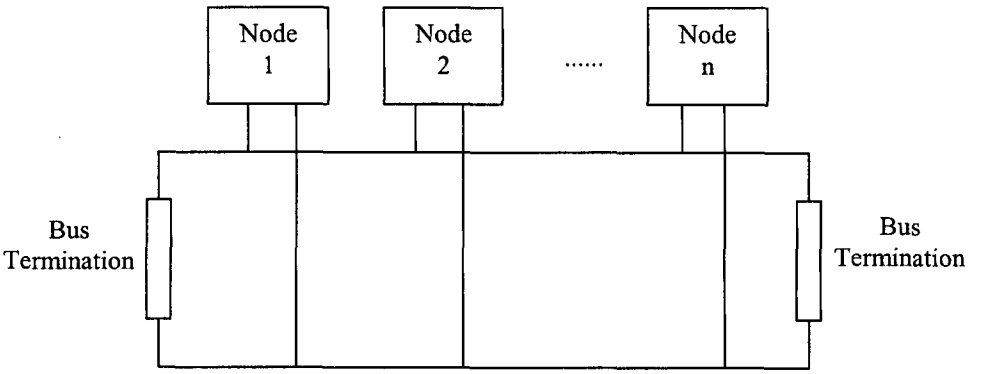

Figure 2.10 – Two-wire implementation of a CAN network.

With any CAN implementation, bus length becomes dependent on the characteristics of the transmission medium. An important parameter of an electric line is its characteristic impedance. The characteristic impedance Z_o is given by:

$$Z_o = \sqrt{\frac{L}{C}} \tag{2.5}$$

Where L and C represent the inductance and capacitance per unit length of the electric line. The typical line impedance of a two-wire line is about 120 Ω. To avoid signal reflections it follows that the two-wire line be terminated at each end with a resistor equivalent to the characteristic impedance, Z_L. The signal reflection coefficient Γ is given by:

$$\Gamma = \frac{Z_L - Z_o}{Z_L + Z_o}. \tag{2.6}$$

When the load impedance is equal to the characteristic impedance, there is no signal reflection.

Another parameter of the electrical medium that needs to be considered is the signal propagation time. The signal propagation velocity v_p is given by:

$$v_p = \frac{c}{\sqrt{\mu_r \cdot \varepsilon_r}}, \tag{2.7}$$

With c equal to the velocity of light ($3 \cdot 10^8$ m/s), μ_r denoting relative permeability, and ε_r denoting relative permittivity. It can also be shown that the propagation velocity can be described in terms of the electrical parameters of the line:

$$v_p = \frac{1}{\sqrt{LC}} \tag{2.8}$$

Now it can easily be seen that as the inductance and capacitance of the medium increase the propagation velocity decreases. As a result, to obtain low propagation times for a given bit rate, the required signal velocity must be large. However, the tradeoff is that the total propagation delay is dependent upon the bus length and the signal propagation velocity. Total propagation delay is expressed as:

$$t_{prop} = v_p \cdot l \tag{2.9}$$

Therefore it is essential to keep the overall bus length to a minimum to achieve low propagation delays.

Propagation times are not the only limiting factor for bus length. With longer bus lengths, the impact of voltage drop over the bus cannot be ignored. The maximum voltage drop over the bus must be such that the any receiving node in the network can

reliably interpret the signal level of the most remote sending node operating at a minimum signal level. To determine the maximum bus length, the distributed line resistances R_x in Figure 2.11 are calculated:

$$R_x < \frac{R_{itot} \parallel R_t \cdot (V_1 - V_n)}{2 \cdot V_n} \tag{2.10}$$

The model demonstrates node 1 supplying a voltage level (V_1) at one end of the bus line and reception of a voltage level (V_n) by a receiving node, n, at the opposite end of the bus. The termination resistors are labeled R_t and R_{itot} is the input resistance of the receiver.

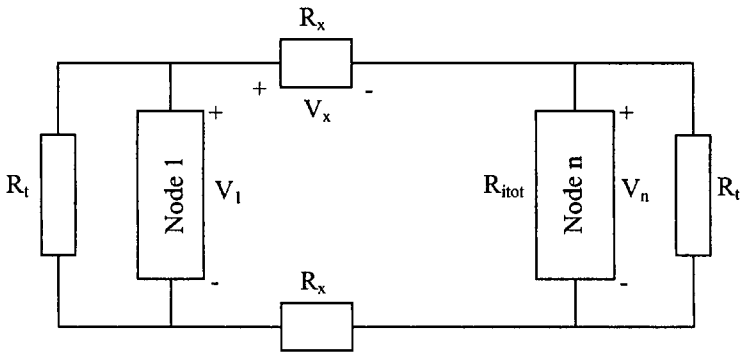

Figure 2.11 – Voltage conditions on a two-wire bus [25].

Once the maximum line resistance is known, the maximum bus length can be determined considering the conductor resistance, ρ, per unit length and the cross section of the wire.

$$L = \frac{R_x \cdot A}{\rho} \tag{2.11}$$

Typical wire gauges used in CAN are usually in the range of 18-20 AWG, which have a respective resistance per unit length in the range of about 6-10 ohms per 1000 feet.

For simplicity, keeping the length of the bus to a minimum will eliminate many issues relative to propagation times and voltage drop over the network.

2.2.3 CAN Application Layer

The CAN standard only specifies layers 1 and 2 of the OSI model. Higher layers, such as the application layer (layer 7), are typically implemented in software. Leaving this higher layer undefined allows application-specific design. Additional functionality not provided by the physical layer is usually required to conduct network management, node supervision functions, and other functions. This can be done using standardized higher layer protocols to bridge the gap between the application and the CAN standard.

Because of the widespread usage of CAN and the varying objectives of each implementation, several higher layer protocols have been standardized. The three predominant protocols [35-37] include:

1) CAN Application Layer (CAL)
2) CANopen
3) DeviceNet

The development of these higher layer protocols has led to interoperability of devices from varying manufacturers.

Full details of the specifications for the higher layer protocols are beyond the scope of this treatment; the interested reader is referred to pertinent references [35-37]. In brief, the functionality provided by all three protocols allows for improved communications. For example, CANopen enables the configuration of each node by a master node or by some external configuration tool. The parameters set by the configuration govern the communication behavior of a device. This allocates which data

is stuffed where, into which message or address, and when the message is triggered. Essentially, the higher layer protocols support the intelligence of a given application.

In this chapter, an overview of the CAN protocol has been presented. The protocol was described in terms of the ISO/OSI layers 1 and 2 and establishes the CAN message structure, bus arbitration process, error detection, and timing specifications. With an understanding of the protocol, the following chapter describes the approach for the HDL implementation of the protocol. Application of this work as an implementation for an in-vehicle network is also presented.

Chapter 3: Approach

Implementing the Controller Area Network specification can be handled in one of two ways. A CAN controller node can be constructed by integrating off-the-shelf components or by creating the node via a hardware description language. For the purposes of this thesis, the CAN node will be constructed using a hybrid of hardware description language (HDL) provided by OPENCORES.org [38] and off-the-shelf components. The provided HDL for the CAN controller is modeled after the Philips SJA1000 stand-alone CAN controller [39].

VHDL, an acronym for Very High Speed Integrated Circuit (VHSIC) Hardware Description Language, and Verilog are two common types of hardware description language that can be used to model a digital system from the algorithmic level to the gate level [40]. These languages allow a system to be described hierarchically; that is, the system can be modeled as an interconnection of components and in turn each component can be modeled as a set of interconnected subcomponents.

The HDL design process [41] is illustrated in Figure 3.1. The design process begins with a specification phase where the functional requirements of the system are delineated. Regardless of the complexity of the system, it requires specifying combinations and interactions among simpler components. Once the requirements are established, individual modules are defined on a system level and their behaviors modeled. Hardware design flow then describes functions at the logic level and upon synthesis; Boolean functions of the system are derived and mapped to elements of an Application Specific Integrated Circuit (ASIC) or Field Programmable Gate Array (FPGA).

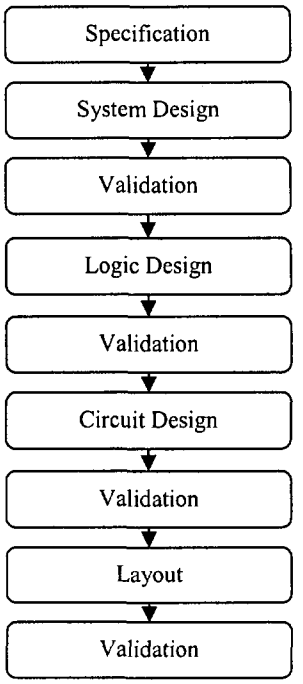

Figure 3.1 – Overall HDL design process from [41].

The use of an HDL to create a CAN node allows for the integration of the CAN controller subsystem along with other logic elements in an FPGA. For example, a complete microprocessor could be included if a sufficiently large FPGA is used. Using an FPGA provides flexibility in an environment that is characterized by the need for reconfigurability. In addition, HDL also provides an easy means for transferring the system design to full-custom silicon or other target.

3.1 CAN Module Specification

In general, a CAN module, or node, can be divided into functional blocks as illustrated in Figure 3.2. At the lowest level, the CAN bus lines, CAN high and CAN low, are connected to the module via a CAN transceiver. The transceiver controls both the transmission and reception of messages to and from the CAN bus. The next level up from

the transceiver is a CAN controller. The CAN protocol, which was discussed in Chapter 2, is implemented inside the CAN controller. The controller is oftentimes connected to a microcontroller that handles the overall application. Finally, the highest level is the application level, which can have a variety of inputs and outputs. For this work, the intended platform for the CAN network is the NJSP pursuit light package. The application aspect of this platform would be the "intelligence" of the respective functions of the light package; for instance, turning on the front strobe lights for 30 seconds and then switching to the rear strobe lights for another 30 seconds.

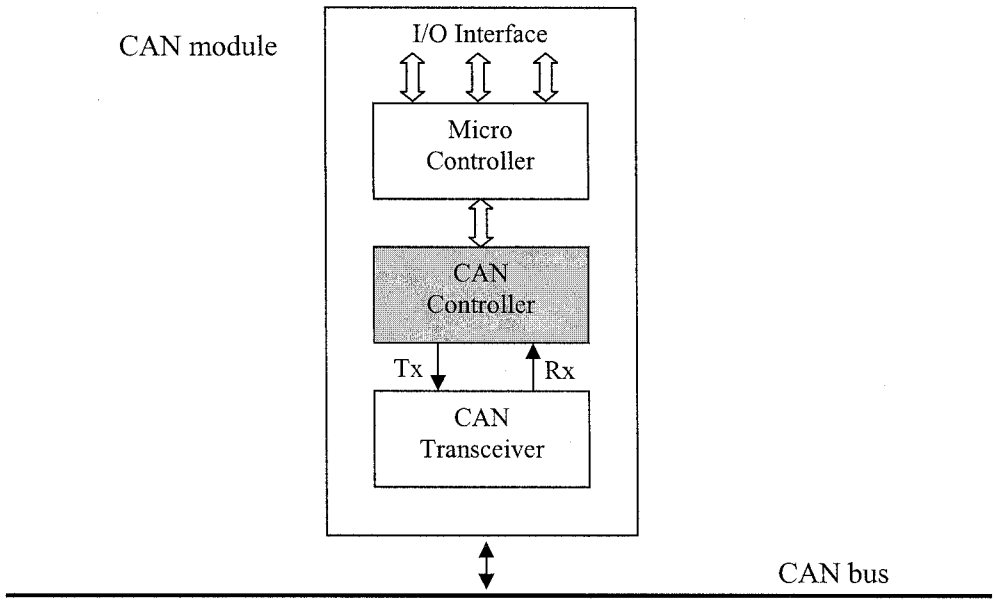

Figure 3.2 - Typical CAN module configuration.

3.2 CAN Protocol Controller

The approach chosen focuses on the CAN controller design with interfaces to integrate with an external transceiver and microcontroller. The first course of action for the HDL design process is to define all the requirements of the CAN controller. It is

assumed that the CAN controller will implement all tasks related to layers 1 and 2 of the CAN specification that were partially described in Chapter 2. Among the tasks included in the CAN controller are:

1.) Bus arbitration

2.) CRC check

3.) Frame generation

4.) Insertion and deletion of stuff bits

5.) Synchronization

6.) Error detection

Overall, the CAN controller should perform all the protocol related functions in a separate entity apart from a host microcontroller to relieve the microcontroller of time-critical tasks.

The CAN controller module can also be defined by functional blocks, as illustrated in Figure 3.3. Each task can therefore be equivalently modeled with a functional block in the CAN controller. The advantage of designing the system as a collection of fundamental blocks relieves the designer from the burden of attempting to create the system in one step. Instead, each block is built individually and then combined after all blocks have been completed. Furthermore, in keeping with sound software engineering principles, each block can be validated independently eliminating the need to troubleshoot an entire, complex system.

Defining the specifications as a list of requirements or as pseudo code, eases the transition from design concept to functional code. If the CAN controller core was being created from scratch, the following sections provide the requirements needed to make the

transition to HDL. However, since the CAN core has already been established the following specifications outline how the core was constructed.

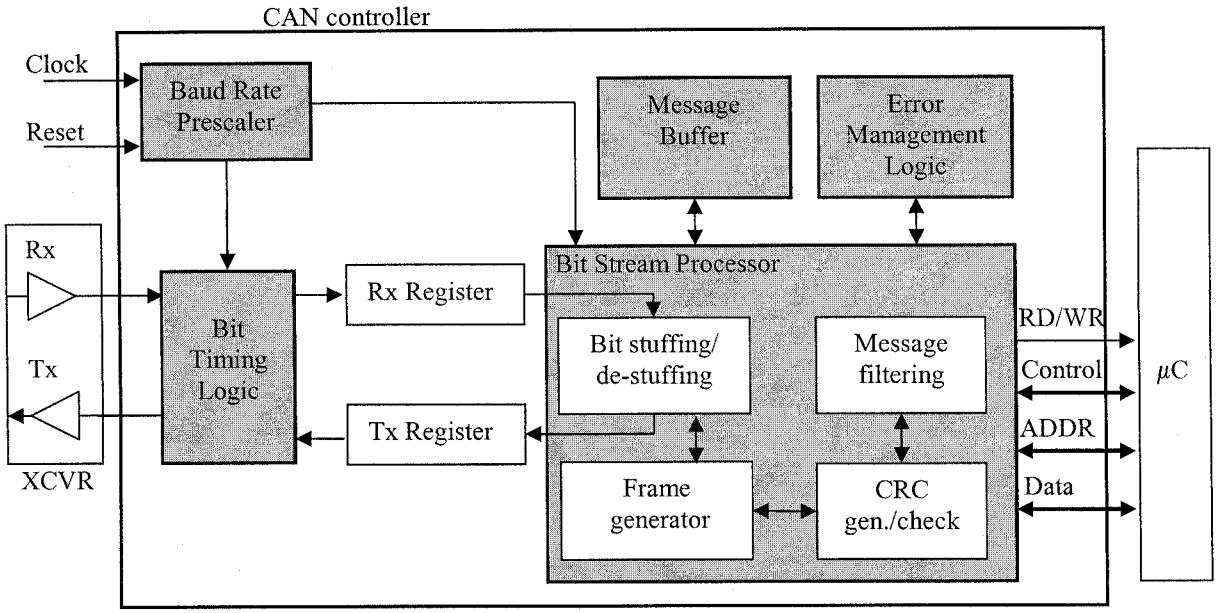

Figure 3.3 – Structure of the CAN protocol controller.

3.2.1 Bit Timing Logic

The first block that a bus message encounters is the bit timing logic (BTL). This block continually monitors the bus line input and handles the related bit timing according to the CAN protocol. When the BTL detects a transition from recessive to dominant, it synchronizes on that edge and reception of a new frame can begin. However, as was discussed in Chapter 2, there are propagation delays and phase shifts that affect the sampling point.

The BTL provides programmable time segments to compensate for delays. Bit timing configuration is programmed in two separate registers. Referring to Figure 2.6, the

sum of the propagation delay segment and phase buffer segment 1 (time segment 1) is combined with phase buffer segment 2 (time segment 2) in one register. In another register, the synchronization jump width (SJW), which defines how far a resynchronization may move the sample point, and BRP are combined. The programmable ranges of these segments are shown in Table 3.1.

Table 3.1 – CAN bit time parameters [8].

Parameter	Range
Synchronization segment	$1t_q$
Propagation delay segment	[1,2...,8] t_q
Phase buffer segment 1	[1,2...,8] t_q
Phase buffer segment 2	[1,2...,8] t_q
Synchronization jump width	[1,2...,4] t_q

In a CAN network, synchronization occurs on edges from recessive to dominant levels. The purpose is to control the distance between the edges and the sample points. Edges are detected by sampling the bus level in each time quantum and comparing it with the bus level at the previous sample point. Depending on the location of the edge relative to the synchronization segment, the phase error, e, will be negative, positive, or zero:

1) $e = 0$, if the edge lies within the synchronization segment.
2) $e > 0$, if the edge occurs after the synchronization segment.
3) $e < 0$, if the edge occurs before the synchronization segment.

There are two types of synchronization that can occur; hard synchronization, and resynchronization. Hard synchronization is performed at the start of a frame; inside a frame, only resynchronization can occur. After a hard synchronization, the bit time is restarted with the end of the synchronization segment. The hard synchronization forces the edge that caused the hard synchronization to be positioned within the synchronization segment of the restarted bit time.

Bit resynchronization involves shortening or lengthening the bit time such that the position of the sample point is shifted relative to the edge. When the phase error is positive, phase buffer segment 1 is expanded (Figure 2.7). If the magnitude of the error is less than SJW, phase buffer segment 1 is lengthened by the amount of the error; else, it is lengthened by SJW. Conversely, if the phase error is negative, phase buffer segment 2 is shortened. If the magnitude of the error is less than SJW, phase buffer segment 2 is shortened by the magnitude of the phase error; else, it is shortened by SJW.

The following rules apply for hard synchronization and resynchronization:

1) Only one synchronization is allowed between sample points within one bit time.
2) An edge will only be used for synchronization if the value detected at the previous sample point differs from the signal level immediately after the signal edge.
3) A hard synchronization is performed whenever there is a recessive to dominant edge during bus idle conditions.

These features of the bit timing logic make it possible to place the sample point at the right position inside of a bit depending on the quality of the signal received from the bus.

3.2.2 Transmit and Receive Registers

The transmit and receive registers serialize messages to be sent and parallelize messages that are received. These registers hold the data stream bits to allow access to the data by the bit stream processor (BSP), which controls the loading and shifting of the registers.

3.2.3 Bit Stream Processor

The bit stream processor (BSP) performs several protocol tasks. The BSP translates messages into frames and frames into messages. Messages are stuffed and destuffed within the BSP and undergo CRC checking while in the BSP. The BSP is also responsible for performing the process of bus arbitration (Figure 3.4) according to the

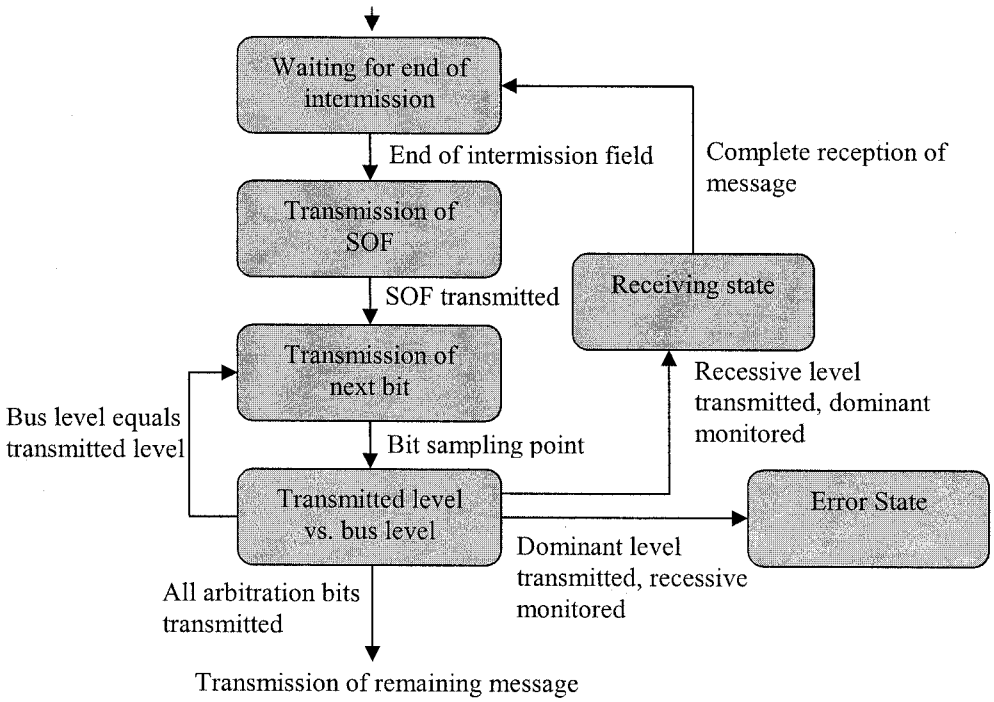

Figure 3.4 – State diagram of bus arbitration process [24].

protocol. Recall that the protocol describes bus arbitration in a non-destructive, bit-wise process (section 2.2.1.1). While the bus is idle, the bus is in the recessive state. Arbitration is initiated with a change from recessive to dominant via the start of frame bit. Every successive bit transmitted is compared with the bus level as illustrated in the state diagram in Figure 3.4. If the transmitted level matches the bus level, then the next bit is transmitted. If the bus level does not match the transmitted level, then the node either becomes a receiver or there is an error. Once a node has completed the arbitration process, the rest of the message (including control and data fields) is transmitted.

3.2.3.1 Frame Generator

Depending on the CAN format (standard or extended), the indexing of the data frame is known. Therefore, as the received message is accepted its corresponding frames can be determined based on the indexing. For example, the data frame illustrated in Figure 2.4 shows that the arbitration field consists of the first 12 bits. The first bit in that field is the SOF and the next 11 bits describe the identifier field. With this format defined the frame generator simply reads the message stream and upon reception of the SOF begins to index the message. Therefore, the first bit is stored in a register labeled SOF and the next 11 bits in a register labeled identifier. The same holds true for the remaining message and the corresponding fields related to the data frame.

3.2.3.2 Bit Stuffing and De-stuffing

The method of bit stuffing is applied to the following frame segments: SOF, arbitration field, control field, and CRC sequence. When transmitting a message stream, the transmitter inserts a complementary bit after detecting five consecutive bits of

identical value. When receiving a message, the stuff bits are removed and the message is broken down into frames.

3.2.3.3 Cyclic Redundancy Check

One component of the BSP generates the CRC code to be transmitted within the data frame, as well as checking the CRC code of incoming messages as described in 2.2.1.2.1. The equivalent 15-bit shift register CRC_REG (14:0) implementation demonstrated with pseudo code [8] in Figure 3.5 exhibits how the CRC sequence would be developed using an HDL. The resulting CRC sequence is stored in the 15-bit register after completion of a data frame.

```
CRC_REG (14:0) = (0, ..., 0);          //Initialize shift register
Repeat
CRC_NEXT = NEXT_BIT XOR CRC_REG (14);
CRC_REG (14:1) =CRC_REG (13:0);        //Shift left by one position
CRC_REG (0) = 0;                       //Fill LSB with 0
        If CRC_NEXT then
                CRC_REG (14:0) = CRC_REG (14:0) XOR (4599H);
        End
Until (NEXT_BIT = end of data or error condition)
```

Figure 3.5 – CRC 15-bit shift register algorithm [8].

3.2.3.4 Message Filtering

Data transmission in a CAN network is based on the principle of message broadcasting. All messages transmitted on the bus are accessible to all nodes and received by the controller. However, each node is only interested in those transmitted messages intended for it. For this reason, each CAN node implements a message filtering mechanism in the protocol controller to ensure that the host microcontroller is only informed of a message that is relevant for it.

A simple method involves constructing an acceptance filter using an 8-bit acceptance mask (Figure 3.6). The acceptance mask register contains the eight most significant bits of the message identifier ID [10:3], which a message must have to pass the filter. The bits of the acceptance mask can be set to 0, 1, or don't care (X). If a bit in the mask is set to 0, then the mask will only allow identifier bits of value 0, and if the mask value is set to 1 then the mask will only allow bit values of 1 to filter through. If the

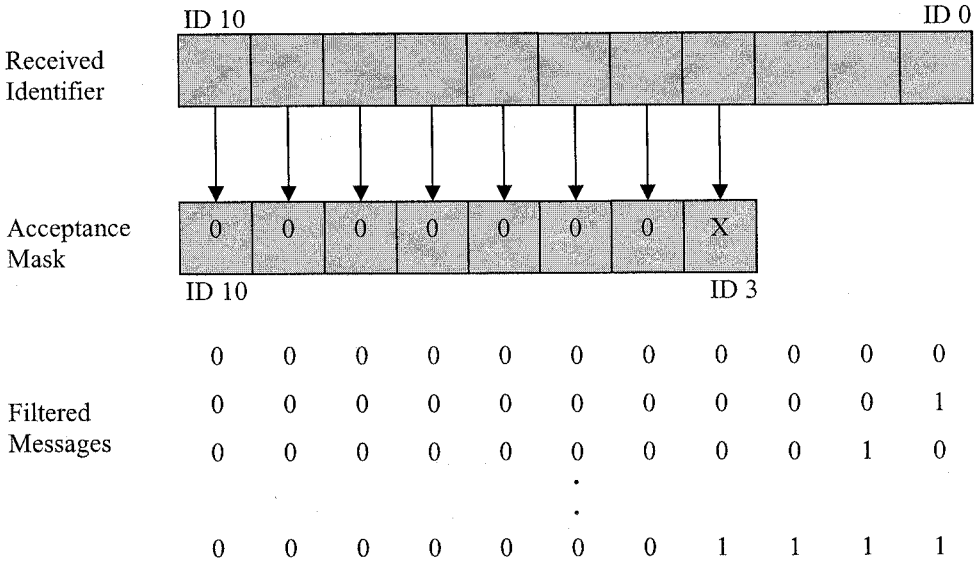

Figure 3.6 – Basic principle of message filtering.

mask is set to don't care, then bit values of either 1 or 0 can filter through. Figure 3.6 illustrates an acceptance mask set up to accept values 0-15. This is accomplished by setting all mask bits to 0, except for ID 3, which is set to a don't care state.

CAN implements a two-register acceptance filter. The two registers, acceptance code and acceptance mask, contain ID [10:3]. The acceptance code register contains the bits of the message identifier, which a message must have. The acceptance mask register then determines which of those bits are relevant for filtering. Therefore, if both registers are set to 0, then the range of filtered messages have identifier values from 0 to 7. If both registers are completely set to don't care, then all messages with identifiers from 0 to 2048 are received.

Once the messages are filtered, they can then be sent to the host microcontroller. The host microcontroller will subsequently interpret the messages and perform the tasks that are related to them.

3.2.4 Message Buffer

The message buffer is composed of a transmit and receive buffer. The transmit buffer is an interface between the host microcontroller and the BSP that is able to store a complete message that will be transmitted over the network. To fill the buffer the host microcontroller writes to the buffer the frame information and then the BSP reads the data and the information is then sent.

Conversely the receive buffer stores received messages that have passed through the message filtering. The buffer represents a FIFO, first in first out, that allows the host microcontroller to process one message at a time. Once a message is read from the FIFO, the message is released and the next message is pushed up.

The layout of the receive and transmit buffers are detailed in the product specification for the Philips SJA1000 stand-alone controller [39]. The address allocation and layout of these buffers are shown in Table 3.2.

Table 3.2 – Transmit and receive buffer layout of the CAN controller [39].

Address	Segment	Name
10	Transmit Buffer	Identifier (ID 10:3)
11		Identifier (ID 2:0), RTR, DLC
12		Transmit data 1
13		Transmit data 2
14		Transmit data 3
15		Transmit data 4
16		Transmit data 5
17		Transmit data 6
18		Transmit data 7
19		Transmit data 8
20	Receive Buffer	Identifier (ID 10:3)
21		Identifier (ID 2:0), RTR, DLC
22		Receive data 1
23		Receive data 2
24		Receive data 3
25		Receive data 4
26		Receive data 5
27		Receive data 6
28		Receive data 7
29		Receive data 8

3.2.5 Error Management Logic

The BSP continually monitors the data stream for errors as it processes the data stream. The BSP looks for the five different error mechanisms discussed in section 2.2.1.3. If the BSP detects a violation in any one of the mechanisms, the result is an error and the BSP notifies the error management logic (EML). In turn, the EML increments either a receive error counter (REC) or a transmit error counter (TEC) depending upon

whether the error occurred during transmission or reception. According to the CAN protocol, the values of the error counter determines if the controller is an error-active, error-passive, or bus-off state.

The different error states make it possible to implement error confinement, which ensures that a faulty node will not block communications along the bus. A node in an error-active state takes part in normal communications and sends an active error flag when errors are detected. An error-passive node having a relatively high error count can still take part in communications, but can only send passive error flags if further errors are detected. The passive error flag does not disturb the remaining bus communications. Finally, a node in the bus-off state is not allowed to have any communications on the bus. Figure 3.7 illustrates the error state diagram of a CAN node.

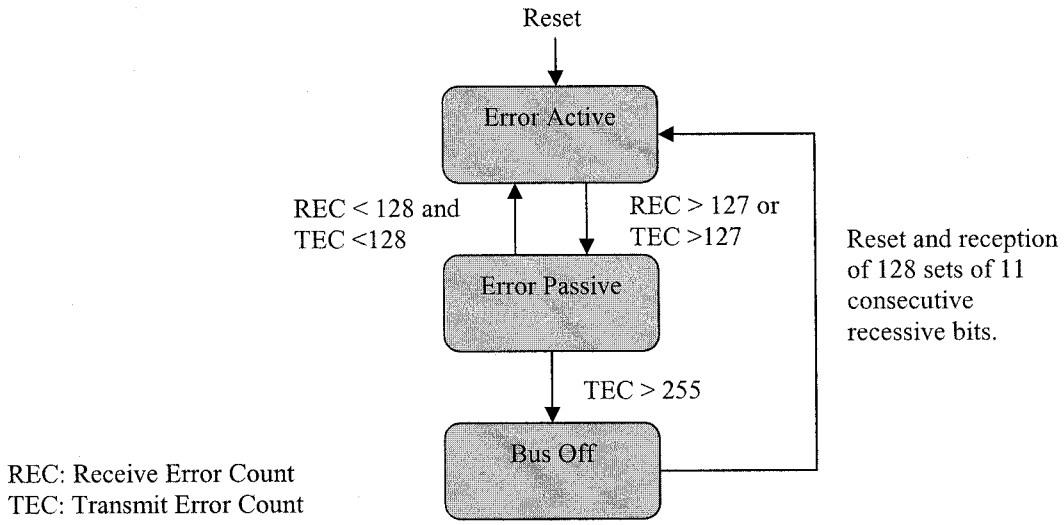

Figure 3.7 – Error state diagram of a CAN node [8].

After a reset, a CAN node is immediately placed in an error-active state. Once either one of the error counters exceeds 127, the node then enters the error-passive state.

A node can return to the error-active state if both counters are below 128. Once the transmit error counter exceeds 255 the node is disconnected from the bus. From this state, a node can return to the error-active state only when the counters are reset to zero after the node has detected 128 sequences of 11 consecutive bits of recessive value.

The error counters are incremented and decremented by the following set of rules defined by the CAN protocol:

1) When a receiver detects an error, the REC is increased by one, except when the detected error is a bit error during the transmission of an active-error flag or an overload flag.
2) When a receiver detects a dominant bit as the first bit after sending an error flag, the REC is incremented by 8.
3) When a transmitter sends an error flag the TEC is increased by eight.
    a. If the transmitter is error passive and detects an acknowledgement error, but does not detect a dominant bit while sending its passive error flag, the TEC is not incremented.
    b. If the transmitter transmits an error flag due to a stuff error during arbitration, when the stuff bit should have been recessive and was transmitted as recessive, but is monitored as dominant the TEC remains unchanged.
4) If a transmitter detects a bit error while sending an active-error flag or an overload flag, the TEC is increased by eight.
5) If a receiver detects a bit error while sending an active-error flag or an overload flag, the REC is increased by eight.

6) Any node endures up to seven consecutive dominant bits after transmitting an active-error, passive-error, or an overload flag. After detecting the fourteenth consecutive dominant bit following an active-error flag or the eighth following a passive-error flag, and after each additional sequence of eight consecutive dominant bits, every transmitter increases its TEC by eight. Simultaneously, every receiver increases its REC by eight.
7) After the successful transmission of a frame, the TEC is decremented if it currently has a non-zero value.
8) After the successful reception of a frame, the REC is decremented if it is between 1 and 127. If the count is zero, it remains unchanged; if it was greater than 127, it is set to a value between 119 and 127.

The rules for fault confinement state that the TEC of a node is increased by eight when a node detects an error during transmission and is only decreased by one for a successful transmission. As a result, a CAN network remains operational when less than every 16 frames is flawed.

3.3 FPGA Implementation

The CAN protocol and the HDL implementation of the protocol are established entities. Synthesizing the CAN model in a Field Programmable Gate Array (FPGA) [42] has also been explored since the inception of CAN. Utilizing CAN as an in-vehicle network for various automobiles has been achieved and has been discussed in Chapter 1. However, an aspect that has never been investigated nor fully exploited is the combination of these three aspects applied to an application related to law enforcement vehicles. Section 1.4 discussed potential opportunities to apply the knowledge of CAN to

a new platform, in particular the pursuit light package of the NJSP Troop car (Figures 1.1 and 1.2).

To demonstrate the ability of CAN to control the NJSP light package, the CAN controller core needs to be synthesized into a FPGA. An FPGA offers configurable logic blocks to allow a designer's program to define the function of the FPGA. This results in a custom logic circuit that can be reconfigured. As mentioned earlier, the automotive environment is characterized by the need for reconfigurability; the FPGA allows reprogramming the part so that it isn't necessary to supply a new FPGA each time a modification is made.

3.3.1 FPGA with CAN Applied to Light Control

Modeling the control of the NJSP light package can be demonstrated without direct installation of the CAN network into the vehicle. Instead, a proof-of-concept approach will be explored. To accomplish this goal, the synthesized FPGA with CAN is evaluated in a lab environment. The assumption that is made is that if the FPGA design can demonstrate the ability to control a modified light package, it is suitable to control the actual NJSP light package.

The application controls when a given light or group of lights in the package are to be turned on, and for how long they are to remain on before being turned off. Once turned off, another light or set of lights can be activated, and the process continues to create a desired light pattern. For example, in a pursuit condition, the pattern can consist of the front strobes activated for 10 seconds, followed by the rear strobes for 10 seconds, while all the LEDs are activated continuously.

Commanding a light pattern is initiated by the user using a switch interface. For this approach, the activation of a switch generates a CAN message. Embedded in that message are two key components, the intended address, and the data to be transferred to that address. In the complete light package, each of the 14 lights could be individually addressed or a combination of lights could be addressed together. The data transmitted to a CAN controlled light can vary, but in the simplest form, a logic level 1 would turn on a light and logic level 0 would turn off a light.

Translating logic levels to operate a light is accomplished by using a transistor as a switch to turn on and off the current through a load. Using the convention that a binary 1 represents a "high" voltage, typically 5 V, and a binary 0 represents a "low" voltage or 0 V, the following circuit (Figure 3.8) is an example of using digital signals to use a

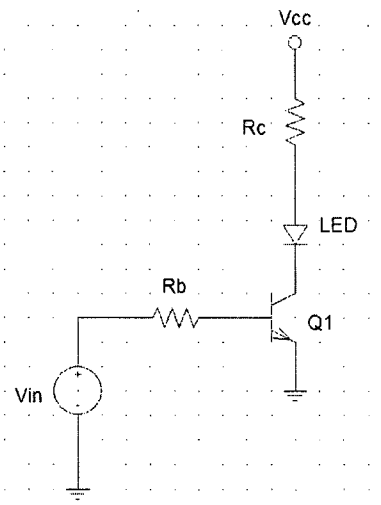

Figure 3.8 – Transistor used as a basic digital switch.

transistor as a switch. If V_{in} is high, then the transistor is driven to saturation and the LED is in the "on" state. However, if V_{in} is low then the LED is in the "off" state.

3.3.1.1 CAN Data Frame Coding

The proposed approach begins with defining the message that will be transmitted when the switch is activated. To begin with, the message that will be generated will be a standard CAN frame, which has an 11-bit identifier. Since this is a modified and limited representation of the light package, the choice of address to assign the FPGA is arbitrary.

The data field can be set using the format explained earlier. Logic level 1 will be interpreted as "on" and logic level 0 will equate to "off." This is a simple format and ideal for controlling a single light, but with one data byte, or eight bits, then 8 lights can be controlled using this format. This is accomplished by defining each bit position as a light and assigning either a one or zero to each position as shown in Table 3.3.

The bit stream produced by the coding scheme is then processed by the host microcontroller; if a logic level 1 appears in any of the bit positions, the microcontroller will supply a transistor switch for the appropriate light or lights with a "high" voltage condition and place the circuit in the "on" state. Conversely, a logic level 0 would place the circuit in the "off" state.

Before any lights are activated the complete data frame that is to be transmitted over the CAN bus must be defined. To start, the SOF bit (logic 0) marks the beginning of the data frame. The 12-bit arbitration field follows the SOF. The 11 most significant bits in this field represent the identifier ID [10:0], or the address. The choice of address is arbitrary since this is a limited network representation, but for simulation purposes the identifier will be set to 741h. The RTR bit is also contained in the arbitration field, and this is transmitted as logic 0 for a data frame.

Table 3.3 – Data coding using single bit position per light.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Light State
X	X	X	X	X	X	X	0	Light 1 off
X	X	X	X	X	X	X	1	Light 1 on
X	X	X	X	X	X	0	X	Light 2 off
X	X	X	X	X	X	1	X	Light 2 on
X	X	X	X	X	0	X	X	Light 3 off
X	X	X	X	X	1	X	X	Light 3 on
X	X	X	X	0	X	X	X	Light 4 off
X	X	X	X	1	X	X	X	Light 4 on
X	X	X	0	X	X	X	X	Light 5 off
X	X	X	1	X	X	X	X	Light 5 on
X	X	0	X	X	X	X	X	Light 6 off
X	X	1	X	X	X	X	X	Light 6 on
X	0	X	X	X	X	X	X	Light 7 off
X	1	X	X	X	X	X	X	Light 7 on
0	X	X	X	X	X	X	X	Light 8 off
1	X	X	X	X	X	X	X	Light 8 on

The control field consists of an additional six bits. The first bit in this field, IDE, indicates what CAN format is being applied. For standard format, the IDE is transmitted as 0 along with the r0 bit, which is reserved for future expansion. The last four bits of this field is the data length code which determines the number of data bytes in the succeeding data field. For one byte of data, the DLC code is 0001 (Table 2.1).

Following the control field is the data field. In this field is the 1 byte of data that will be coded as demonstrated in Table 3.3. The value of this data byte will be set to 12h. The 16-bit CRC field is then calculated from the bit stream starting from the SOF to the final data bit and is appended to the frame after the data field. The CRC computation uses the method demonstrated in Figure 3.5 and for the values that have been defined is calculated to be 30BBh. The CRC sequence is followed by the CRC delimiter bit and is transmitted high. The acknowledgement field follows the CRC delimiter and consists of

two bits transmitted high. The data frame is completed with a sequence of seven recessive bits and the recessive acknowledgement delimiter bit. Table 3.4 illustrates the bit streams for each bit field in the data frame.

Table 3.4 – Format of the data frame bit fields.

Bit Field	Bit Stream
Start of Frame	0
Arbitration Field	111010000010
Control Field	000001
Data Field	00010010
CRC Field	011000010111011
Acknowledgement Field	11
End of Frame	11111111

Once constructed, the data frame needs to undergo the bit-stuffing process before it can be placed onto the bus. In a data frame, the frame segments SOF, arbitration field, control field, data field, and CRC sequence undergo bit stuffing (Figure 3.9). With a stuff width of five, a complementary bit is inserted in the bit stream after five consecutive bits

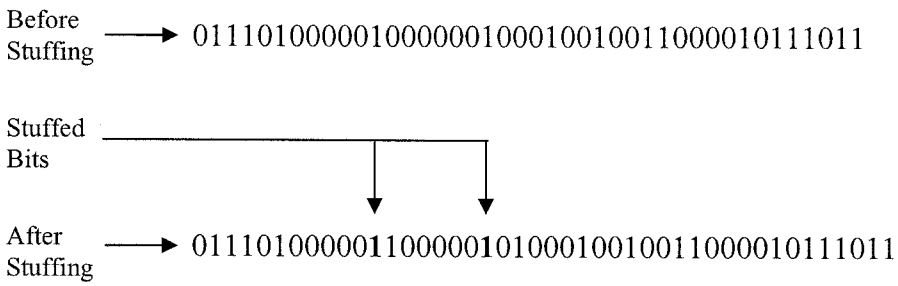

Figure 3.9 – Bit-stuffing of a sample CAN frame from SOF to end of CRC sequence.

of identical value. Figure 3.9 illustrates bit stuffing using the values of the bit fields in Table 3.4. Notice that in this example the frame is extended by two bits.

3.3.1.2 Transmitting and Receiving a CAN Frame

Combining the synthesized core, switch interface, and light control into a complete CAN node is accomplished with the use of the Digilent Digilab2 FPGA development board, which is based on the Xilinx XC2S200 FPGA [43], host microcontroller, and a CAN transceiver. The application diagram for the CAN node is detailed in Figure 3.10. Typically, a CAN controller such as the SJA1000 is interfaced with an 8-bit microcontroller of the 8051 family. However, the interface occurs via a multiplexed 8-bit data/ address bus and thus the SJA1000 is compatible with all common

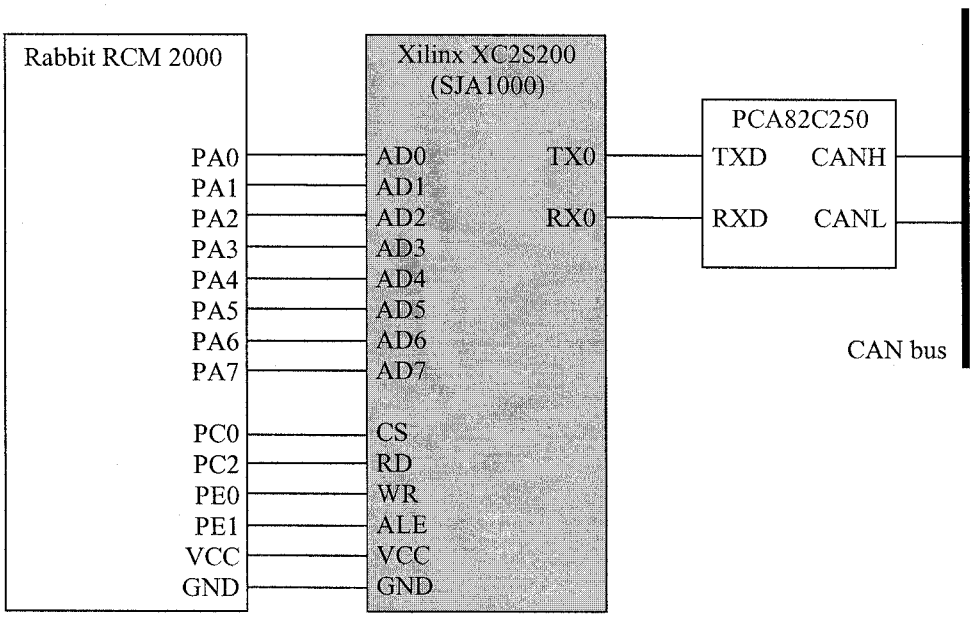

Figure 3.10 – Application diagram of the CAN node implementation.

8-bit microcontrollers. The basic I/O provided by the microcontroller allows for the construction of the CAN message via an on board push button. Depressing the push button will generate a CAN message in accordance to how it has been defined in Table 3.4. After the message is generated, it is sent to the FPGA, which contains the synthesized CAN controller core. The transmitting FPGA sends the message to a receiving CAN core via an attached CAN transceiver, the Philips PCA82C251 transceiver [44]. The receiving CAN node filters the message and passes along the data field to its host microcontroller. The embedded message is interpreted by the microcontroller and the corresponding lights are turned on or off. An overall block diagram of this approach is presented in Figure 3.11.

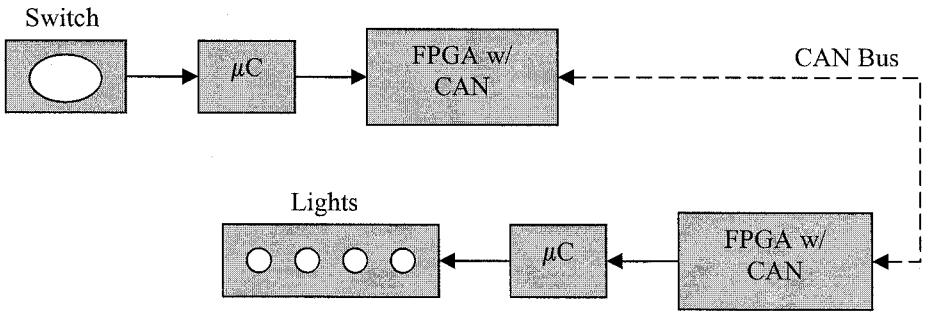

Figure 3.11 – Process flow of FPGA light control approach. The user depresses the switch to activate a light pattern. A CAN message is generated and sent to the receiving FPGA, which filters the message. The microcontroller reads the data field and controls the lights accordingly.

3.3.1.2.1 Controlling the CAN Functions

The link between the user interface, the push button or other switch interface, and the CAN message generated by the process of activating the switch interface is the host microcontroller. Since the CAN controller core that is being used in this thesis is a stand-

alone controller that only implements the CAN protocol, there is a demand for another controller that manages the functionality that goes beyond the basic protocol processing. In all CAN controllers there are a series of registers and message buffers that need to be configured and monitored. These tasks plus management of reading received messages and supplying messages to be transmitted are performed by the host microcontroller. The microcontroller used in this thesis is the Rabbit 2000 8-bit microprocessor [45].

For connection to the host microcontroller, the SJA1000 provides a multiplexed data/address bus (*AD [7:0]*) and additional read/write control signals (*ALE,RD,WR, and CS*) as shown in Figure 3.10. Before any communication can be initiated the controller must go through a series of initialization actions that configure the device for communications (Figure 3.12). Upon power-on or after a reset, the microcontroller uses these control signals to perform an initialization that configures the CAN controller for CAN communication. The initialization process begins with a reset pulse, low logic applied to the reset pin, enabling the CAN controller to enter the reset mode. With the reset mode enabled the microcontroller can then configure the necessary registers. The configuration of these registers defines such aspects as the acceptance code and acceptance mask, bit rate, sampling point, and other important properties necessary for CAN communication.

After configuration the CAN controller can then be started. With the start of the CAN controller, the controller can begin to receive messages (see Figure 3.12) off the CAN bus and also transmit messages (see Figure 3.13). The flow diagram of message reception is demonstrated in Figure 3.12. If a message is waiting in the receive buffer the host microcontroller is notified and is able to read the message. Once the message is read

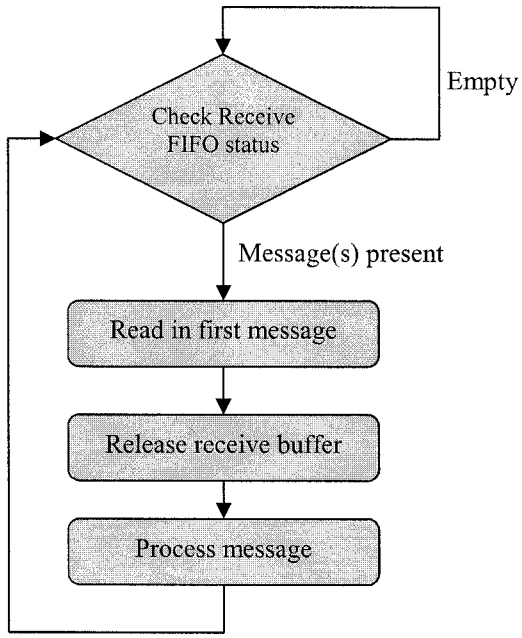

Figure 3.12 – Flow diagram of message reception [46].

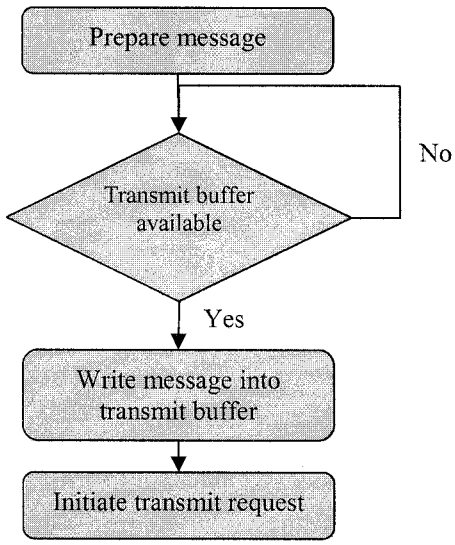

Figure 3.13 – Flow diagram of message transmission [46].

the host microcontroller sends a command to the CAN controller to release the receive buffer and the process of reading incoming messages continues. To transmit a message (see Figure 3.13), the message is copied from the host microcontroller to the transmit buffer contained in the CAN controller. The microcontroller initiates a transmit request command and the message stored in the transmit buffer is sent. The overall general program flow of the microcontroller tasks are described by Figure 3.14. Message transmission and message reception are covered under the main process of the application.

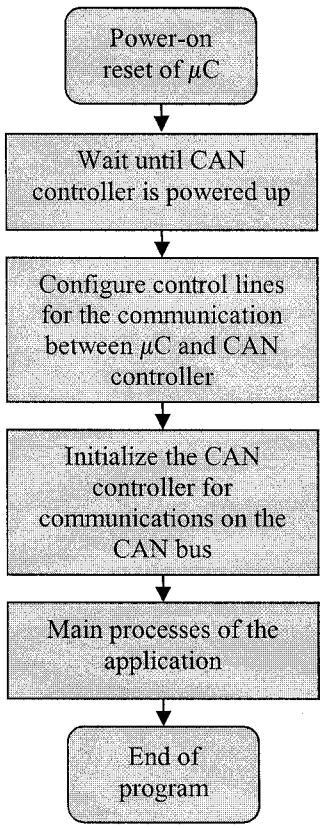

Figure 3.14 – General program flow of the host microcontroller [46].

For this thesis work the C-language is used to describe the program flow of the SJA1000. The target controller is the Rabbit 2000 8-bit microprocessor. The program embedded in the microprocessor defines the initialization procedure of the CAN controller and controls when the CAN controller is to send and receive messages to and from the CAN bus.

Chapter 3 has presented an overall approach for defining requirements necessary to describe the CAN protocol in a hardware description language. Also, a specific approach for designing and implementing the CAN protocol in a FPGA to control a light application related to a law enforcement platform has been detailed. This approach is a proof of concept methodology aimed at demonstrating the ability of a serial network such as CAN to control a complete light package typical of the NJSP troop car. The following chapter presents the compilation of results obtained by applying the methods proposed in this thesis.

Chapter 4: Results

OPENCORES.org provided the HDL code for the CAN protocol controller, which again is modeled after the Philips SJA1000 stand-alone CAN controller [39]. This code contained the various subsystems of the controller core as described in Chapter 3. This code was developed under the agreement that to implement the CAN IP core on silicon a CAN protocol license must be obtained. However, there are no legal restrictions for development of an open source IP.

The code was then synthesized into a Xilinx XC2S200 FPGA [43] using Xilinx software tools and interfaced with the Rabbit RCM2000 microprocessor [45] and the Philips PCA82C251 transceiver [44]. Validating that the synthesized FPGA exhibits the functioning of a CAN communication node was achieved by the approach detailed in the preceding chapter. The proceeding sections present the results of this process.

4.1 Synthesis

Before downloading a design to a FPGA the code is synthesized. Synthesis is a process of translating HDL to a gate level netlist and the code is optimized for the target device. The first step of synthesis is to compile the HDL source code and check it for possible syntax errors. If syntax errors do occur they are corrected and the code is recompiled. Although it may seem like a direct step there are some constructs that are not easily accepted by the synthesis tool, Xilinx Foundation Series 4.2i. Some of the common restrictions that occurred when compiling the code were: initialization of signals and variables, restricted used of wait statement, and assigning delays. After a successful syntax check the netlist is then generated.

4.2 Simulation

Once synthesis was completed the CAN controller core was simulated. To simulate the design the CAN controller was subjected to a testbench. A testbench is a top-level hierarchical model that instantiates a desired model, the device under test, and drives that model with a set of test vectors and generates the responses of the model. The block diagram for a testbench setup is illustrated in Figure 4.1.

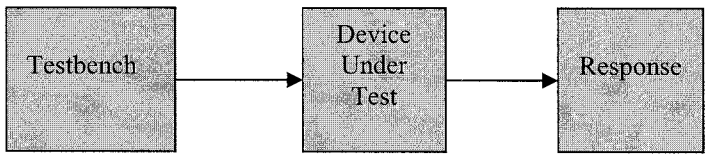

Figure 4.1 – The testbench model.

For the purposes of this thesis the testbench instantiated the CAN controller core and then proceeded to construct a CAN message with 1 data byte. The messages were then transmitted and the CAN controller received these messages and the messages were then stored to the receive FIFO discussed in section 3.2.4. Using the ModelSim design environment the following preliminary results were obtained.

Unlike the setup demonstrated in Figure 3.10, where one CAN node communicates with another CAN node, the configuration modeled under simulation is a self reception mode. In this configuration, as shown in Figure 4.2, an individual CAN node transmits a message with the receive line connected to transmit line to allow self reception of the transmitted message. This simulation will test the operation of the CAN node before synthesizing the model to an FPGA.

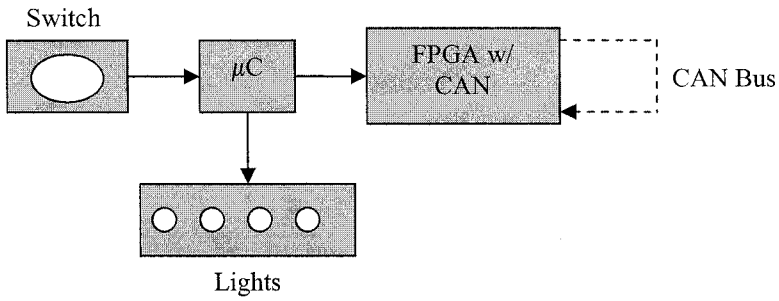

Figure 4.2 – Process flow of simulated FPGA light control approach using self reception mode. A message is generated and transmitted by the CAN node and the message is looped directly back to the receive line.

To begin with, the testbench sent a string of 11 recessive bits, logic level 1, indicating that the bus was idle. After this sequence the bus is available again and then a transmission request was initiated. To transmit a message, the transmission request bit is set in the command register. This is accomplished by writing this bit high into the command register. This register is addressable by CAN address 1 as specified by the SJA1000 datasheet [39]. Using the display command within the testbench, this process was able to be monitored and a portion of the transcript is shown in Figure 4.3.

After transmission is requested the message is able to be sent. For the simulation the standard CAN message was addressed to identifier 741h and contained one data byte of value 12h. The remaining bit fields are defined in Table 3.4. These are the raw bit streams and must undergo bit stuffing before they are transmitted. In addition to the fields shown, there is also 3 recessive bits appended to the end of the message. These bits represent an interframe space and combined with the acknowledgement field and the end of frame, make up the bus idle state.

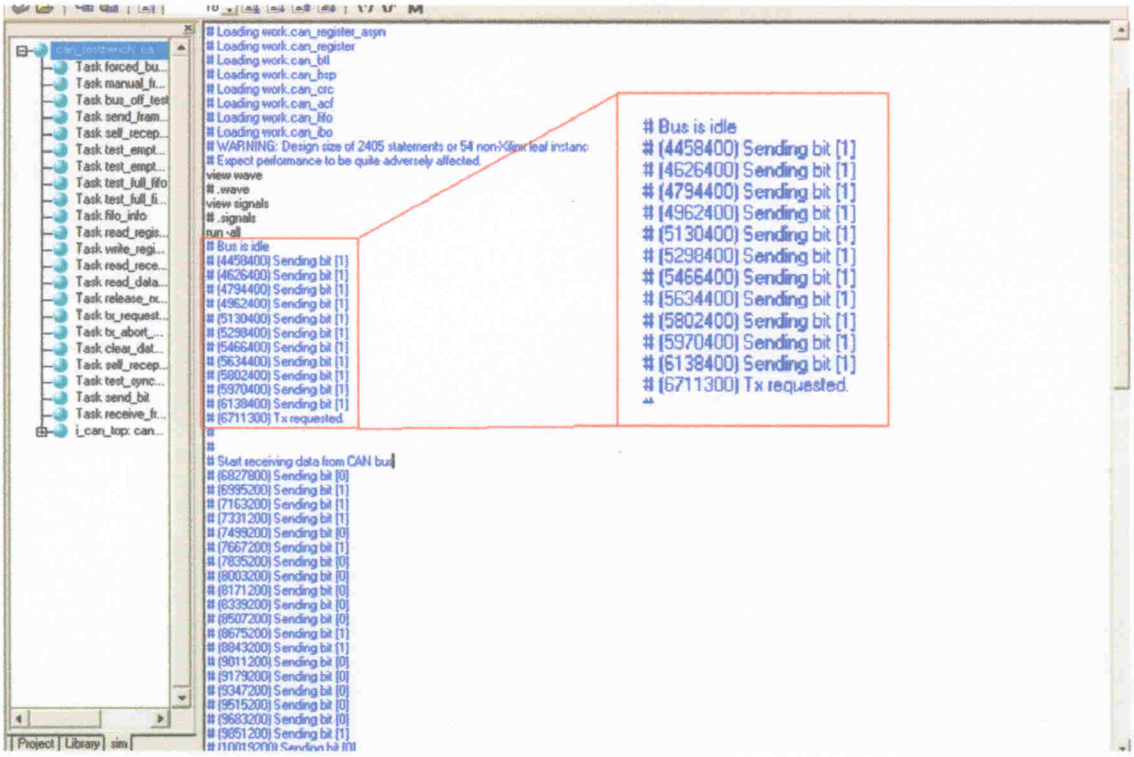

Figure 4.3 – Bus idle monitoring results from ModelSim.

Employing the display statement again the process of transmitting the CAN message bit by bit is monitored through ModelSim (Figure 4.4). The reception of the CAN message by the CAN controller is observed in the associated timing diagram (Figure 4.5) generated by the testbench. In the diagram it can be seen that as the message is received the receive signal *rx* changes logic levels according to the message. The receive signal starts with a sequence of recessive bits indicating that the bus is idle as expected. The request for transmission is initiated and the command register at address 1 is written high. At the next transition the start of frame bit indicates the beginning of a new frame and the message is received. Reception is able to take place because the bus was idle and there were no other communications taking place.

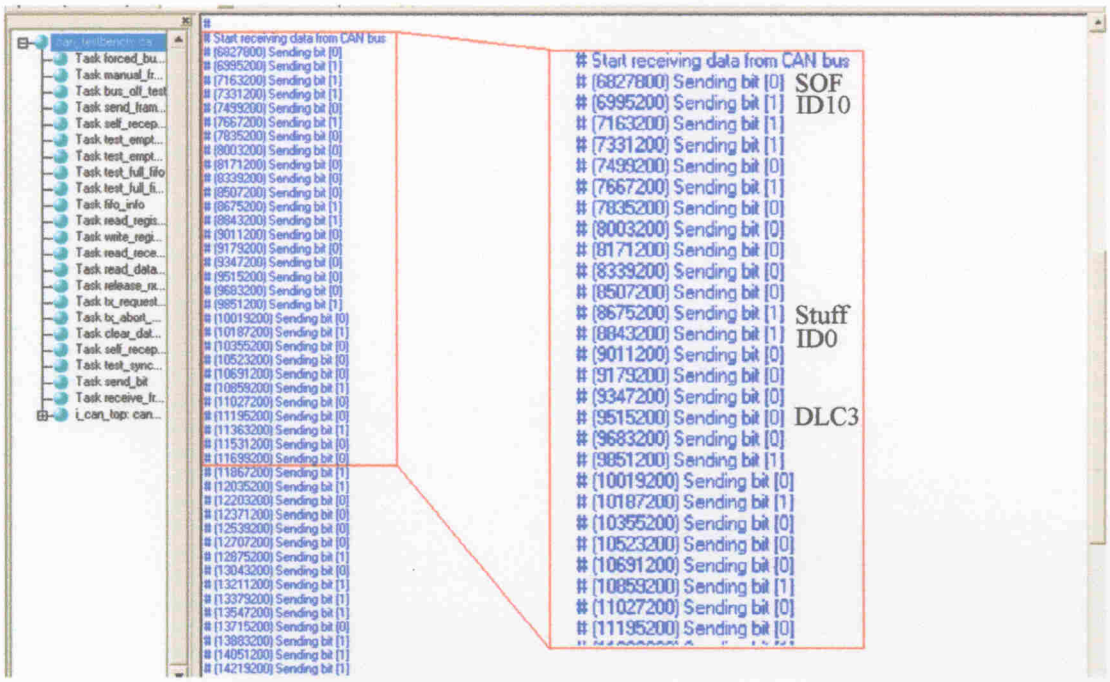

Figure 4.4 – Transmission monitoring through ModelSim.

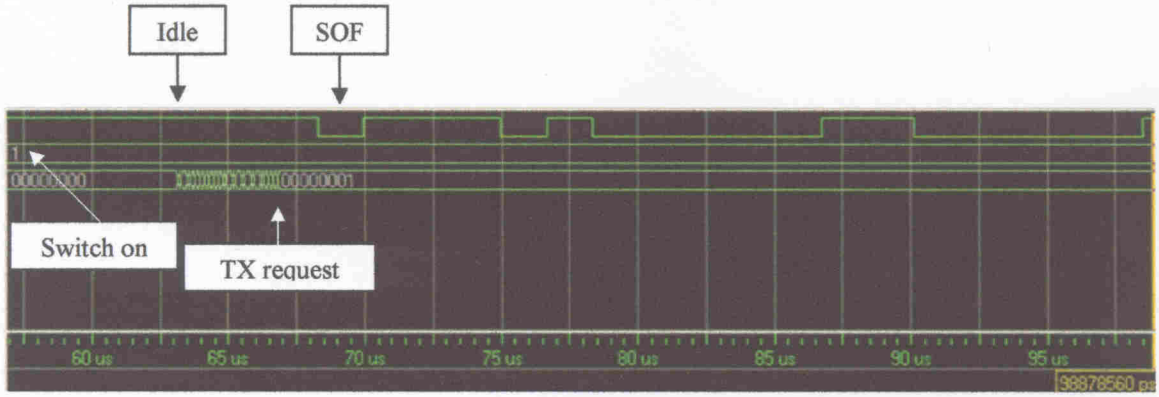

Figure 4.5 – Timing diagram of message reception.

If the received message is accepted by the message filtering scheme then the message is stored in the receive buffer. The most significant eight bits of the message identifier (ID 10:3) are stored in the identifier byte 1 at address 20. Identifier byte 2 at address 21 is determined by the three remaining bits of the identifier (ID 2:0), RTR, and the DLC. Addresses 22 to 29 in the receive buffer are reserved for the data bytes. Since, the message in this simulation contains one data byte of value 12h, this value is stored to address 22. Table 4.1 illustrates what the receive buffer should contain after the simulation.

Table 4.1 – Layout of receive buffer after reception of message for the first simulation.

Address	Name	Value
20	Identifier (ID 10:3)	11101000 → E8h
21	Identifier (ID 2:0), RTR, and DLC	00100001 → 21h
22	Data Byte 1	00010010 → 12h
23	Data Byte 2	XX
24	Data Byte 3	XX
25	Data Byte 4	XX
26	Data Byte 5	XX
27	Data Byte 6	XX
28	Data Byte 7	XX
29	Data Byte 8	XX

Now that the message should be waiting in the receive buffer, the testbench reads the receive buffer and displays the results (Figure 4.6). As it can be seen in the graphic, the value read from the register at address 20 was E8h, address 21 was 21h, and the data

byte in address 22 was 12h. This is also confirmed in the timing diagram shown in Figure 4.7.

Finally, with the message received and positioned in the receive buffer, the testbench retrieves the data byte from the buffer. The data byte is then assigned to a signal defined as *light1*, which is a temporary holding variable. With a value of 12h or in binary form 00010010, it is evident that there are two bit positions containing an "on" command. However, for the purposes of this simulation it is interpreted that this is a command to turn on the LED. Therefore, it can be seen in Figure 4.8 that after register 22 is read the data it contains is copied to a holding variable. The interpretation to turn on the LED is performed and the LED signal is forced "high" or "on."

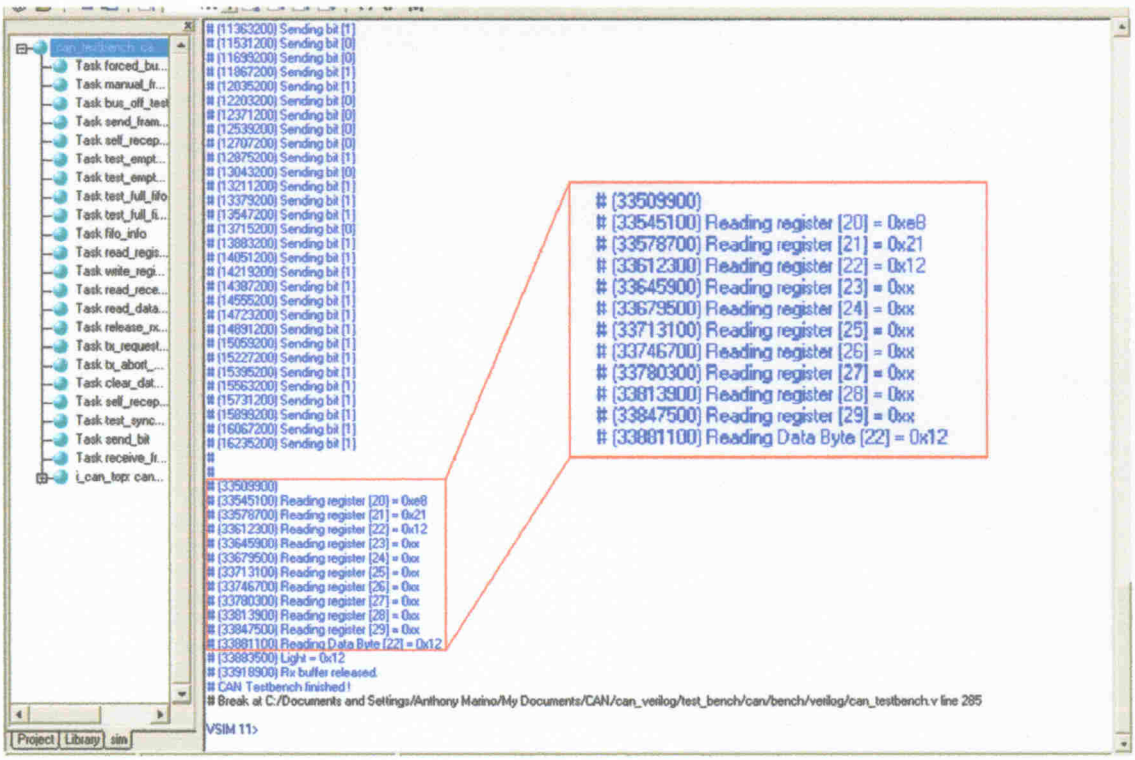

Figure 4.6 – Reading of the receive buffer monitored through ModelSim.

the host microcontroller, the embedded program in the microcontroller mainly reads and writes to the various registers. Operating the CAN controller in basic mode, the initialization process begins with enabling the reset mode by configuring the control register (see Table 4.2). The reset mode is entered when the reset request (RR) is set in the control register. Under the reset mode the registers that receive the configuration information are accessible. The host microcontroller then configures the clock divider register, acceptance code and acceptance mask registers, bus timing registers, and finally the output control register.

Table 4.2 – Bit layout of the control register; CAN address 0.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
-	-	-	OIE	EIE	TIE	RIE	RR
0	0	0	0	0	0	0	1

The clock divider register is an 8-bit register, the architecture of the clock divider register is shown in Table 4.3, that determines if the CAN controller will operate in the BasicCAN mode or the PeliCAN mode, which offers extended features. For this implementation the controller is operated in the BasicCAN mode and the microcontroller sets the CAN mode bit to logic 0. The clock divider register also determines if the controller provides an output clock for the microcontroller and at what frequency. Since there are two independent clocks in this implementation the CAN controller clock is deactivated and this is reflected with logic 1 for the clock off bit. The internal CAN input comparator of the CAN controller is bypassed since the external transceiver is being used. This is achieved by setting CBP to logic 1.

4.3 Two CAN Node Network Implementation

The simulation results obtained in the previous section demonstrated that the CAN controller was functional in a self reception mode, which only requires one CAN node (Figure 4.2). The implementation of more than one node will provide a better representation of the CAN functionality as it would eventually appear within a vehicle. As it was demonstrated in Figure 3.11 the two node network consists of a transmitting node and a receiving node. It is possible that both nodes can perform the actions of both transmitting and receiving CAN frames. However, in this implementation the transmitting node, which contains the user interface, represents the control point for the light application. The receiving node represents the CAN nodes that will be attached to each light in the light package. For the receiving nodes there is no need for a user interface since they will be remoted in various locations of the car to control the functions of the light package dependent upon the messages constructed by the transmitting node.

Once simulation is complete and the CAN controller is verified the HDL model can be implemented and programmed to the FPGA. With the FPGA programmed the data and control lines of the CAN controller are then connected to the microprocessor and the transmit and receive lines attached to the CAN transceiver as detailed in Figure 3.10. The microprocessor is then programmed with its respective control program and CAN communications can proceed.

4.3.1 Configuration and Initialization

As it was described in Section 3.3.1.2.1 the host microcontroller activates an initialization process that configures the CAN controller for communication. Since the CAN controller is seen as nothing more than a peripheral memory mapped I/O device for

the host microcontroller, the embedded program in the microcontroller mainly reads and writes to the various registers. Operating the CAN controller in basic mode, the initialization process begins with enabling the reset mode by configuring the control register (see Table 4.2). The reset mode is entered when the reset request (RR) is set in the control register. Under the reset mode the registers that receive the configuration information are accessible. The host microcontroller then configures the clock divider register, acceptance code and acceptance mask registers, bus timing registers, and finally the output control register.

Table 4.2 – Bit layout of the control register; CAN address 0.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
-	-	-	OIE	EIE	TIE	RIE	RR
0	0	0	0	0	0	0	1

The clock divider register is an 8-bit register, the architecture of the clock divider register is shown in Table 4.3, that determines if the CAN controller will operate in the BasicCAN mode or the PeliCAN mode, which offers extended features. For this implementation the controller is operated in the BasicCAN mode and the microcontroller sets the CAN mode bit to logic 0. The clock divider register also determines if the controller provides an output clock for the microcontroller and at what frequency. Since there are two independent clocks in this implementation the CAN controller clock is deactivated and this is reflected with logic 1 for the clock off bit. The internal CAN input comparator of the CAN controller is bypassed since the external transceiver is being used. This is achieved by setting CBP to logic 1.

Table 4.3 – Bit layout of the clock divider register; CAN address 31.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
CAN mode	CBP	RXINTEN	CDR.4	Clock off	CD.2	CD.1	CD.0
0	1	0	0	1	0	0	0

In the Basic mode the acceptance filter is defined by the acceptance code register and the acceptance mask register. As described earlier, the acceptance filter provides the capability of the CAN controller to filter received messages when the identifier bits of the received message match those of the acceptance filter. The acceptance code was set to zero and the acceptance mask was set to don't care, which results in an acceptance filter that allows every identifier to pass through. The resulting layouts of the acceptance code register and the acceptance mask register are detailed in Tables 4.4 and 4.5.

Table 4.4 – Bit layout of the acceptance code register; CAN address 4.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
AC.7	AC.6	AC.5	AC.4	AC.3	AC.2	AC.1	AC.0
0	0	0	0	0	0	0	0

Table 4.5 – Bit layout of the acceptance mask register; CAN address 5.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
AM.7	AM.6	AM.5	AM.4	AM.3	AM.2	AM.1	AM.0
1	1	1	1	1	1	1	1

Configuration of the bus timing is governed by the bus timing registers, bus timing register 0 and 1. Bus timing register 0 contains the values of the baud rate prescaler (BRP) and the synchronization jump width (SJW). The SJA1000 application note [46] defines values for these parameters for several examples. The application note assumes that a 24 MHz clock is being used to clock both the CAN controller and the microprocessor for this implementation. However, the CAN controller and the microprocessor are being clocked by separate clocks. The clock of the microprocessor for this implementation is 22 MHz and the CAN controller is clocked by its internal 24 MHz clock. For a bit rate of 1 Mb/s the BRP and SJW are both set to zero (see Table 4.6). The contents of the bus timing register 1 include time segment 1 (TSEG1), time segment 2 (TSEG2), and sampling (SAM), which define the length of the bit period, location of the sample point, and the number of times the bus is sampled. To reach the bit rate of 1Mb/s the application note defines TSEG1 to be 08h and TSEG2 to be 10h and the bus is sampled only once (see Table 4.7).

Table 4.6 – Bit layout of the bus timing register 0; CAN address 6.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
SJW.1	SJW.0	BRP.5	BRP.4	BRP.3	BRP.2	BRP.1	BRP.0
0	0	0	0	0	0	0	0

Table 4.7 – Bit layout of the bus timing register 1; CAN address 7.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
SAM	TSEG2.2	TSEG2.1	TSEG2.0	TSEG1.3	TSEG1.2	TSEG1.1	TSEG1.0
0	0	0	1	1	0	0	0

To complete the initialization process the output control register is configured to allow normal output mode and the driver characteristics of the transmit lines (TX0 and TX1) are defined. The application diagram detailed in Figure 3.10 shows that only TX0 from the controller is being used. Therefore, TX1 is floating and TX0 is configured as push/pull. This configuration is described by the bit layout described by Table 4.8.

Table 4.8 – Bit layout of the output control register 1; CAN address 8.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
OCTP1	OCTN1	OCPOL1	OCTPO	OCTN0	OCPOL0	OCMODE1	OCMODE0
0	0	0	1	1	0	1	0

Completion of the initialization and configuration routine is signaled with the deactivation of the reset mode. Following initialization, CAN message transmission and reception can then occur.

4.3.2 Message Transmission

Before a message can be transmitted across the CAN bus the message must first be constructed. The host microcontroller prepares the message, only the identifier, RTR, DLC, and data bytes need to be defined, and then transfers the message into the transmit buffer (see Table 3.2) of the CAN controller. The CAN controller will process the message defined by the microcontroller to ensure it conforms to the CAN protocol. The same identifier that was used for simulation is used again for this implementation with a data byte of value 01h. The resulting layout of the transmit buffer is detailed in Table 4.9.

Table 4.9 – Resulting bit layout after the host microcontroller writes the prepared message to the transmit buffer.

Address	Name	Value
10	Identifier (ID 10:3)	11101000 → E8h
11	Identifier (ID 2:0), RTR, and DLC	00100001 → 21h
12	Data Byte 1	00000001 → 01h
13 - 19	Data Byte 2 - 8	XX

Once the message is stored into the transmit buffer the message is ready for transmission. By writing a transmit request (TR) to the command register (see Table 4.10) the CAN controller is alerted that there is a message queued and the controller subsequently transmits the message. The status of the message transmission can be checked by monitoring the transmit buffer status (TBS) and transmission complete status (TCS) flags in the read only status register (see Table 4.11). If the TCS flag is present then the most recent requested transmission has completed successfully. If the message is not yet completed then the TCS is low and the TBS is also low indicating that a message is still waiting in the transmit buffer. Table 4.11 details the content of the status register after the successful transmission of the CAN message.

Table 4.10 – Bit layout of the command register for transmitting node; CAN address 1.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
-	-	-	GTS	CDO	RRB	AT	TR
X	X	X	0	0	0	0	1

Table 4.11 – Bit layout of the status register for the transmitting node; CAN address 2.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
BS	ES	TS	RS	TCS	TBS	DOS	RBS
0	0	0	0	1	1	0	0

4.3.3 Message Reception

Following the transmission of the message from the transmitting node the receiving node obtains the transmitted message and places it directly into the receive buffer (see Table 3.2). The resulting layout of the receive buffer (see Table 4.12) is analogous to the transmit buffer, with the only difference being the CAN addresses. The host microcontroller is made aware of the received message through constant polling of the status register. In addition to the transmit flags (TBS and TCS) the status register also contains a receive buffer status (RBS) flag that indicates whether or not the receive FIFO contains any messages. A logic 1 for the RBS flag reflects that there is one or more messages waiting in the buffer and a logic 0 reveals that the buffer is empty.

Table 4.12 – Resulting bit layout of the receive buffer after the CAN controller receives the CAN message from the transmitting node.

Address	Name	Value
20	Identifier (ID 10:3)	11101000 → E8h
21	Identifier (ID 2:0), RTR, and DLC	00100001 → 21h
22	Data Byte 1	00000001 → 01h
23 - 29	Data Byte 2 - 8	XX

Table 4.13 – Bit layout of the status register for the receiving node; CAN address 2.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
BS	ES	TS	RS	TCS	TBS	DOS	RBS
0	0	0	1	0	0	0	1

The host microcontroller performs a loop that polls the status register, continually checking if the receive buffer status flag indicates that at least one message is present in the receive buffer. If the RBS is low then the loop continues until the RBS is high, at which point the host microcontroller reads the first message from the receive buffer. After retrieving the message from the buffer the microcontroller sends a release receive buffer (RRB) command to the command register, as illustrated in Table 4.14. The release command purges the last read message and pushes the next message up to be read by the microcontroller.

Table 4.14 – Bit layout of the command register for the receiving node; CAN address 1.

BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
-	-	-	GTS	CDO	RRB	AT	TR
X	X	X	0	0	1	0	0

Before the host microcontroller checks for additional received messages it processes the most recently read message. Using the coding convention identified in Table 3.3 the embedded data byte in the received message, 00000001, reflects that all lights are off except for light 1. The microcontroller examines the bit positions of the data

byte(s) and determines if a light in the light package should be turned on or off according to the logic level present at each bit position. The flow diagram for the bit inspection process is detailed in Figure 4.9.

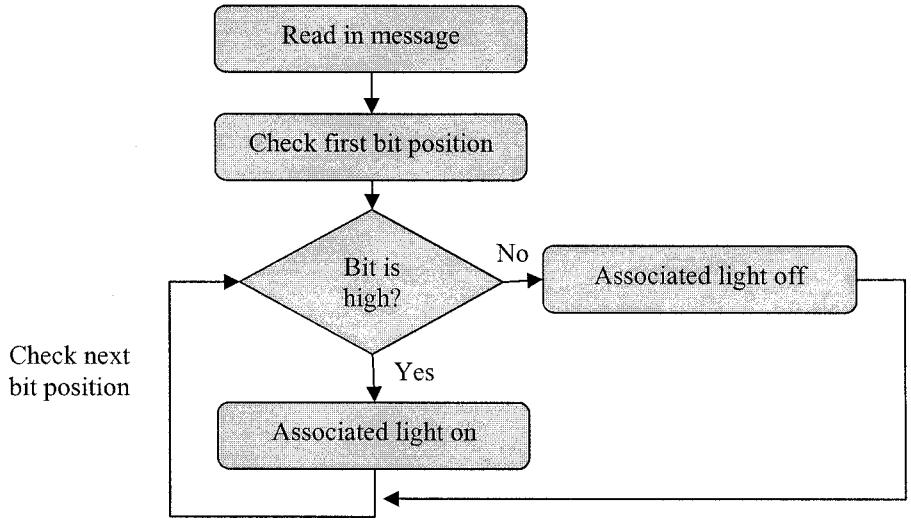

Figure 4.9 – Flow diagram for bit inspection of received data byte.

Chapter 4 has presented the results obtained by using the approach outlined in Chapter 3. The results have detailed the process of configuring CAN for communications and the process of sending and receiving messages over the CAN bus. With the results collected by the proof-of-concept system it can be concluded that CAN is capable of controlling the NJSP light package. Chapter 5 will continue to discuss the accomplishments of this work and recommendations on how to move this research ahead in the future.

Chapter 5: Conclusions

Increased wiring complexity in automobiles can be attributed to rising demands for new features, which in turn require more communication. In-vehicle telematics have transformed from mere entertainment, such as control of a CD player, to complex mobile information systems including global positioning units, vehicle tracking systems, and cellular communications. Additionally, law enforcement vehicles require another level of telematics not found in passenger cars. Unfortunately, excessive in-vehicle electronics and telematics represent new levels of complexity for the car interior. Integration of added displays, controls, and wiring becomes more challenging and can lead to decreased safety and convenience for the user.

Achieving successful integration of technologies presented in any vehicle requires a communications backbone providing a conduit for sharing information among all the electronic systems. Such a backbone will significantly reduce cabling costs and cabling volume and it will simplify technology installation. The automotive industry realized that future market success depended on the availability of such a system; since the beginning of the 1980s, almost every automotive company initiated research and development to define communication protocols. This research led to a number of protocols including CAN, OBD, MOST, and LIN. Since the early 1990s, CAN has become the protocol of choice for automotive applications due to its high performance, robust error detection and relatively low cost.

This thesis sought to implement the CAN protocol in hardware designed for the NJSP troop car platform. This work lays the foundation for a highly integrated police car that provides greater functionality to the user and makes the vehicle more flexible to

facilitate additional technology growth in the future. Further benefits include reduced wiring costs and weight.

5.1 Summary of Accomplishments

The methodologies and results presented in this thesis suggest a promising approach for achieving integrated control of in-vehicle electronics in conjunction with reduced cabling in a law enforcement application. The proposed network, CAN, offers a common data bus that would eliminate the need for point-to-point wiring of the current systems. Furthermore, this network provides the capability to control applications such as the NJSP light package.

When the user selects a light combination, a CAN message is generated that contains the address of the target CAN receiver node. The receiver node filters the CAN frame and decodes the embedded data field, which contains the instruction for turning on or off the selected lights.

The approach was designed and implemented as a proof-of-concept prototype. It is expected that the CAN nodes would work as presented in this research if installed in the NJSP vehicle. The design demonstrated the control of a modified light package; the actual light package would perform as successfully. CAN is versatile enough to accomplish more than just control. An entity with CAN capability can report diagnostic information obtained from other elements on the shared network, thus supplying the user with a "smart" technology suite. Above and beyond control and diagnostic applications CAN theoretically has the ability to transfer media across the bus, even though CAN is not designed for multimedia applications such as a MOST network.

However, the driving force behind the development of an in-vehicle network for a law enforcement platform was not motivated by multimedia applications. The original objective was to determine whether CAN could perform the functions associated with the light control application and reduce wiring across the vehicle environment. The work presented in this thesis, although presented as a proof-of-concept, demonstrates that CAN is a suitable technology to carry out the control of the light package. Furthermore, the inherent two-wire data bus associated with the CAN implementation automatically presents an opportunity to reduce and centralize wiring.

The light package discussed in section 1.3 detailed the 14 different light locations and the fact that in existing cars, each light has a dedicated cable installation from the trunk of the car to the light position. Considering that the overall length of the Troopcar is roughly 17.5 feet, it is estimated that there is about 100 feet of cable installed to support the lights in the current Troopcar. Furthermore, each cable is a two-wire cable. Factoring this into the initial estimate results in a conservative approximation of over 200 feet of individual wires installed to operate the light package. In addition to the potential of adding to fuel consumption, power demands, and decreasing vehicle space, the compilation long wire causes the NJSP to incur installation costs that could be minimized with the implementation of CAN nodes.

As an alternative to the traditional point-to-point installation scheme, the creation of a CAN bus within the vehicle would consist of a twisted-pair of wires running the length of the car to facilitate data flow. Power would be obtained from nearby sources to minimize the amount of additional lengths of power wiring. Therefore, the backbone of the CAN network would consist of a data cable, a twisted-pair of wires, and power

wiring. Additionally, each CAN node installed in the car would have to connect to the CAN bus. In view of the fact that the CAN bus is centralized the wire required to connect the CAN nodes to the bus is minimum. With an efficient layout that minimizes the amount of CAN nodes implemented and maximizes the amount of lights controlled it is possible to reduce the amount of installed wire by at least half. The CAN nodes are designed to be able to control more than just one light and hence each light in the light package does not require an individual node. Instead, the car can be viewed as three sections; front, middle, and rear, with a CAN node controlling each section as demonstrated in Figure 5.1.

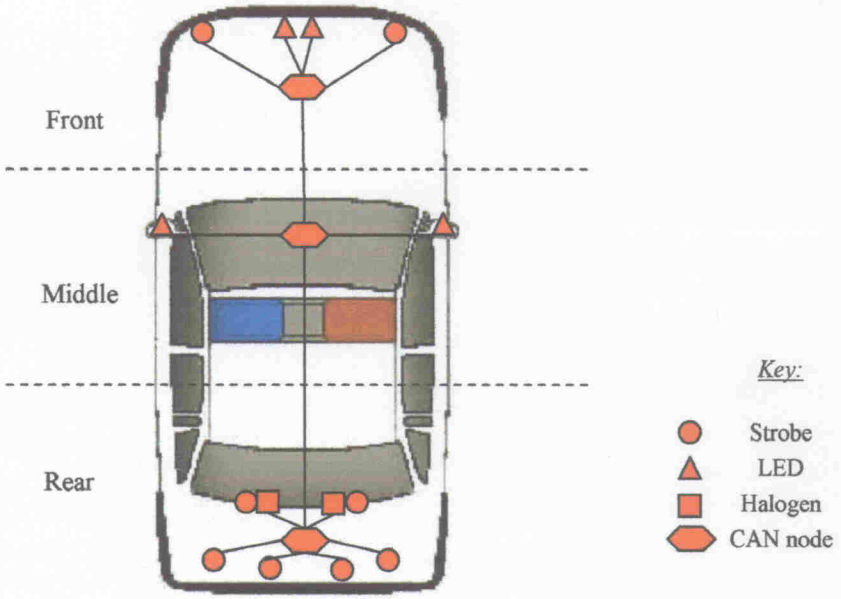

Figure 5.1 – Sectionalized design for CAN node locations within the NJSP troop car.

For a relatively low cost, a CAN implementation of the light package offers economic advantages over the current installation method. In addition to the impending savings in installation costs, CAN provides greater flexibility and versatility over point-

to-point wiring. Not only does the CAN implementation reduce wiring but it provides a communication channel that can facilitate other technologies and systems in the future. In short, the benefits of a CAN implementation outweigh any shortcomings.

5.2 Recommendations for Future Work

The proof-of-concept methodology demonstrated that the CAN protocol is suitable for controlling the NJSP light package. This approach lays the groundwork for achieving a fully integrated vehicle with CAN communications. However, some questions remain unanswered, and there are other potential opportunities to explore.

One area of concern is to ensure that the CAN node design is ruggedized sufficiently to withstand the environment encountered in a typical law enforcement vehicle. The following sections outline future areas of research that extend the foundation of the current research.

5.2.1 Durability of CAN

An in-vehicle network should be able to outlast the useful life of the vehicle it is installed in. Aside from new substitutable technologies becoming available and protocol modifications to CAN, which are sustainability issues, the network should be able to endure the rigors of minor accidents and inclement environments. It is more than likely that law enforcement agencies have life cycle plans that abbreviate the life span of their vehicles, but nonetheless the CAN network should be durable enough to survive the life spans encountered.

To achieve this durability, future considerations should focus on the design and testing of packaging for the CAN nodes. Without proper packaging of a CAN node,

whether it is FPGA based, COTS, or custom silicon, the network could be damaged during an accident or other event, rendering the applications operating on the network useless.

5.2.2 Integrity of CAN

In an automotive environment that is surrounded with omnipresent electromagnetic interference (EMI) from the ignition system to all the in-vehicle electronics, it is essential that data transmission remains immune to interference. EMI effects on data transmission can cause bit errors, and may cause CAN nodes to become inactive if they are excessive. In the event that multiple nodes succumb to EMI, the CAN network can crash and again render applications functioning on the network inoperative. This is of critical importance in a law enforcement vehicle, where there are numerous essential applications that must work at a moments notice in the harshest conditions. When an emergency call is received, the officer must respond. Therefore, the effects of EMI disturbances must be minimized so that the officer can activate the pursuit lights and other functions, which use the CAN network.

Error mechanisms that were discussed in section 2.2.1.3 were almost a direct result of trying to adapt a system around the potential of error due to EMI. In addition, implementing a twisted pair, two-wire bus with differential signaling increases immunity to EMI and also continues communications even if one of the lines breaks or is short-circuited. Although, the protocol is standardized and others have tested it to prove that it provides greater immunity to EMI, this has not been shown in this thesis.

Future efforts to analyze the integrity of CAN in a law enforcement environment should examine the EMI immunity of the network to other in-vehicle electronics, such as

two-way radios, on board computers, etc. In addition to susceptibility to EMI, the amount of EMI produced by the network should also be evaluated.

Data integrity can also be compromised when the data bus is damaged. Therefore, an assessment of the two-wire CAN bus with various faults needs to be completed.

5.2.3 Compatibility of CAN

A major factor that resulted in the CAN research discussed in this thesis was the fact that there were numerous in-vehicle electronics in the NJSP troop car that resulted in excessive wiring but also were incompatible. In its current form the NJSP troop car has multiple systems that are add-ins and therefore they have not been designed to share information along a common medium. This incompatibility can potentially be solved by implementing a common network such as CAN that has been proposed throughout this thesis.

Establishing an in-vehicle network, such as CAN, provides a common information channel that can be accessed by numerous systems. Therefore, it is conceptually feasible to bridge two separate systems that were once completely independent of each other, such as the light package and the mobile computer, so as to allow sharing information with each other. With CAN compatibility, the light package can be controlled through the PC and diagnostic information, for instance if a light was malfunctioning or burned out, can be sent from the lights back to the PC supplying valuable information for the user.

Further exploration dedicated to providing CAN capability and compatibility for additional law enforcement applications should be considered. A future goal of this research should focus on the development of a fully CAN-compatible law enforcement

vehicle that provides increased functionality, simultaneously creating an environment that is more attuned with the user.

Another compatibility issue that may be questioned is the CAN formatting. Concerns involving compatibility between the standard format and extended format are resolved by the identifier extension (IDE) bit in the arbitration field that distinguishes between standard and extended CAN frames. No future work involving formatting is anticipated to be needed.

5.2.4 Sustainability of CAN

All three aspects discussed so far, durability, integrity, and compatibility, play a part in sustainable design. Sustainable design can be defined as a design that provides economic, social, and environmental benefits in the long term. For an in-vehicle network to endure the life span of a vehicle it needs to be sustainable. Without characteristics as durability, integrity, and compatibility accounted for there is no design, regardless if it is an in-vehicle network or not, that can sustain a long lifetime without being replaced. Eliminating the need to continually replace or update a given piece of technology decreases waste and decreases the amount of capital and resources spent on purchasing and installing new technology, thus providing economic and environmental benefits.

An obvious advantage that FPGA design affords is the fact that the logic components can be reconfigured. Therefore, if there were ever any modifications to the CAN protocol or a need to adjust the current design, the FPGA could be reconfigured without having to replace it. Nevertheless, there are environmental concerns with any printed circuit board (PCB) design. From the process of manufacturing to the time the PCB is discarded there are numerous pollutants and wastes generated. These wastes

include but are not limited to CFCs, acid and alkaline solutions, flux residue, scrap board metal, and solder [47]. However, this is a problem faced by the entire PCB industry and is being solved collectively. With the use of a FPGA, these effects can be minimized by reducing the number of PCBs needed.

Another issue related to sustainability is the aspect of cost. The design should be at a cost point where it gains acceptance and also provides a profit for the designer. The dominant choices for implementing CAN are using FPGAs or ASICs. FPGAs allow for post fabrication modification, whereas custom silicon (ASICs) does not. However, the use of an ASIC allows for more efficient use of gates. Using only 40k of a 100k-gate FPGA can be considered inefficient. Instead, an ASIC designer can develop a custom IC that implements the CAN protocol using just the number of gates needed. This reduces the footprint size of the design and saves money. If the price per gate is assumed to be the same for FPGA and ASIC designs, then ASIC designs will prove to be more economical. However, price per gate is not the same and therefore there are many tradeoffs involved with FPGA design versus ASIC design. One tradeoff is whether or not it is more economical to mass produce a design in a FPGA, which can be reconfigured and result in future savings, or produce the design with an ASIC, which cannot be reconfigured and will result in further overhead in the future. There is also an issue of fixed costs such as nonrecurring-engineering (NRE) that results in higher initial design cost for ASIC design while FPGA design has a low initial design cost. The end result is that there exists a break-even point between the unit costs for ASIC and FPGA technology that needs to be determined to decide when it is more cost-effective to implement either technology. It

also must be considered that as fabrication technology continues to advance, the price per gate will become cheaper and efficiency of gate usage may become an issue of the past.

Future research in the sustainable design of CAN devices should focus on these environmental and cost issues. For CAN to endure the life span of a vehicle it should be cost effective to implement, able to adapt to future modifications, and not pose a threat to the environment during manufacturing, throughout its life and until the time it is disposed of.

References

1. G. Leen, D. Heffernan, "Expanding Automotive Electronic Systems," *IEEE Computer,* Vol. 35, No. 1, pp. 88 -93, 2002.

2. D. Holt, "Wiring Down," SAE Service Tech, Sept. 2002.

3. Intel, "Introduction to In-Vehicle Networking," *http://www.intel.com/design/auto/ autolxbk.htm.*

4. Illinois Environmental Protection Agency, "Understanding On Board Diagnostics (OBD)," *http://www.epa.state.il.us/air/vim/faq/obdlong.html.*

5. J. Oliver, "Implementing the J1850 Protocol," Intel Corporation, *http://www.intel.com/design/intarch/papers/j1850_wp.pdf.*

6. LIN Consortium, LIN Specification, Version 1.3, Dec. 2002.

7. MOST Cooperation, MOST Specification Framework, Version 2.2, Nov. 2002.

8. R. Bosch GmbH, CAN Specification, Version 2, Sept. 1991.

9. G. Leen, D. Heffernan, A. Dunne, "Digital Networks in the Automotive Vehicle," *IEEE Computer and Control Eng. Journal,* Dec. 1999, pp. 257-266.

10. M. Krug, A. Schendl, "New Demands for In-Vehicle Networks," *Euromicro: Proc. of the Euromicro Conference,* Vol. 23, pp. 601-605, 1997.

11. D. Marsh, "CAN Bus Networks Break Into Mainstream Use," EDN Magazine, Aug. 2002.

12. W. Lawrenz, "Worldwide Status of CAN: Present and Future," *Proc. of the 2nd Int. CAN Conf.,* 1995.

13. K.D. Rupp, O. Wurst Putzmeister, "Implementation of CAN System in Truck Based Aircraft Washing System," *Proc. of the 1st Int. CAN Conf.,1994.*

14. J. Dattolo, "Society of Automotive Engineers Implementation of CAN for Heavy Duty Truck and Bus Market, Specification J1939," *Proc. of the 2nd Int. CAN Conf.,* 1995.

15. W. Appel, J. Dorner, "Integration of External Functions in CAN Based In-Vehicle Networks," *Proc. of the 2nd Int. CAN Conf.,* 1995.

16. T. Moon, "Control in Automotive Body System Networks," *Proc. of the 3rd Int. CAN Conf.,* 1996.

17. G. Bourdon, P. Raux, S. Delaplace, "CAN for Autonomous Mobile Robots," *Proc. of the 3rd Int. CAN Conf., 1996.*

18. H. Zeltwanger, "Designing Devices Using CAN and CANopen Buses for Networking," *Medical Electronics Manufacturing,* 1999.

19. S. Crisp, "CAN in Las Vegas' Monorail," *CiA CAN Newsletter*, pp. 12-14, Dec. 2002.

20. M. Bailey, "CAN in Electric Vehicles," *Proc. of the 3rd Int. CAN Conf.,* 1996.

21. L. Fredriksson, "CAN for Critical Embedded Automotive Networks," *IEEE Micro*, Vol. 22, No. 4, pp. 28-35, 2002.

22. J. Ferreira, P. Pedreiras, "The FFT-CAN Protocol for Flexibility in Safety-Critical Systems," *IEEE Micro*, Vol. 22, No. 4, pp. 46-55, 2002.

23. W. Stallings, *Data and Computer Communications*, Prentice Hall, 1997.

24. W. Lawrenz, *CAN System Engineering: From Theory to Practical Applications*, Springer-Verlag New York, 1997.

25. K. Etschberger, *Controller Area Network: Basics, Protocols, Chips and Applications*, IXXAT Automation, 2001.

26. H. Hegering, A. Lapple, *Ethernet: Building a Communications Infrastructure,* Addison-Wesley, 1993.

27. Microchip Technology Inc., "Controller Area Network Basics," *http://www.microchip.com/download/appnote/analog/can/00713a.pdf*.

28. J. Massey, "Step-by-step Decoding of the Bose-Chaudhuri-Hocquenghem Codes," *IEEE Trans. On Information Theory,* Vol. 11, No.4, pp. 580-585, Oct. 1965.

29. J. Charzinski, "Performance of the Error Detection Mechanisms in CAN," *Proc. of the 1st Int. CAN Conf.,* 1994.

30. ISO-CD 11898-1, Road Vehicles - Controller Area Network (CAN) - Part 1: Data Link Layer and Physical Signaling, 1999.

31. ISO-CD 11898-2, Road Vehicles - Controller Area Network (CAN) - Part 2: High Speed Medium Access Unit, 1999.

32. F. Hartwich, A. Bassemir, "The Configuration of the CAN Bit Timing," *Proc. of the 6th Int. CAN Conf.,* 1999.

33. J. Irvine, D. Harle, *Data Communications and Networks: An Engineering Approach.* John Wiley and Sons, 2002.

34. T. Nolte, H. Hansson, C. Norstrom, S. Punnekkat, "Using Bit Stuffing Distributions in CAN Analysis," *IEEE Real-Time Embedded Systems Workshop,* Dec. 2001.

35. CAN-in-Automation, CAN Application Layer for Industrial Application, Version 1.1, CiA DS 201-207, 1996.

36. CAN-in-Automation, CANopen Application Layer and Communication Profile, Version 4.02, CiA DS 301, 2002

37. Open DeviceNet Vendor Association Inc., DeviceNet Specifications, Version 2.0, Vol. I and II, 1998.

38. OPENCORES.org, *http://www.opencores.org.*

39. Philips Semiconductors, SJA1000 Stand-alone CAN Controller Product Specification, Jan. 2000.

40. J. Bhasker, *A VHDL Primer,* Prentice Hall, 1999.

41. U. Heinkel, M. Padeffke, W. Haas, T. Buerner, H. Braisz, T. Genter, and A. Grassmand, *The VHDL Reference: A Practical Guide to Computer-Aided Integrated Circuit Design Including VHDL-AMS*, John Wiley and Sons, 2000.

42. L. Lemus, J. Gracia, P. Gil, "Designing, Modeling and Implementation of a Controller Area Network (CAN) on a FPGA using VHDL," *Second International Forum on Design Languages (FDL 99)*, Sept. 1999.

43. Digilent Incorporated, Digilab 2 Reference Manual, *http://www.digilentinc.com/assets/documents/d2_rm.pdf.*

44. Philips Semiconductors, PCA82C250 CAN Controller Interface Product Specification, Jan. 2000.

45. Rabbit Semiconductor, Rabbit 2000 Microprocessor User's Manual,

46. Philips Semiconductors, SJA1000 Stand-alone CAN Controller Application Note, Dec. 1997.

47. CorpWatch, The Environmental Cost of Printed Circuit Boards, *http://www.corpwatch.org/issues/PID.jsp?articleid=3433.*

Appendix A: HDL Code

The following appendix contains all of the HDL code components used to construct the CAN controller. This code was written by Igor Mohor and provided through OPENCORES.org. The code was synthesized, simulated, and implemented using Xilinx Foundation 4.2i and ModelSim Xilinx Edition II version 5.6e.

A.1: Top Level HDL Code

The following code is the top layer of the CAN controller that connects the underlying subcomponents together.

```
// can_top.v
// synopsys translate_off
//`include "timescale.v"
`timescale 1ns/10ps
// synopsys translate_on
//`include "can_defines.v"

// Uncomment following line if you want to use WISHBONE interface.
Otherwise
// 8051 interface is used.
//`define   CAN_WISHBONE_IF
//`define Intel_mode

// Uncomment following line if you want to use CAN in Actel APA devices
(embedded memory used)
//define   ACTEL_APA_RAM

// Uncomment following line if you want to use CAN in Xilinx devices
(embedded memory used)
`define   XILINX_RAM

module can_top
(
  `ifdef CAN_WISHBONE_IF
    wb_clk_i,
    wb_rst_i,
    wb_dat_i,
    wb_dat_o,
    wb_cyc_i,
    wb_stb_i,
    wb_we_i,
    wb_adr_i,
```

```
    wb_ack_o,
  `else
    rst_i,
    ale_i,
    rd_i,
    wr_i,
    port_0_io,
  `endif
  cs_can_i,
  clk_i,
  rx_i,
  tx_o,
  irq_on,
  clkout_o
);

parameter Tp = 1;

`ifdef CAN_WISHBONE_IF
  input        wb_clk_i;
  input        wb_rst_i;
  input  [7:0] wb_dat_i;
  output [7:0] wb_dat_o;
  input        wb_cyc_i;
  input        wb_stb_i;
  input        wb_we_i;
  input  [7:0] wb_adr_i;
  output       wb_ack_o;

  reg          wb_ack_o;
  reg          cs_sync1;
  reg          cs_sync2;
  reg          cs_sync3;

  reg          cs_ack1;
  reg          cs_ack2;
  reg          cs_ack3;
  reg          cs_sync_rst1;
  reg          cs_sync_rst2;
`else
  input        rst_i;
  input        ale_i;
  input        rd_i;
  input        wr_i;
  inout  [7:0] port_0_io;

  reg    [7:0] addr_latched;
  reg          wr_i_q;
  reg          rd_i_q;
`endif

input        cs_can_i;
input        clk_i;
input        rx_i;
output       tx_o;
output       irq_on;
output       clkout_o;
```

```
reg          data_out_fifo_selected;

wire         irq_o;
wire   [7:0] data_out_fifo;
wire   [7:0] data_out_regs;

/* Mode register */
wire         reset_mode;
wire         listen_only_mode;
wire         acceptance_filter_mode;
wire         self_test_mode;

/* Command register */
wire         release_buffer;
wire         tx_request;
wire         abort_tx;
wire         self_rx_request;
wire         single_shot_transmission;

/* Arbitration Lost Capture Register */
wire         read_arbitration_lost_capture_reg;

/* Error Code Capture Register */
wire         read_error_code_capture_reg;
wire   [7:0] error_capture_code;

/* Bus Timing 0 register */
wire   [5:0] baud_r_presc;
wire   [1:0] sync_jump_width;

/* Bus Timing 1 register */
wire   [3:0] time_segment1;
wire   [2:0] time_segment2;
wire         triple_sampling;

/* Error Warning Limit register */
wire   [7:0] error_warning_limit;

/* Rx Error Counter register */
wire         we_rx_err_cnt;

/* Tx Error Counter register */
wire         we_tx_err_cnt;

/* Clock Divider register */
wire         extended_mode;

/* This section is for BASIC and EXTENDED mode */
/* Acceptance code register */
wire   [7:0] acceptance_code_0;

/* Acceptance mask register */
wire   [7:0] acceptance_mask_0;
/* End: This section is for BASIC and EXTENDED mode */
```

```
/* This section is for EXTENDED mode */
/* Acceptance code register */
wire    [7:0] acceptance_code_1;
wire    [7:0] acceptance_code_2;
wire    [7:0] acceptance_code_3;

/* Acceptance mask register */
wire    [7:0] acceptance_mask_1;
wire    [7:0] acceptance_mask_2;
wire    [7:0] acceptance_mask_3;
/* End: This section is for EXTENDED mode */

/* Tx data registers. Holding identifier (basic mode), tx frame
information (extended mode) and data */
wire    [7:0] tx_data_0;
wire    [7:0] tx_data_1;
wire    [7:0] tx_data_2;
wire    [7:0] tx_data_3;
wire    [7:0] tx_data_4;
wire    [7:0] tx_data_5;
wire    [7:0] tx_data_6;
wire    [7:0] tx_data_7;
wire    [7:0] tx_data_8;
wire    [7:0] tx_data_9;
wire    [7:0] tx_data_10;
wire    [7:0] tx_data_11;
wire    [7:0] tx_data_12;
/* End: Tx data registers */

wire          cs;

/* Output signals from can_btl module */
wire          clk_en;
wire          sample_point;
wire          sampled_bit;
wire          sampled_bit_q;
wire          tx_point;
wire          hard_sync;
wire          resync;


/* output from can_bsp module */
wire          rx_idle;
wire          transmitting;
wire          last_bit_of_inter;
wire          set_reset_mode;
wire          node_bus_off;
wire          error_status;
wire    [7:0] rx_err_cnt;
wire    [7:0] tx_err_cnt;
wire          rx_err_cnt_dummy;  // The MSB is not displayed. It is just
used for easier calculation (no counter overflow).
wire          tx_err_cnt_dummy;  // The MSB is not displayed. It is just
used for easier calculation (no counter overflow).
wire          transmit_status;
wire          receive_status;
wire          tx_successful;
```

```
wire          need_to_tx;
wire          overrun;
wire          info_empty;
wire          set_bus_error_irq;
wire          set_arbitration_lost_irq;
wire    [4:0] arbitration_lost_capture;
wire          node_error_passive;
wire          node_error_active;
wire    [6:0] rx_message_counter;
wire          tx_out;
wire          tx_oen;
wire          rst;
wire          we;
wire    [7:0] addr;
wire    [7:0] data_in;
reg     [7:0] data_out;


/* Connecting can_registers module */
can_registers i_can_registers
(
  .clk(clk_i),
  .rst(rst),
  .cs(cs),
  .we(we),
  .addr(addr),
  .data_in(data_in),
  .data_out(data_out_regs),
  .irq(irq_o),

  .sample_point(sample_point),
  .transmitting(transmitting),
  .set_reset_mode(set_reset_mode),
  .node_bus_off(node_bus_off),
  .error_status(error_status),
  .rx_err_cnt(rx_err_cnt),
  .tx_err_cnt(tx_err_cnt),
  .transmit_status(transmit_status),
  .receive_status(receive_status),
  .tx_successful(tx_successful),
  .need_to_tx(need_to_tx),
  .overrun(overrun),
  .info_empty(info_empty),
  .set_bus_error_irq(set_bus_error_irq),
  .set_arbitration_lost_irq(set_arbitration_lost_irq),
  .arbitration_lost_capture(arbitration_lost_capture),
  .node_error_passive(node_error_passive),
  .node_error_active(node_error_active),
  .rx_message_counter(rx_message_counter),


  /* Mode register */
  .reset_mode(reset_mode),
  .listen_only_mode(listen_only_mode),
  .acceptance_filter_mode(acceptance_filter_mode),
  .self_test_mode(self_test_mode),
```

```
  /* Command register */
  .clear_data_overrun(),
  .release_buffer(release_buffer),
  .abort_tx(abort_tx),
  .tx_request(tx_request),
  .self_rx_request(self_rx_request),
  .single_shot_transmission(single_shot_transmission),

  /* Arbitration Lost Capture Register */

.read_arbitration_lost_capture_reg(read_arbitration_lost_capture_reg),

  /* Error Code Capture Register */
  .read_error_code_capture_reg(read_error_code_capture_reg),
  .error_capture_code(error_capture_code),

  /* Bus Timing 0 register */
  .baud_r_presc(baud_r_presc),
  .sync_jump_width(sync_jump_width),

  /* Bus Timing 1 register */
  .time_segment1(time_segment1),
  .time_segment2(time_segment2),
  .triple_sampling(triple_sampling),

  /* Error Warning Limit register */
  .error_warning_limit(error_warning_limit),

  /* Rx Error Counter register */
  .we_rx_err_cnt(we_rx_err_cnt),

  /* Tx Error Counter register */
  .we_tx_err_cnt(we_tx_err_cnt),

  /* Clock Divider register */
  .extended_mode(extended_mode),
  .clkout(clkout_o),

  /* This section is for BASIC and EXTENDED mode */
  /* Acceptance code register */
  .acceptance_code_0(acceptance_code_0),

  /* Acceptance mask register */
  .acceptance_mask_0(acceptance_mask_0),
  /* End: This section is for BASIC and EXTENDED mode */

  /* This section is for EXTENDED mode */
  /* Acceptance code register */
  .acceptance_code_1(acceptance_code_1),
  .acceptance_code_2(acceptance_code_2),
  .acceptance_code_3(acceptance_code_3),

  /* Acceptance mask register */
  .acceptance_mask_1(acceptance_mask_1),
  .acceptance_mask_2(acceptance_mask_2),
  .acceptance_mask_3(acceptance_mask_3),
  /* End: This section is for EXTENDED mode */
```

```
  /* Tx data registers. Holding identifier (basic mode), tx frame
information (extended mode) and data */
  .tx_data_0(tx_data_0),
  .tx_data_1(tx_data_1),
  .tx_data_2(tx_data_2),
  .tx_data_3(tx_data_3),
  .tx_data_4(tx_data_4),
  .tx_data_5(tx_data_5),
  .tx_data_6(tx_data_6),
  .tx_data_7(tx_data_7),
  .tx_data_8(tx_data_8),
  .tx_data_9(tx_data_9),
  .tx_data_10(tx_data_10),
  .tx_data_11(tx_data_11),
  .tx_data_12(tx_data_12)
  /* End: Tx data registers */
);

assign irq_on = ~irq_o;

/* Connecting can_btl module */
can_btl i_can_btl
(
  .clk(clk_i),
  .rst(rst),
  .rx(rx_i),

  /* Mode register */
  .reset_mode(reset_mode),

  /* Bus Timing 0 register */
  .baud_r_presc(baud_r_presc),
  .sync_jump_width(sync_jump_width),

  /* Bus Timing 1 register */
  .time_segment1(time_segment1),
  .time_segment2(time_segment2),
  .triple_sampling(triple_sampling),

  /* Output signals from this module */
  .clk_en(clk_en),
  .sample_point(sample_point),
  .sampled_bit(sampled_bit),
  .sampled_bit_q(sampled_bit_q),
  .tx_point(tx_point),
  .hard_sync(hard_sync),
  .resync(resync),

  /* output from can_bsp module */
  .rx_idle(rx_idle),
  .transmitting(transmitting),
  .last_bit_of_inter(last_bit_of_inter)
);

can_bsp i_can_bsp
(
```

```
  .clk(clk_i),
  .rst(rst),

  /* From btl module */
  .sample_point(sample_point),
  .sampled_bit(sampled_bit),
  .sampled_bit_q(sampled_bit_q),
  .tx_point(tx_point),
  .hard_sync(hard_sync),

  .addr(addr),
  .data_in(data_in),
  .data_out(data_out_fifo),
  .fifo_selected(data_out_fifo_selected),

  /* Mode register */
  .reset_mode(reset_mode),
  .listen_only_mode(listen_only_mode),
  .acceptance_filter_mode(acceptance_filter_mode),
  .self_test_mode(self_test_mode),

  /* Command register */
  .release_buffer(release_buffer),
  .tx_request(tx_request),
  .abort_tx(abort_tx),
  .self_rx_request(self_rx_request),
  .single_shot_transmission(single_shot_transmission),

  /* Arbitration Lost Capture Register */

.read_arbitration_lost_capture_reg(read_arbitration_lost_capture_reg),

  /* Error Code Capture Register */
  .read_error_code_capture_reg(read_error_code_capture_reg),
  .error_capture_code(error_capture_code),

  /* Error Warning Limit register */
  .error_warning_limit(error_warning_limit),

  /* Rx Error Counter register */
  .we_rx_err_cnt(we_rx_err_cnt),

  /* Tx Error Counter register */
  .we_tx_err_cnt(we_tx_err_cnt),

  /* Clock Divider register */
  .extended_mode(extended_mode),

  /* output from can_bsp module */
  .rx_idle(rx_idle),
  .transmitting(transmitting),
  .last_bit_of_inter(last_bit_of_inter),
  .set_reset_mode(set_reset_mode),
  .node_bus_off(node_bus_off),
  .error_status(error_status),
  .rx_err_cnt({rx_err_cnt_dummy, rx_err_cnt[7:0]}),   // The MSB is not
displayed. It is just used for easier calculation (no counter
```

```
overflow).
  .tx_err_cnt({tx_err_cnt_dummy, tx_err_cnt[7:0]}),   // The MSB is not
displayed. It is just used for easier calculation (no counter
overflow).
  .transmit_status(transmit_status),
  .receive_status(receive_status),
  .tx_successful(tx_successful),
  .need_to_tx(need_to_tx),
  .overrun(overrun),
  .info_empty(info_empty),
  .set_bus_error_irq(set_bus_error_irq),
  .set_arbitration_lost_irq(set_arbitration_lost_irq),
  .arbitration_lost_capture(arbitration_lost_capture),
  .node_error_passive(node_error_passive),
  .node_error_active(node_error_active),
  .rx_message_counter(rx_message_counter),

  /* This section is for BASIC and EXTENDED mode */
  /* Acceptance code register */
  .acceptance_code_0(acceptance_code_0),

  /* Acceptance mask register */
  .acceptance_mask_0(acceptance_mask_0),
  /* End: This section is for BASIC and EXTENDED mode */

  /* This section is for EXTENDED mode */
  /* Acceptance code register */
  .acceptance_code_1(acceptance_code_1),
  .acceptance_code_2(acceptance_code_2),
  .acceptance_code_3(acceptance_code_3),

  /* Acceptance mask register */
  .acceptance_mask_1(acceptance_mask_1),
  .acceptance_mask_2(acceptance_mask_2),
  .acceptance_mask_3(acceptance_mask_3),
  /* End: This section is for EXTENDED mode */

  /* Tx data registers. Holding identifier (basic mode), tx frame
information (extended mode) and data */
  .tx_data_0(tx_data_0),
  .tx_data_1(tx_data_1),
  .tx_data_2(tx_data_2),
  .tx_data_3(tx_data_3),
  .tx_data_4(tx_data_4),
  .tx_data_5(tx_data_5),
  .tx_data_6(tx_data_6),
  .tx_data_7(tx_data_7),
  .tx_data_8(tx_data_8),
  .tx_data_9(tx_data_9),
  .tx_data_10(tx_data_10),
  .tx_data_11(tx_data_11),
  .tx_data_12(tx_data_12),
  /* End: Tx data registers */

  /* Tx signal */
  .tx(tx_out),
  .tx_oen(tx_oen)
```

```
);

assign tx_o = tx_oen? 1'bz : tx_out;

// Multiplexing wb_dat_o from registers and rx fifo
always @ (extended_mode or addr or reset_mode)
begin
  if (extended_mode & (~reset_mode) & ((addr >= 8'd16) && (addr <=
8'd28)) | (~extended_mode) & ((addr >= 8'd20) && (addr <= 8'd29)))
    data_out_fifo_selected <= 1'b1;
  else
    data_out_fifo_selected <= 1'b0;
end

always @ (posedge clk_i)
begin
//  if (wb_cyc_i & (~wb_we_i))
  if (cs & (~we))
    begin
      if (data_out_fifo_selected)
        data_out <=#Tp data_out_fifo;
      else
        data_out <=#Tp data_out_regs;
    end
end

`ifdef CAN_WISHBONE_IF
  // Combining wb_cyc_i and wb_stb_i signals to cs signal. Than
synchronizing to clk_i clock domain.
  always @ (posedge clk_i or posedge rst)
  begin
    if (rst)
      begin
        cs_sync1     <= 1'b0;
        cs_sync2     <= 1'b0;
        cs_sync3     <= 1'b0;
        cs_sync_rst1 <= 1'b0;
        cs_sync_rst2 <= 1'b0;
      end
    else
      begin
        cs_sync1<=#Tp wb_cyc_i & wb_stb_i & (~cs_sync_rst2) & cs_can_i;
        cs_sync2     <=#Tp cs_sync1            & (~cs_sync_rst2);
        cs_sync3     <=#Tp cs_sync2            & (~cs_sync_rst2);
        cs_sync_rst1 <=#Tp cs_ack3;
        cs_sync_rst2 <=#Tp cs_sync_rst1;
      end
  end

  assign cs = cs_sync2 & (~cs_sync3);

  always @ (posedge wb_clk_i)
  begin
    cs_ack1 <=#Tp cs_sync3;
    cs_ack2 <=#Tp cs_ack1;
    cs_ack3 <=#Tp cs_ack2;
  end
```

```
  // Generating acknowledge signal
  always @ (posedge wb_clk_i)
  begin
    wb_ack_o <=#Tp (cs_ack2 & (~cs_ack3));
  end

  assign rst      = wb_rst_i;
  assign we       = wb_we_i;
  assign addr     = wb_adr_i;
  assign data_in  = wb_dat_i;
  assign wb_dat_o = data_out;

`else

  // Latching address
  always @ (negedge clk_i or posedge rst)
  begin
    if (rst)
      addr_latched <= 8'h0;
    else if (ale_i)
      addr_latched <=#Tp port_0_io;
  end

  // Generating delayed wr_i and rd_i signals
  always @ (posedge clk_i or posedge rst)
  begin
    if (rst)
      begin
        wr_i_q <= 1'b0;
        rd_i_q <= 1'b0;
      end
    else
      begin
        wr_i_q <=#Tp wr_i;
        rd_i_q <=#Tp rd_i;
      end
  end

  assign cs = ((wr_i & (~wr_i_q)) | (rd_i & (~rd_i_q))) & cs_can_i;

  assign rst       = rst_i;
  assign we        = wr_i;
  assign addr      = addr_latched;
  assign data_in   = port_0_io;
  assign port_0_io = (cs_can_i & rd_i)? data_out : 8'hz;

`endif
endmodule
```

A.2: BTL HDL Code

This code is the bit timing logic (BTL) component of the CAN controller. The BTL continually monitors the input and output data streams and handle the related bit timing according to the CAN protocol.

```
// can_btl.v
// synopsys translate_off
//`include "timescale.v"
`timescale 1ns/10ps
// synopsys translate_on
//`include "can_defines.v"

// Uncomment following line if you want to use WISHBONE interface.
Otherwise
// 8051 interface is used.
//`define   CAN_WISHBONE_IF
//`define Intel_mode

// Uncomment following line if you want to use CAN in Actel APA devices
(embedded memory used)
//define   ACTEL_APA_RAM

// Uncomment following line if you want to use CAN in Xilinx devices
(embedded memory used)
`define   XILINX_RAM

module can_btl
(
  clk,
  rst,
  rx,

  /* Mode register */
  reset_mode,

  /* Bus Timing 0 register */
  baud_r_presc,
  sync_jump_width,

  /* Bus Timing 1 register */
  time_segment1,
  time_segment2,
  triple_sampling,

  /* Output signals from this module */
  clk_en,
  sample_point,
  sampled_bit,
  sampled_bit_q,
  tx_point,
```

```
  hard_sync,
  resync,
    /* Output from can_bsp module */
  rx_idle,
  transmitting,
  last_bit_of_inter
);

parameter Tp = 1;

input         clk;
input         rst;
input         rx;

  /* Mode register */
input         reset_mode;

/* Bus Timing 0 register */
input   [5:0] baud_r_presc;
input   [1:0] sync_jump_width;

/* Bus Timing 1 register */
input   [3:0] time_segment1;
input   [2:0] time_segment2;
input         triple_sampling;

/* Output from can_bsp module */
input         rx_idle;
input         transmitting;
input         last_bit_of_inter;

/* Output signals from this module */
output        clk_en;
output        sample_point;
output        sampled_bit;
output        sampled_bit_q;
output        tx_point;
output        hard_sync;
output        resync;

reg     [8:0] clk_cnt;
reg           clk_en;
reg           sync_blocked;
reg           resync_blocked;
reg           sampled_bit;
reg           sampled_bit_q;
reg     [7:0] quant_cnt;
reg     [3:0] delay;
reg           sync;
reg           seg1;
reg           seg2;
reg           resync_latched;
reg           sample_point;
reg     [1:0] sample;

wire          go_sync;
wire          go_seg1;
```

```
wire          go_seg2;
wire [8:0]    preset_cnt;
wire          sync_window;

assign preset_cnt = (baud_r_presc + 1'b1)<<1;        // (BRP+1)*2
assign hard_sync  =   (rx_idle | last_bit_of_inter)  & (~rx) &
sampled_bit & (~sync_blocked) & (~transmitting);  // Hard
synchronization
assign resync     =  (~rx_idle)                      & (~rx) &
sampled_bit & (~sync_blocked) & (~resync_blocked) & (~transmitting);
// Re-synchronization

/* Generating general enable signal that defines baud rate. */
always @ (posedge clk or posedge rst)
begin
  if (rst)
    clk_cnt <= 0;
  else if (clk_cnt == (preset_cnt-1))
    clk_cnt <=#Tp 0;
  else
    clk_cnt <=#Tp clk_cnt + 1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    clk_en  <= 1'b0;
  else if (clk_cnt == (preset_cnt-1))
    clk_en  <=#Tp 1'b1;
  else
    clk_en  <=#Tp 1'b0;
end

/* Changing states */
 assign go_sync = clk_en & (seg2 & (~hard_sync) & (~resync) &
((quant_cnt == time_segment2)));
 assign go_seg1 = clk_en & (sync | hard_sync | (resync & seg2 &
sync_window) | (resync_latched & sync_window));
 assign go_seg2 = clk_en & (seg1 & (~hard_sync) & (quant_cnt ==
(time_segment1 + delay)));

/* When early edge is detected outside of the SJW field,
synchronization request is latched and performed when
   SJW is reached */
always @ (posedge clk or posedge rst)
begin
  if (rst)
    resync_latched <= 1'b0;
  else if (resync & seg2 & (~sync_window))
    resync_latched <=#Tp 1'b1;
  else if (go_seg1)
    resync_latched <= 1'b0;
end

/* Synchronization stage/segment */
always @ (posedge clk or posedge rst)
begin
```

```
  if (rst)
    sync <= 0;
  else if (go_sync)
    sync <=#Tp 1'b1;
  else if (go_seg1)
    sync <=#Tp 1'b0;
end

assign tx_point = go_sync;

/* Seg1 stage/segment (together with propagation segment which is 1
quant long) */
always @ (posedge clk or posedge rst)
begin
  if (rst)
    seg1 <= 1;
  else if (go_seg1)
    seg1 <=#Tp 1'b1;
  else if (go_seg2)
    seg1 <=#Tp 1'b0;
end

/* Seg2 stage/segment */
always @ (posedge clk or posedge rst)
begin
  if (rst)
    seg2 <= 0;
  else if (go_seg2)
    seg2 <=#Tp 1'b1;
  else if (go_sync | go_seg1)
    seg2 <=#Tp 1'b0;
end

/* Quant counter */
always @ (posedge clk or posedge rst)
begin
  if (rst)
    quant_cnt <= 0;
  else if (go_sync | go_seg1 | go_seg2)
    quant_cnt <=#Tp 0;
  else if (clk_en)
    quant_cnt <=#Tp quant_cnt + 1'b1;
end

/* When late edge is detected (in seg1 stage), stage seg1 is prolonged.
*/
always @ (posedge clk or posedge rst)
begin
  if (rst)
    delay <= 0;
  else if (clk_en & resync & seg1)
    delay <=#Tp (quant_cnt > sync_jump_width)? (sync_jump_width + 1) :
(quant_cnt + 1);
  else if (go_sync | go_seg1)
    delay <=#Tp 0;
end
```

```
// If early edge appears within this window (in seg2 stage), phase
error is fully compensated
assign sync_window = ((time_segment2 - quant_cnt) < ( sync_jump_width +
1));

// Sampling data (memorizing two samples all the time).
always @ (posedge clk or posedge rst)
begin
  if (rst)
    sample <= 2'b11;
  else if (clk_en)
    sample <= {sample[0], rx};
end

// When enabled, tripple sampling is done here.
always @ (posedge clk or posedge rst)
begin
  if (rst)
    begin
      sampled_bit <= 1;
      sampled_bit_q <= 1;
      sample_point <= 0;
    end
  else if (clk_en & (~hard_sync))
    begin
      if (seg1 & (quant_cnt == (time_segment1 + delay)))
        begin
          sample_point <=#Tp 1;
          sampled_bit_q <=#Tp sampled_bit;
          if (triple_sampling)
            sampled_bit <=#Tp (sample[0] & sample[1]) | ( sample[0] &
rx) | (sample[1] & rx);
          else
            sampled_bit <=#Tp rx;
        end
    end
  else
    sample_point <=#Tp 0;
end

/* Blocking synchronization (can occur only once in a bit time) */

always @ (posedge clk or posedge rst)
begin
  if (rst)
    sync_blocked <=#Tp 1'b0;
  else if (clk_en)
    begin
      if (hard_sync | resync)
        sync_blocked <=#Tp 1'b1;
      else if (seg2 & quant_cnt == time_segment2)
        sync_blocked <=#Tp 1'b0;
    end
end

/* Blocking resynchronization until reception starts (needed because
after reset mode exits we are waiting for
```

```
   end-of-frame and interframe. No resynchronization is needed
meanwhile). */
always @ (posedge clk or posedge rst)
begin
  if (rst)
    resync_blocked <=#Tp 1'b1;
  else if (reset_mode)
    resync_blocked <=#Tp 1'b1;
  else if (hard_sync)
    resync_blocked <=#Tp 1'b0;
end

endmodule
```

A.3: BSP HDL Code

The following code is the bit stream processor (BSP). The function of the BSP is to translate messages into frames and frames into messages. Other tasks performed by the BSP are bit stuffing/ destuffing, CRC checking, and bus arbitration.

```
// can_bsp.v
// synopsys translate_off
//`include "timescale.v"
`timescale 1ns/10ps
// synopsys translate_on
//`include "can_defines.v"

// Uncomment following line if you want to use WISHBONE interface.
Otherwise
// 8051 interface is used.
//`define   CAN_WISHBONE_IF
//`define Intel_mode

// Uncomment following line if you want to use CAN in Actel APA devices
(embedded memory used)
//define   ACTEL_APA_RAM

// Uncomment following line if you want to use CAN in Xilinx devices
(embedded memory used)
`define   XILINX_RAM

module can_bsp
(
  clk,
  rst,
  sample_point,
  sampled_bit,
  sampled_bit_q,
  tx_point,
  hard_sync,
  addr,
  data_in,
  data_out,
  fifo_selected,

  /* Mode register */
  reset_mode,
  listen_only_mode,
  acceptance_filter_mode,
  self_test_mode,

  /* Command register */
  release_buffer,
  tx_request,
  abort_tx,
```

```
self_rx_request,
single_shot_transmission,
/* Arbitration Lost Capture Register */
read_arbitration_lost_capture_reg,

/* Error Code Capture Register */
read_error_code_capture_reg,
error_capture_code,

/* Error Warning Limit register */
error_warning_limit,

/* Rx Error Counter register */
we_rx_err_cnt,

/* Tx Error Counter register */
we_tx_err_cnt,

/* Clock Divider register */
extended_mode,

rx_idle,
transmitting,
last_bit_of_inter,
set_reset_mode,
node_bus_off,
error_status,
rx_err_cnt,
tx_err_cnt,
transmit_status,
receive_status,
tx_successful,
need_to_tx,
overrun,
info_empty,
set_bus_error_irq,
set_arbitration_lost_irq,
arbitration_lost_capture,
node_error_passive,
node_error_active,
rx_message_counter,

/* This section is for BASIC and EXTENDED mode */
/* Acceptance code register */
acceptance_code_0,

/* Acceptance mask register */
acceptance_mask_0,
/* End: This section is for BASIC and EXTENDED mode */

/* This section is for EXTENDED mode */
/* Acceptance code register */
acceptance_code_1,
acceptance_code_2,
acceptance_code_3,

/* Acceptance mask register */
```

```
  acceptance_mask_1,
  acceptance_mask_2,
  acceptance_mask_3,
  /* End: This section is for EXTENDED mode */

  /* Tx data registers. Holding identifier (basic mode), tx frame
information (extended mode) and data */
  tx_data_0,
  tx_data_1,
  tx_data_2,
  tx_data_3,
  tx_data_4,
  tx_data_5,
  tx_data_6,
  tx_data_7,
  tx_data_8,
  tx_data_9,
  tx_data_10,
  tx_data_11,
  tx_data_12,
  /* End: Tx data registers */

  /* Tx signal */
  tx,
  tx_oen
);

parameter Tp = 1;

input         clk;
input         rst;
input         sample_point;
input         sampled_bit;
input         sampled_bit_q;
input         tx_point;
input         hard_sync;
input   [7:0] addr;
input   [7:0] data_in;
output  [7:0] data_out;
input         fifo_selected;

input         reset_mode;
input         listen_only_mode;
input         acceptance_filter_mode;
input         extended_mode;
input         self_test_mode;

/* Command register */
input         release_buffer;
input         tx_request;
input         abort_tx;
input         self_rx_request;
input         single_shot_transmission;

/* Arbitration Lost Capture Register */
input         read_arbitration_lost_capture_reg;
```

```
/* Error Code Capture Register */
input         read_error_code_capture_reg;
output  [7:0] error_capture_code;

/* Error Warning Limit register */
input   [7:0] error_warning_limit;

/* Rx Error Counter register */
input         we_rx_err_cnt;

/* Tx Error Counter register */
input         we_tx_err_cnt;

output        rx_idle;
output        transmitting;
output        last_bit_of_inter;
output        set_reset_mode;
output        node_bus_off;
output        error_status;
output  [8:0] rx_err_cnt;
output  [8:0] tx_err_cnt;
output        transmit_status;
output        receive_status;
output        tx_successful;
output        need_to_tx;
output        overrun;
output        info_empty;
output        set_bus_error_irq;
output        set_arbitration_lost_irq;
output  [4:0] arbitration_lost_capture;
output        node_error_passive;
output        node_error_active;
output  [6:0] rx_message_counter;


/* This section is for BASIC and EXTENDED mode */
/* Acceptance code register */
input   [7:0] acceptance_code_0;

/* Acceptance mask register */
input   [7:0] acceptance_mask_0;

/* End: This section is for BASIC and EXTENDED mode */

/* This section is for EXTENDED mode */
/* Acceptance code register */
input   [7:0] acceptance_code_1;
input   [7:0] acceptance_code_2;
input   [7:0] acceptance_code_3;

/* Acceptance mask register */
input   [7:0] acceptance_mask_1;
input   [7:0] acceptance_mask_2;
input   [7:0] acceptance_mask_3;
/* End: This section is for EXTENDED mode */

/* Tx data registers. Holding identifier (basic mode), tx frame
```

```
information (extended mode) and data */
input    [7:0] tx_data_0;
input    [7:0] tx_data_1;
input    [7:0] tx_data_2;
input    [7:0] tx_data_3;
input    [7:0] tx_data_4;
input    [7:0] tx_data_5;
input    [7:0] tx_data_6;
input    [7:0] tx_data_7;
input    [7:0] tx_data_8;
input    [7:0] tx_data_9;
input    [7:0] tx_data_10;
input    [7:0] tx_data_11;
input    [7:0] tx_data_12;
/* End: Tx data registers */

/* Tx signal */
output         tx;
output         tx_oen;

reg            reset_mode_q;
reg      [5:0] bit_cnt;

reg      [3:0] data_len;
reg     [28:0] id;
reg      [2:0] bit_stuff_cnt;
reg      [2:0] bit_stuff_cnt_tx;
reg            tx_point_q;

reg            rx_idle;
reg            rx_id1;
reg            rx_rtr1;
reg            rx_ide;
reg            rx_id2;
reg            rx_rtr2;
reg            rx_r1;
reg            rx_r0;
reg            rx_dlc;
reg            rx_data;
reg            rx_crc;
reg            rx_crc_lim;
reg            rx_ack;
reg            rx_ack_lim;
reg            rx_eof;
reg            rx_inter;
reg            go_early_tx_latched;

reg            rtr1;
reg            ide;
reg            rtr2;
reg     [14:0] crc_in;

reg      [7:0] tmp_data;
reg      [7:0] tmp_fifo [0:7];
reg            write_data_to_tmp_fifo;
reg      [2:0] byte_cnt;
reg            bit_stuff_cnt_en;
```

```
reg           bit_stuff_cnt_tx_en;
reg           crc_enable;
reg     [2:0] eof_cnt;
reg     [2:0] passive_cnt;

reg           transmitting;

reg           error_frame;
reg           error_frame_q;
reg           enable_error_cnt2;
reg     [2:0] error_cnt1;
reg     [2:0] error_cnt2;
reg     [2:0] delayed_dominant_cnt;
reg           enable_overload_cnt2;
reg           overload_frame;
reg           overload_frame_blocked;
reg     [2:0] overload_cnt1;
reg     [2:0] overload_cnt2;
reg           tx;
reg           crc_err;

reg           arbitration_lost;
reg           arbitration_lost_q;
reg     [4:0] arbitration_lost_capture;
reg           arbitration_cnt_en;
reg           arbitration_blocked;
reg           tx_q;

reg           need_to_tx;   // When the CAN core has something to
transmit and a dominant bit is sampled at the third bit
reg     [3:0] data_cnt;     // Counting the data bytes that are written
to FIFO
reg     [2:0] header_cnt;   // Counting header length
reg           wr_fifo;      // Write data and header to 64-byte fifo
reg     [7:0] data_for_fifo;// Multiplexed data that is stored to 64-
byte fifo

reg     [5:0] tx_pointer;
reg           tx_bit;
reg           tx_state;
reg           transmitter;
reg           finish_msg;

reg     [8:0] rx_err_cnt;
reg     [8:0] tx_err_cnt;
reg           rx_err_cnt_blocked;
reg     [3:0] bus_free_cnt;
reg           bus_free_cnt_en;
reg           bus_free;
reg           waiting_for_bus_free;

reg           node_error_passive;
reg           node_bus_off;
reg           node_bus_off_q;
reg           ack_err_latched;
reg           bit_err_latched;
reg           stuff_err_latched;
```

```
reg             form_err_latched;
reg             rule3_exc1_1;
reg             rule3_exc1_2;
reg             rule3_exc2;
reg             suspend;
reg             susp_cnt_en;
reg     [2:0]   susp_cnt;
reg             error_flag_over_blocked;

reg     [7:0]   error_capture_code;
reg     [7:6]   error_capture_code_type;
reg             error_capture_code_blocked;

wire    [4:0]   error_capture_code_segment;
wire            error_capture_code_direction;

wire            bit_de_stuff;
wire            bit_de_stuff_tx;

wire            rule5;

/* Rx state machine */
wire            go_rx_idle;
wire            go_rx_id1;
wire            go_rx_rtr1;
wire            go_rx_ide;
wire            go_rx_id2;
wire            go_rx_rtr2;
wire            go_rx_r1;
wire            go_rx_r0;
wire            go_rx_dlc;
wire            go_rx_data;
wire            go_rx_crc;
wire            go_rx_crc_lim;
wire            go_rx_ack;
wire            go_rx_ack_lim;
wire            go_rx_eof;
wire            go_overload_frame;
wire            go_rx_inter;
wire            go_error_frame;

wire            go_crc_enable;
wire            rst_crc_enable;

wire            bit_de_stuff_set;
wire            bit_de_stuff_reset;

wire            go_early_tx;
wire            go_tx;

wire    [14:0]  calculated_crc;
wire    [15:0]  r_calculated_crc;
wire            remote_rq;
wire    [3:0]   limited_data_len;
wire            form_err;

wire            error_frame_ended;
```

```
wire          overload_frame_ended;
wire          bit_err;
wire          ack_err;
wire          stuff_err;
                                    // of intermission, it starts
reading the identifier (and transmitting its own).
wire          overload_needed = 0;  // When receiver is busy, it needs
to send overload frame. Only 2 overload frames are allowed to
                                    // be send in a row. This is not
implemented because host can not send an overload request.

wire          id_ok;                // If received ID matches ID set in
registers
wire          no_byte0;             // There is no byte 0 (RTR bit set
to 1 or DLC field equal to 0). Signal used for acceptance filter.
wire          no_byte1;             // There is no byte 1 (RTR bit set
to 1 or DLC field equal to 1). Signal used for acceptance filter.

wire    [2:0] header_len;
wire          storing_header;
wire    [3:0] limited_data_len_minus1;
wire          reset_wr_fifo;
wire          err;

wire          arbitration_field;

wire   [18:0] basic_chain;
wire   [63:0] basic_chain_data;
wire   [18:0] extended_chain_std;
wire   [38:0] extended_chain_ext;
wire   [63:0] extended_chain_data;

wire          rst_tx_pointer;

wire    [7:0] r_tx_data_0;
wire    [7:0] r_tx_data_1;
wire    [7:0] r_tx_data_2;
wire    [7:0] r_tx_data_3;
wire    [7:0] r_tx_data_4;
wire    [7:0] r_tx_data_5;
wire    [7:0] r_tx_data_6;
wire    [7:0] r_tx_data_7;
wire    [7:0] r_tx_data_8;
wire    [7:0] r_tx_data_9;
wire    [7:0] r_tx_data_10;
wire    [7:0] r_tx_data_11;
wire    [7:0] r_tx_data_12;

wire          send_ack;
wire          bit_err_exc1;
wire          bit_err_exc2;
wire          bit_err_exc3;
wire          bit_err_exc4;
wire          bit_err_exc5;
wire          error_flag_over;
wire          overload_flag_over;
```

```
assign go_rx_idle     =                    sample_point &  sampled_bit &
last_bit_of_inter | bus_free & (~node_bus_off);
assign go_rx_id1      =                    sample_point &
(~sampled_bit) & (rx_idle | last_bit_of_inter);
assign go_rx_rtr1     = (~bit_de_stuff) & sample_point &  rx_id1  &
(bit_cnt == 10);
assign go_rx_ide      = (~bit_de_stuff) & sample_point &  rx_rtr1;
assign go_rx_id2      = (~bit_de_stuff) & sample_point &  rx_ide  &
sampled_bit;
assign go_rx_rtr2     = (~bit_de_stuff) & sample_point &  rx_id2  &
(bit_cnt == 17);
assign go_rx_r1       = (~bit_de_stuff) & sample_point &  rx_rtr2;
assign go_rx_r0       = (~bit_de_stuff) & sample_point & (rx_ide  &
(~sampled_bit) | rx_r1);
assign go_rx_dlc      = (~bit_de_stuff) & sample_point &  rx_r0;
assign go_rx_data     = (~bit_de_stuff) & sample_point &  rx_dlc  &
(bit_cnt == 3) &  (sampled_bit   |   (|data_len[2:0])) & (~remote_rq);
assign go_rx_crc      = (~bit_de_stuff) & sample_point & (rx_dlc  &
(bit_cnt == 3) & ((~sampled_bit) & (~(|data_len[2:0])) | remote_rq) |
                                                          rx_data &
(bit_cnt == ((limited_data_len<<3) - 1'b1)));
assign go_rx_crc_lim  = (~bit_de_stuff) & sample_point &  rx_crc  &
(bit_cnt == 14);
assign go_rx_ack      =                    sample_point &  rx_crc_lim;
assign go_rx_ack_lim  =                    sample_point &  rx_ack;
assign go_rx_eof      =                    sample_point &  rx_ack_lim;
assign go_rx_inter    =                  ((sample_point &  rx_eof  &
(eof_cnt == 6)) | error_frame_ended | overload_frame_ended) &
(~overload_needed);

assign go_error_frame = (form_err | stuff_err | bit_err | ack_err |
(crc_err & go_rx_eof));
assign error_frame_ended = (error_cnt2 == 7) & tx_point;
assign overload_frame_ended = (overload_cnt2 == 7) & tx_point;

assign go_overload_frame = (   ((sample_point &  rx_eof  & (eof_cnt ==
6)) | error_frame_ended | overload_frame_ended) & overload_needed |
                               sample_point & (~sampled_bit) & rx_inter
& (bit_cnt < 2)                                                    |
                               sample_point & (~sampled_bit) &
((error_cnt2 == 7) | (overload_cnt2 == 7))
                           )
                           & (~overload_frame_blocked)
                           ;

assign go_crc_enable  = hard_sync | go_tx;
assign rst_crc_enable = go_rx_crc;

assign bit_de_stuff_set   = go_rx_id1 & (~go_error_frame);
assign bit_de_stuff_reset = go_rx_crc_lim | reset_mode | go_error_frame
| go_overload_frame;

assign remote_rq = ((~ide) & rtr1) | (ide & rtr2);
assign limited_data_len = (data_len < 8)? data_len : 4'h8;

assign ack_err = rx_ack & sample_point & sampled_bit & tx_state &
(~self_test_mode);
```

```
assign bit_err = (tx_state | error_frame | overload_frame | rx_ack) &
sample_point & (tx != sampled_bit) & (~bit_err_exc1) & (~bit_err_exc2)
& (~bit_err_exc3) & (~bit_err_exc4) & (~bit_err_exc5);
assign bit_err_exc1 = tx_state & arbitration_field & tx;
assign bit_err_exc2 = rx_ack & tx;
assign bit_err_exc3 = error_frame & node_error_passive & (error_cnt1 <
7);
assign bit_err_exc4 = (error_frame & (error_cnt1 == 7) &
(~enable_error_cnt2)) | (overload_frame & (overload_cnt1 == 7) &
(~enable_overload_cnt2));
assign bit_err_exc5 = (error_frame & (error_cnt2 == 7)) |
(overload_frame & (overload_cnt2 == 7));

assign arbitration_field = rx_id1 | rx_rtr1 | rx_ide | rx_id2 |
rx_rtr2;

assign last_bit_of_inter = rx_inter & (bit_cnt == 2);

// Rx idle state
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_idle <= 1'b0;
  else if (reset_mode | go_rx_id1 | error_frame)
    rx_idle <=#Tp 1'b0;
  else if (go_rx_idle)
    rx_idle <=#Tp 1'b1;
end

// Rx id1 state
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_id1 <= 1'b0;
  else if (reset_mode | go_rx_rtr1 | error_frame)
    rx_id1 <=#Tp 1'b0;
  else if (go_rx_id1)
    rx_id1 <=#Tp 1'b1;
end

// Rx rtr1 state
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_rtr1 <= 1'b0;
  else if (reset_mode | go_rx_ide | error_frame)
    rx_rtr1 <=#Tp 1'b0;
  else if (go_rx_rtr1)
    rx_rtr1 <=#Tp 1'b1;
end

// Rx ide state
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_ide <= 1'b0;
  else if (reset_mode | go_rx_r0 | go_rx_id2 | error_frame)
```

```
    rx_ide <=#Tp 1'b0;
  else if (go_rx_ide)
    rx_ide <=#Tp 1'b1;
end

// Rx id2 state
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_id2 <= 1'b0;
  else if (reset_mode | go_rx_rtr2 | error_frame)
    rx_id2 <=#Tp 1'b0;
  else if (go_rx_id2)
    rx_id2 <=#Tp 1'b1;
end

// Rx rtr2 state
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_rtr2 <= 1'b0;
  else if (reset_mode | go_rx_r1 | error_frame)
    rx_rtr2 <=#Tp 1'b0;
  else if (go_rx_rtr2)
    rx_rtr2 <=#Tp 1'b1;
end

// Rx r0 state
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_r1 <= 1'b0;
  else if (reset_mode | go_rx_r0 | error_frame)
    rx_r1 <=#Tp 1'b0;
  else if (go_rx_r1)
    rx_r1 <=#Tp 1'b1;
end

// Rx r0 state
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_r0 <= 1'b0;
  else if (reset_mode | go_rx_dlc | error_frame)
    rx_r0 <=#Tp 1'b0;
  else if (go_rx_r0)
    rx_r0 <=#Tp 1'b1;
end

// Rx dlc state
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_dlc <= 1'b0;
  else if (reset_mode | go_rx_data | go_rx_crc | error_frame)
    rx_dlc <=#Tp 1'b0;
  else if (go_rx_dlc)
```

```
    rx_dlc <=#Tp 1'b1;
end
// Rx data state
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_data <= 1'b0;
  else if (reset_mode | go_rx_crc | error_frame)
    rx_data <=#Tp 1'b0;
  else if (go_rx_data)
    rx_data <=#Tp 1'b1;
end

// Rx crc state
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_crc <= 1'b0;
  else if (reset_mode | go_rx_crc_lim | error_frame)
    rx_crc <=#Tp 1'b0;
  else if (go_rx_crc)
    rx_crc <=#Tp 1'b1;
end

// Rx crc delimiter state
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_crc_lim <= 1'b0;
  else if (reset_mode | go_rx_ack | error_frame)
    rx_crc_lim <=#Tp 1'b0;
  else if (go_rx_crc_lim)
    rx_crc_lim <=#Tp 1'b1;
end

// Rx ack state
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_ack <= 1'b0;
  else if (reset_mode | go_rx_ack_lim | error_frame)
    rx_ack <=#Tp 1'b0;
  else if (go_rx_ack)
    rx_ack <=#Tp 1'b1;
end

// Rx ack delimiter state
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_ack_lim <= 1'b0;
  else if (reset_mode | go_rx_eof | error_frame)
    rx_ack_lim <=#Tp 1'b0;
  else if (go_rx_ack_lim)
    rx_ack_lim <=#Tp 1'b1;
end
```

```
// Rx eof state
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_eof <= 1'b0;
  else if (go_rx_inter | error_frame | go_overload_frame)
    rx_eof <=#Tp 1'b0;
  else if (go_rx_eof)
    rx_eof <=#Tp 1'b1;
end

// Interframe space
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_inter <= 1'b0;
  else if (reset_mode | go_rx_idle | go_rx_id1 | go_overload_frame |
go_error_frame)
    rx_inter <=#Tp 1'b0;
  else if (go_rx_inter)
    rx_inter <=#Tp 1'b1;
end

// ID register
always @ (posedge clk or posedge rst)
begin
  if (rst)
    id <= 0;
  else if (sample_point & (rx_id1 | rx_id2) & (~bit_de_stuff))
    id <=#Tp {id[27:0], sampled_bit};
end

// rtr1 bit
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rtr1 <= 0;
  else if (sample_point & rx_rtr1 & (~bit_de_stuff))
    rtr1 <=#Tp sampled_bit;
end

// rtr2 bit
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rtr2 <= 0;
  else if (sample_point & rx_rtr2 & (~bit_de_stuff))
    rtr2 <=#Tp sampled_bit;
end

// ide bit
always @ (posedge clk or posedge rst)
begin
  if (rst)
    ide <= 0;
  else if (sample_point & rx_ide & (~bit_de_stuff))
    ide <=#Tp sampled_bit;
```

```
end

// Data length
always @ (posedge clk or posedge rst)
begin
  if (rst)
    data_len <= 0;
  else if (sample_point & rx_dlc & (~bit_de_stuff))
    data_len <=#Tp {data_len[2:0], sampled_bit};
end

// Data
always @ (posedge clk or posedge rst)
begin
  if (rst)
    tmp_data <= 0;
  else if (sample_point & rx_data & (~bit_de_stuff))
    tmp_data <=#Tp {tmp_data[6:0], sampled_bit};
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    write_data_to_tmp_fifo <= 0;
  else if (sample_point & rx_data & (~bit_de_stuff) & (&bit_cnt[2:0]))
    write_data_to_tmp_fifo <=#Tp 1'b1;
  else
    write_data_to_tmp_fifo <=#Tp 0;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    byte_cnt <= 0;
  else if (write_data_to_tmp_fifo)
    byte_cnt <=#Tp byte_cnt + 1;
  else if (reset_mode | (sample_point & go_rx_crc_lim))
    byte_cnt <=#Tp 0;
end

always @ (posedge clk)
begin
  if (write_data_to_tmp_fifo)
    tmp_fifo[byte_cnt] <=#Tp tmp_data;
end

// CRC
always @ (posedge clk or posedge rst)
begin
  if (rst)
    crc_in <= 0;
  else if (sample_point & rx_crc & (~bit_de_stuff))
    crc_in <=#Tp {crc_in[13:0], sampled_bit};
end

// bit_cnt
always @ (posedge clk or posedge rst)
```

```
begin
  if (rst)
    bit_cnt <= 0;
  else if (go_rx_id1 | go_rx_id2 | go_rx_dlc | go_rx_data | go_rx_crc |
           go_rx_ack | go_rx_eof | go_rx_inter | go_error_frame |
go_overload_frame)
    bit_cnt <=#Tp 0;
  else if (sample_point & (~bit_de_stuff))
    bit_cnt <=#Tp bit_cnt + 1'b1;
end

// eof_cnt
always @ (posedge clk or posedge rst)
begin
  if (rst)
    eof_cnt <= 0;
  else if (sample_point)
    begin
      if (reset_mode | go_rx_inter | go_error_frame |
go_overload_frame)
        eof_cnt <=#Tp 0;
      else if (rx_eof)
        eof_cnt <=#Tp eof_cnt + 1'b1;
    end
end

// Enabling bit de-stuffing
always @ (posedge clk or posedge rst)
begin
  if (rst)
    bit_stuff_cnt_en <= 1'b0;
  else if (bit_de_stuff_set)
    bit_stuff_cnt_en <=#Tp 1'b1;
  else if (bit_de_stuff_reset)
    bit_stuff_cnt_en <=#Tp 1'b0;
end

// bit_stuff_cnt
always @ (posedge clk or posedge rst)
begin
  if (rst)
    bit_stuff_cnt <= 1;
  else if (bit_de_stuff_reset)
    bit_stuff_cnt <=#Tp 1;
  else if (sample_point & bit_stuff_cnt_en)
    begin
      if (bit_stuff_cnt == 5)
        bit_stuff_cnt <=#Tp 1;
      else if (sampled_bit == sampled_bit_q)
        bit_stuff_cnt <=#Tp bit_stuff_cnt + 1'b1;
      else
        bit_stuff_cnt <=#Tp 1;
    end
end

// Enabling bit de-stuffing for tx
always @ (posedge clk or posedge rst)
```

```
begin
  if (rst)
    bit_stuff_cnt_tx_en <= 1'b0;
  else if (bit_de_stuff_set & transmitting)
    bit_stuff_cnt_tx_en <=#Tp 1'b1;
  else if (bit_de_stuff_reset)
    bit_stuff_cnt_tx_en <=#Tp 1'b0;
end

// bit_stuff_cnt_tx
always @ (posedge clk or posedge rst)
begin
  if (rst)
    bit_stuff_cnt_tx <= 1;
  else if (bit_de_stuff_reset)
    bit_stuff_cnt_tx <=#Tp 1;
  else if (tx_point_q & bit_stuff_cnt_en)
    begin
      if (bit_stuff_cnt_tx == 5)
        bit_stuff_cnt_tx <=#Tp 1;
      else if (tx == tx_q)
        bit_stuff_cnt_tx <=#Tp bit_stuff_cnt_tx + 1'b1;
      else
        bit_stuff_cnt_tx <=#Tp 1;
    end
end

assign bit_de_stuff = bit_stuff_cnt == 5;
assign bit_de_stuff_tx = bit_stuff_cnt_tx == 5;

// stuff_err
assign stuff_err = sample_point & bit_stuff_cnt_en & bit_de_stuff &
(sampled_bit == sampled_bit_q);

// Generating delayed signals
always @ (posedge clk)
begin
  reset_mode_q <=#Tp reset_mode;
  node_bus_off_q <=#Tp node_bus_off;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    crc_enable <= 1'b0;
  else if (go_crc_enable)
    crc_enable <=#Tp 1'b1;
  else if (reset_mode | rst_crc_enable)
    crc_enable <=#Tp 1'b0;
end

// CRC error generation
always @ (posedge clk or posedge rst)
begin
  if (rst)
    crc_err <= 1'b0;
  else if (go_rx_ack)
```

```
    crc_err <=#Tp crc_in != calculated_crc;
  else if (reset_mode | error_frame_ended)
    crc_err <=#Tp 1'b0;
end

// Conditions for form error
assign form_err = sample_point & ( ((~bit_de_stuff) & rx_ide     &
sampled_bit & (~rtr1)      ) |
                                   (                  rx_crc_lim &
(~sampled_bit)               ) |
                                   (                  rx_ack_lim &
(~sampled_bit)               ) |
                                   ((eof_cnt < 6)   & rx_eof     &
(~sampled_bit) & (~tx_state) ) |
                                   (                & rx_eof     &
(~sampled_bit) &   tx_state  )
                                 );

always @ (posedge clk or posedge rst)
begin
  if (rst)
    ack_err_latched <= 1'b0;
  else if (reset_mode | error_frame_ended | go_overload_frame)
    ack_err_latched <=#Tp 1'b0;
  else if (ack_err)
    ack_err_latched <=#Tp 1'b1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    bit_err_latched <= 1'b0;
  else if (reset_mode | error_frame_ended | go_overload_frame)
    bit_err_latched <=#Tp 1'b0;
  else if (bit_err)
    bit_err_latched <=#Tp 1'b1;
end

// Rule 5 (Fault confinement).
assign rule5 = (~node_error_passive) & bit_err &  (error_frame    &
(error_cnt1    < 7) |
                                                   overload_frame &
(overload_cnt1 < 7) );

// Rule 3 exception 1 - first part (Fault confinement).
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rule3_exc1_1 <= 1'b0;
  else if (reset_mode | error_flag_over | rule3_exc1_2)
    rule3_exc1_1 <=#Tp 1'b0;
  else if (transmitter & node_error_passive & ack_err)
    rule3_exc1_1 <=#Tp 1'b1;
end

// Rule 3 exception 1 - second part (Fault confinement).
always @ (posedge clk or posedge rst)
```

```
begin
  if (rst)
    rule3_exc1_2 <= 1'b0;
  else if (reset_mode | error_flag_over)
    rule3_exc1_2 <=#Tp 1'b0;
  else if (rule3_exc1_1)
    rule3_exc1_2 <=#Tp 1'b1;
  else if ((error_cnt1 < 7) & sample_point & (~sampled_bit))
    rule3_exc1_2 <=#Tp 1'b0;
end

// Rule 3 exception 2 (Fault confinement).
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rule3_exc2 <= 1'b0;
  else if (reset_mode | error_flag_over)
    rule3_exc2 <=#Tp 1'b0;
  else if (transmitter & stuff_err & arbitration_field & sample_point &
tx & (~sampled_bit))
    rule3_exc2 <=#Tp 1'b1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    stuff_err_latched <= 1'b0;
  else if (reset_mode | error_frame_ended | go_overload_frame)
    stuff_err_latched <=#Tp 1'b0;
  else if (stuff_err)
    stuff_err_latched <=#Tp 1'b1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    form_err_latched <= 1'b0;
  else if (reset_mode | error_frame_ended | go_overload_frame)
    form_err_latched <=#Tp 1'b0;
  else if (form_err)
    form_err_latched <=#Tp 1'b1;
end


// Instantiation of the RX CRC module
can_crc i_can_crc_rx
(
  .clk(clk),
  .data(sampled_bit),
  .enable(crc_enable & sample_point & (~bit_de_stuff)),
  .initialize(go_crc_enable),
  .crc(calculated_crc)
);

assign no_byte0 = rtr1 | (data_len<1);
assign no_byte1 = rtr1 | (data_len<2);
```

```
can_acf i_can_acf
(
  .clk(clk),
  .rst(rst),

  .id(id),

  /* Mode register */
  .reset_mode(reset_mode),
  .acceptance_filter_mode(acceptance_filter_mode),

  // Clock Divider register
  .extended_mode(extended_mode),

  /* This section is for BASIC and EXTENDED mode */
  /* Acceptance code register */
  .acceptance_code_0(acceptance_code_0),

  /* Acceptance mask register */
  .acceptance_mask_0(acceptance_mask_0),
  /* End: This section is for BASIC and EXTENDED mode */

  /* This section is for EXTENDED mode */
  /* Acceptance code register */
  .acceptance_code_1(acceptance_code_1),
  .acceptance_code_2(acceptance_code_2),
  .acceptance_code_3(acceptance_code_3),

  /* Acceptance mask register */
  .acceptance_mask_1(acceptance_mask_1),
  .acceptance_mask_2(acceptance_mask_2),
  .acceptance_mask_3(acceptance_mask_3),
  /* End: This section is for EXTENDED mode */

  .go_rx_crc_lim(go_rx_crc_lim),
  .go_rx_inter(go_rx_inter),
  .go_error_frame(go_error_frame),

  .data0(tmp_fifo[0]),
  .data1(tmp_fifo[1]),
  .rtr1(rtr1),
  .rtr2(rtr2),
  .ide(ide),
  .no_byte0(no_byte0),
  .no_byte1(no_byte1),

  .id_ok(id_ok)

);

assign header_len[2:0] = extended_mode ? (ide? (3'h5) : (3'h3)) : 3'h2;
assign storing_header = header_cnt < header_len;
assign limited_data_len_minus1[3:0] = remote_rq? 4'hf : ((data_len <
8)? (data_len -1'b1) : 4'h7);   // - 1 because counter counts from 0
assign reset_wr_fifo = (data_cnt == (limited_data_len_minus1 +
header_len)) | reset_mode;
```

```
assign err = form_err | stuff_err | bit_err | ack_err |
form_err_latched | stuff_err_latched | bit_err_latched |
ack_err_latched | crc_err;

// Write enable signal for 64-byte rx fifo
always @ (posedge clk or posedge rst)
begin
  if (rst)
    wr_fifo <= 1'b0;
  else if (reset_wr_fifo)
    wr_fifo <=#Tp 1'b0;
  else if (go_rx_inter & id_ok & (~error_frame_ended) & ((~tx_state) |
self_rx_request))
    wr_fifo <=#Tp 1'b1;
end


// Header counter. Header length depends on the mode of operation and
frame format.
always @ (posedge clk or posedge rst)
begin
  if (rst)
    header_cnt <= 0;
  else if (reset_wr_fifo)
    header_cnt <=#Tp 0;
  else if (wr_fifo & storing_header)
    header_cnt <=#Tp header_cnt + 1;
end

// Data counter. Length of the data is limited to 8 bytes.
always @ (posedge clk or posedge rst)
begin
  if (rst)
    data_cnt <= 0;
  else if (reset_wr_fifo)
    data_cnt <=#Tp 0;
  else if (wr_fifo)
    data_cnt <=#Tp data_cnt + 1;
end


// Multiplexing data that is stored to 64-byte fifo depends on the mode
of operation and frame format
always @ (extended_mode or ide or data_cnt or header_cnt or  header_len
or
          storing_header or id or rtr1 or rtr2 or data_len or
          tmp_fifo[0] or tmp_fifo[2] or tmp_fifo[4] or tmp_fifo[6] or
          tmp_fifo[1] or tmp_fifo[3] or tmp_fifo[5] or tmp_fifo[7])
begin
  if (storing_header)
    begin
      if (extended_mode)      // extended mode
        begin
          if (ide)              // extended format
            begin
              case (header_cnt) // synthesis parallel_case
                3'h0  : data_for_fifo <= {1'b1, rtr2, 2'h0, data_len};
```

```
                3'h1  : data_for_fifo <= id[28:21];
                3'h2  : data_for_fifo <= id[20:13];
                3'h3  : data_for_fifo <= id[12:5];
                3'h4  : data_for_fifo <= {id[4:0], 3'h0};
                default: data_for_fifo <= 0;
              endcase
            end
          else                    // standard format
            begin
              case (header_cnt) // synthesis parallel_case
                3'h0  : data_for_fifo <= {1'b0, rtr1, 2'h0, data_len};
                3'h1  : data_for_fifo <= id[10:3];
                3'h2  : data_for_fifo <= {id[2:0], 5'h0};
                default: data_for_fifo <= 0;
              endcase
            end
        end
      else                      // normal mode
        begin
          case (header_cnt) // synthesis parallel_case
            3'h0  : data_for_fifo <= id[10:3];
            3'h1  : data_for_fifo <= {id[2:0], rtr1, data_len};
            default: data_for_fifo <= 0;
          endcase
        end
    end
  else
    data_for_fifo <= tmp_fifo[data_cnt-header_len];
end

// Instantiation of the RX fifo module
can_fifo i_can_fifo
(
  .clk(clk),
  .rst(rst),

  .wr(wr_fifo),

  .data_in(data_for_fifo),
  .addr(addr),
  .data_out(data_out),
  .fifo_selected(fifo_selected),

  .reset_mode(reset_mode),
  .release_buffer(release_buffer),
  .extended_mode(extended_mode),
  .overrun(overrun),
  .info_empty(info_empty),
  .info_cnt(rx_message_counter)
);

// Transmitting error frame.
always @ (posedge clk or posedge rst)
begin
  if (rst)
    error_frame <= 1'b0;
  else if (reset_mode | error_frame_ended | go_overload_frame)
```

```
    error_frame <=#Tp 1'b0;
  else if (go_error_frame)
    error_frame <=#Tp 1'b1;
end

always @ (posedge clk)
begin
  if (sample_point)
    error_frame_q <=#Tp error_frame;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    error_cnt1 <= 1'b0;
  else if (reset_mode | error_frame_ended | go_error_frame |
go_overload_frame)
    error_cnt1 <=#Tp 1'b0;
  else if (error_frame & tx_point & (error_cnt1 < 7))
    error_cnt1 <=#Tp error_cnt1 + 1'b1;
end

assign error_flag_over = ((~node_error_passive) & sample_point &
(error_cnt1 == 7) | node_error_passive  & sample_point & (passive_cnt
== 5)) & (~enable_error_cnt2);

always @ (posedge clk or posedge rst)
begin
  if (rst)
    error_flag_over_blocked <= 1'b0;
  else if (reset_mode | error_frame_ended | go_error_frame |
go_overload_frame)
    error_flag_over_blocked <=#Tp 1'b0;
  else if (error_flag_over)
    error_flag_over_blocked <=#Tp 1'b1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    enable_error_cnt2 <= 1'b0;
  else if (reset_mode | error_frame_ended | go_error_frame |
go_overload_frame)
    enable_error_cnt2 <=#Tp 1'b0;
  else if (error_frame & (error_flag_over & sampled_bit))
    enable_error_cnt2 <=#Tp 1'b1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    error_cnt2 <= 0;
  else if (reset_mode | error_frame_ended | go_error_frame |
go_overload_frame)
    error_cnt2 <=#Tp 0;
  else if (enable_error_cnt2 & tx_point)
    error_cnt2 <=#Tp error_cnt2 + 1'b1;
```

```
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    delayed_dominant_cnt <= 0;
  else if (reset_mode | enable_error_cnt2 | go_error_frame |
enable_overload_cnt2 | go_overload_frame)
    delayed_dominant_cnt <=#Tp 0;
  else if (sample_point & (~sampled_bit) & ((error_cnt1 == 7) |
(overload_cnt1 == 7)))
    delayed_dominant_cnt <=#Tp delayed_dominant_cnt + 1'b1;
end

// passive_cnt
always @ (posedge clk or posedge rst)
begin
  if (rst)
    passive_cnt <= 0;
  else if (reset_mode | error_frame_ended | go_error_frame |
go_overload_frame)
    passive_cnt <=#Tp 0;
  else if (sample_point & (passive_cnt < 5))
    begin
      if (error_frame_q & (~enable_error_cnt2) & (sampled_bit ==
sampled_bit_q))
        passive_cnt <=#Tp passive_cnt + 1'b1;
      else
        passive_cnt <=#Tp 0;
    end
end

// Transmitting overload frame.
always @ (posedge clk or posedge rst)
begin
  if (rst)
    overload_frame <= 1'b0;
  else if (reset_mode | overload_frame_ended | go_error_frame)
    overload_frame <=#Tp 1'b0;
  else if (go_overload_frame)
    overload_frame <=#Tp 1'b1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    overload_cnt1 <= 1'b0;
  else if (reset_mode | overload_frame_ended | go_error_frame |
go_overload_frame)
    overload_cnt1 <=#Tp 1'b0;
  else if (overload_frame & tx_point & (overload_cnt1 < 7))
    overload_cnt1 <=#Tp overload_cnt1 + 1'b1;
end

assign overload_flag_over = sample_point & (overload_cnt1 == 7) &
(~enable_overload_cnt2);
```

```
always @ (posedge clk or posedge rst)
begin
  if (rst)
    enable_overload_cnt2 <= 1'b0;
  else if (reset_mode | overload_frame_ended | go_error_frame |
go_overload_frame)
    enable_overload_cnt2 <=#Tp 1'b0;
  else if (overload_frame & (overload_flag_over & sampled_bit))
    enable_overload_cnt2 <=#Tp 1'b1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    overload_cnt2 <= 0;
  else if (reset_mode | overload_frame_ended | go_error_frame |
go_overload_frame)
    overload_cnt2 <=#Tp 0;
  else if (enable_overload_cnt2 & tx_point)
    overload_cnt2 <=#Tp overload_cnt2 + 1'b1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    overload_frame_blocked <= 0;
  else if (reset_mode | go_error_frame | go_rx_id1)
    overload_frame_blocked <=#Tp 0;
  else if (go_overload_frame & overload_frame)            // This is a
second sequential overload
    overload_frame_blocked <=#Tp 1'b1;
end

assign send_ack = (~tx_state) & rx_ack & (~err) & (~listen_only_mode);

always @ (posedge clk or posedge rst)
begin
  if (rst)
    tx <= 1'b1;
  else if (reset_mode)
// Reset
    tx <=#Tp 1'b1;
  else if (tx_point)
    begin
      if (tx_state)
// Transmitting message
        tx <=#Tp ((~bit_de_stuff_tx) & tx_bit) | (bit_de_stuff_tx &
(~tx_q));
      else if (send_ack)
// Acknowledge
        tx <=#Tp 1'b0;
      else if (overload_frame)
// Transmitting overload frame
        begin
          if (overload_cnt1 < 6)
            tx <=#Tp 1'b0;
          else
```

```
            tx <=#Tp 1'b1;
        end
      else if (error_frame)
// Transmitting error frame
        begin
          if (error_cnt1 < 6)
            begin
              if (node_error_passive)
                tx <=#Tp 1'b1;
              else
                tx <=#Tp 1'b0;
            end
          else
            tx <=#Tp 1'b1;
        end
      else
        tx <=#Tp 1'b1;
    end
end

always @ (posedge clk)
begin
  if (tx_point)
    tx_q <=#Tp tx & (~go_early_tx_latched);
end

/* Delayed tx point */
always @ (posedge clk)
begin
  tx_point_q <=#Tp tx_point;
end

/* Changing bit order from [7:0] to [0:7] */
can_ibo i_ibo_tx_data_0  (.di(tx_data_0),  .do(r_tx_data_0));
can_ibo i_ibo_tx_data_1  (.di(tx_data_1),  .do(r_tx_data_1));
can_ibo i_ibo_tx_data_2  (.di(tx_data_2),  .do(r_tx_data_2));
can_ibo i_ibo_tx_data_3  (.di(tx_data_3),  .do(r_tx_data_3));
can_ibo i_ibo_tx_data_4  (.di(tx_data_4),  .do(r_tx_data_4));
can_ibo i_ibo_tx_data_5  (.di(tx_data_5),  .do(r_tx_data_5));
can_ibo i_ibo_tx_data_6  (.di(tx_data_6),  .do(r_tx_data_6));
can_ibo i_ibo_tx_data_7  (.di(tx_data_7),  .do(r_tx_data_7));
can_ibo i_ibo_tx_data_8  (.di(tx_data_8),  .do(r_tx_data_8));
can_ibo i_ibo_tx_data_9  (.di(tx_data_9),  .do(r_tx_data_9));
can_ibo i_ibo_tx_data_10 (.di(tx_data_10), .do(r_tx_data_10));
can_ibo i_ibo_tx_data_11 (.di(tx_data_11), .do(r_tx_data_11));
can_ibo i_ibo_tx_data_12 (.di(tx_data_12), .do(r_tx_data_12));

/* Changing bit order from [14:0] to [0:14] */
can_ibo i_calculated_crc0 (.di(calculated_crc[14:7]),
.do(r_calculated_crc[7:0]));
can_ibo i_calculated_crc1 (.di({calculated_crc[6:0], 1'b0}),
.do(r_calculated_crc[15:8]));

assign basic_chain = {r_tx_data_1[7:4], 2'h0, r_tx_data_1[3:0],
r_tx_data_0[7:0], 1'b0};
assign basic_chain_data = {r_tx_data_9, r_tx_data_8, r_tx_data_7,
r_tx_data_6, r_tx_data_5, r_tx_data_4, r_tx_data_3, r_tx_data_2};
```

```
assign extended_chain_std = {r_tx_data_0[7:4], 2'h0, r_tx_data_0[1],
r_tx_data_2[2:0], r_tx_data_1[7:0], 1'b0};
assign extended_chain_ext = {r_tx_data_0[7:4], 2'h0, r_tx_data_0[1],
r_tx_data_4[4:0], r_tx_data_3[7:0], r_tx_data_2[7:3], 1'b1, 1'b1,
r_tx_data_2[2:0], r_tx_data_1[7:0], 1'b0};
assign extended_chain_data = {r_tx_data_12, r_tx_data_11, r_tx_data_10,
r_tx_data_9, r_tx_data_8, r_tx_data_7, r_tx_data_6, r_tx_data_5};

always @ (extended_mode or rx_data or tx_pointer or extended_chain_data
or rx_crc or r_calculated_crc or
          r_tx_data_0   or extended_chain_ext or extended_chain_std or
basic_chain_data or basic_chain or
          finish_msg)
begin
  if (extended_mode)
    begin
      if (rx_data)  // data stage
        tx_bit = extended_chain_data[tx_pointer];
      else if (rx_crc)
        tx_bit = r_calculated_crc[tx_pointer];
      else if (finish_msg)
        tx_bit = 1'b1;
      else
        begin
          if (r_tx_data_0[0])    // Extended frame
            tx_bit = extended_chain_ext[tx_pointer];
          else
            tx_bit = extended_chain_std[tx_pointer];
        end
    end
  else  // Basic mode
    begin
      if (rx_data)  // data stage
        tx_bit = basic_chain_data[tx_pointer];
      else if (rx_crc)
        tx_bit = r_calculated_crc[tx_pointer];
      else if (finish_msg)
        tx_bit = 1'b1;
      else
        tx_bit = basic_chain[tx_pointer];
    end
end

assign rst_tx_pointer = ((~bit_de_stuff_tx) & tx_point & (~rx_data) &
extended_mode  &   r_tx_data_0[0]   & tx_pointer == 38
) |   // arbitration + control for extended format
                        ((~bit_de_stuff_tx) & tx_point & (~rx_data) &
extended_mode  & (~r_tx_data_0[0])  & tx_pointer == 18
) |   // arbitration + control for extended format
                        ((~bit_de_stuff_tx) & tx_point & (~rx_data) &
(~extended_mode)                          & tx_pointer == 18
) |   // arbitration + control for standard format
                        ((~bit_de_stuff_tx) & tx_point &   rx_data  &
extended_mode                           & tx_pointer == (8 * tx_data_0[3:0]
- 1)) |   // data
                        ((~bit_de_stuff_tx) & tx_point &   rx_data  &
(~extended_mode)                          & tx_pointer == (8 *
```

```
tx_data_1[3:0] - 1)) |   // data
                        (                       tx_point &   rx_crc_lim
) |   // crc
                        (go_rx_idle
) |   // at the end
                        (reset_mode
) |
                        (overload_frame
) |
                        (error_frame
) ;

always @ (posedge clk or posedge rst)
begin
  if (rst)
    tx_pointer <= 'h0;
  else if (rst_tx_pointer)
    tx_pointer <=#Tp 'h0;
  else if (go_early_tx | (tx_point & tx_state & (~bit_de_stuff_tx)))
    tx_pointer <=#Tp tx_pointer + 1'b1;
end

assign tx_successful = transmitter & go_rx_inter &
((~error_frame_ended) & (~overload_frame_ended) & (~arbitration_lost) |
single_shot_transmission);

always @ (posedge clk or posedge rst)
begin
  if (rst)
    need_to_tx <= 1'b0;
  else if (tx_successful | reset_mode | (abort_tx & (~transmitting)))
    need_to_tx <=#Tp 1'h0;
  else if (tx_request & sample_point)
    need_to_tx <=#Tp 1'b1;
end

assign go_early_tx = (~listen_only_mode) & need_to_tx & (~tx_state) &
(~suspend) & sample_point & (~sampled_bit) & (rx_idle |
last_bit_of_inter);
assign go_tx       = (~listen_only_mode) & need_to_tx & (~tx_state) &
(~suspend) & (go_early_tx | rx_idle);

// go_early_tx latched (for proper bit_de_stuff generation)
always @ (posedge clk or posedge rst)
begin
  if (rst)
    go_early_tx_latched <= 1'b0;
  else if (tx_point_q)
    go_early_tx_latched <=#Tp 1'b0;
  else if (go_early_tx)
    go_early_tx_latched <=#Tp 1'b1;
end

// Tx state
always @ (posedge clk or posedge rst)
begin
  if (rst)
```

```
    tx_state <= 1'b0;
  else if (reset_mode | go_rx_inter | error_frame | arbitration_lost)
    tx_state <=#Tp 1'b0;
  else if (go_tx)
    tx_state <=#Tp 1'b1;
end

// Node is a transmitter
always @ (posedge clk or posedge rst)
begin
  if (rst)
    transmitter <= 1'b0;
  else if (go_tx)
    transmitter <=#Tp 1'b1;
  else if (reset_mode | go_rx_inter)
    transmitter <=#Tp 1'b0;
end

// Signal "transmitting" signals that the core is a transmitting
(message, error frame or overload frame). No synchronization is done
meanwhile.
// Node might be both transmitter or receiver (sending error or
overload frame)
always @ (posedge clk or posedge rst)
begin
  if (rst)
    transmitting <= 1'b0;
  else if (go_error_frame | go_overload_frame | go_tx)
    transmitting <=#Tp 1'b1;
  else if (reset_mode | go_rx_idle | (go_rx_id1 & (~tx_state)) |
(arbitration_lost & tx_state))
    transmitting <=#Tp 1'b0;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    suspend <= 0;
  else if (reset_mode | (sample_point & (susp_cnt == 7)))
    suspend <=#Tp 0;
  else if (go_rx_inter & transmitter & node_error_passive)
    suspend <=#Tp 1'b1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    susp_cnt_en <= 0;
  else if (reset_mode | (sample_point & (susp_cnt == 7)))
    susp_cnt_en <=#Tp 0;
  else if (suspend & sample_point & last_bit_of_inter)
    susp_cnt_en <=#Tp 1'b1;
end


always @ (posedge clk or posedge rst)
begin
```

```
  if (rst)
    susp_cnt <= 0;
  else if (reset_mode | (sample_point & (susp_cnt == 7)))
    susp_cnt <=#Tp 0;
  else if (susp_cnt_en & sample_point)
    susp_cnt <=#Tp susp_cnt + 1'b1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    finish_msg <= 1'b0;
  else if (go_rx_idle | go_rx_id1 | error_frame | reset_mode)
    finish_msg <=#Tp 1'b0;
  else if (go_rx_crc_lim)
    finish_msg <=#Tp 1'b1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    arbitration_lost <= 1'b0;
  else if (go_rx_idle | error_frame | reset_mode)
    arbitration_lost <=#Tp 1'b0;
  else if (tx_state & sample_point & tx & arbitration_field)
    arbitration_lost <=#Tp (~sampled_bit);
end

always @ (posedge clk)
begin
  arbitration_lost_q <=#Tp arbitration_lost;
end

assign set_arbitration_lost_irq = arbitration_lost &
(~arbitration_lost_q) & (~arbitration_blocked);

always @ (posedge clk or posedge rst)
begin
  if (rst)
    arbitration_cnt_en <= 1'b0;
  else if (arbitration_blocked)
    arbitration_cnt_en <=#Tp 1'b0;
  else if (rx_id1 & sample_point & (~arbitration_blocked))
    arbitration_cnt_en <=#Tp 1'b1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    arbitration_blocked <= 1'b0;
  else if (read_arbitration_lost_capture_reg)
    arbitration_blocked <=#Tp 1'b0;
  else if (set_arbitration_lost_irq)
    arbitration_blocked <=#Tp 1'b1;
end

always @ (posedge clk or posedge rst)
```

```
begin
  if (rst)
    arbitration_lost_capture <= 5'h0;
  else if (read_arbitration_lost_capture_reg)
    arbitration_lost_capture <=#Tp 5'h0;
  else if (sample_point & (~arbitration_blocked) & arbitration_cnt_en &
(~bit_de_stuff))
    arbitration_lost_capture <=#Tp arbitration_lost_capture + 1'b1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_err_cnt <= 'h0;
  else if (we_rx_err_cnt & (~node_bus_off))
    rx_err_cnt <=#Tp {1'b0, data_in};
  else if (set_reset_mode)
    rx_err_cnt <=#Tp 'h0;
  else
    begin
      if (~listen_only_mode)
        begin
          if ((~transmitter) & go_rx_ack_lim & (~err) & (rx_err_cnt >
0))
            begin
              if (rx_err_cnt > 127)
                rx_err_cnt <=#Tp 127;
              else
                rx_err_cnt <=#Tp rx_err_cnt - 1'b1;
            end
          else if ((rx_err_cnt < 248) & (~transmitter))    // 248 + 8 =
256
            begin
              if (go_error_frame & (~rule5))
// 1  (rule 5 is just the opposite then rule 1 exception
                rx_err_cnt <=#Tp rx_err_cnt + 1'b1;
              else if ( (error_frame & sample_point & (~sampled_bit) &
(error_cnt1 == 7) & (~rx_err_cnt_blocked)  ) |   // 2
                        (go_error_frame & rule5
) |   // 5
                        (error_frame & sample_point & (~sampled_bit) &
(delayed_dominant_cnt == 7)                )     // 6
                      )
                rx_err_cnt <=#Tp rx_err_cnt + 4'h8;
            end
        end
    end
end


always @ (posedge clk or posedge rst)
begin
  if (rst)
    tx_err_cnt <= 'h0;
  else if (we_tx_err_cnt)
    tx_err_cnt <=#Tp {1'b0, data_in};
  else
```

```
    begin
      if (set_reset_mode)
        tx_err_cnt <=#Tp 127;
      else if ((tx_err_cnt > 0) & (tx_successful | bus_free))
        tx_err_cnt <=#Tp tx_err_cnt - 1'h1;
      else if (transmitter)
        begin
          if ( (sample_point & (~sampled_bit) & (delayed_dominant_cnt
== 7)                    ) |       // 6
               (go_error_frame & rule5
) |       // 4  (rule 5 is the same as rule 4)
               (error_flag_over & (~error_flag_over_blocked) &
(~rule3_exc1_2) & (~rule3_exc2)  )         // 3
             )
            tx_err_cnt <=#Tp tx_err_cnt + 4'h8;
        end
    end
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    rx_err_cnt_blocked <= 1'b0;
  else if (reset_mode | error_frame_ended)
    rx_err_cnt_blocked <=#Tp 1'b0;
  else if (sample_point & (error_cnt1 == 7))
    rx_err_cnt_blocked <=#Tp 1'b1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    node_error_passive <= 1'b0;
  else if ((rx_err_cnt < 128) & (tx_err_cnt < 128) & error_frame_ended)
    node_error_passive <=#Tp 1'b0;
  else if (((rx_err_cnt >= 128) | (tx_err_cnt >= 128)) &
(error_frame_ended | go_error_frame | (~reset_mode) & reset_mode_q) &
(~node_bus_off))
    node_error_passive <=#Tp 1'b1;
end

assign node_error_active = ~(node_error_passive | node_bus_off);

always @ (posedge clk or posedge rst)
begin
  if (rst)
    node_bus_off <= 1'b0;
  else if ((rx_err_cnt == 0) & (tx_err_cnt == 0) & (~reset_mode) |
(we_tx_err_cnt & (data_in < 255)))
    node_bus_off <=#Tp 1'b0;
  else if ((tx_err_cnt >= 256) | (we_tx_err_cnt & (data_in == 255)))
    node_bus_off <=#Tp 1'b1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
```

```
    bus_free_cnt <= 0;
  else if (reset_mode)
    bus_free_cnt <=#Tp 0;
  else if (sample_point)
    begin
      if (sampled_bit & bus_free_cnt_en & (bus_free_cnt < 10))
        bus_free_cnt <=#Tp bus_free_cnt + 1'b1;
      else
        bus_free_cnt <=#Tp 0;
    end
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    bus_free_cnt_en <= 1'b0;
  else if ((~reset_mode) & reset_mode_q | node_bus_off_q &
(~reset_mode))
    bus_free_cnt_en <=#Tp 1'b1;
  else if (sample_point &  (bus_free_cnt==10) & (~node_bus_off))
    bus_free_cnt_en <=#Tp 1'b0;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    bus_free <= 1'b0;
  else if (sample_point & sampled_bit & (bus_free_cnt==10))
    bus_free <=#Tp 1'b1;
  else
    bus_free <=#Tp 1'b0;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    waiting_for_bus_free <= 1'b1;
  else if (bus_free & (~node_bus_off))
    waiting_for_bus_free <=#Tp 1'b0;
  else if ((~reset_mode) & reset_mode_q | node_bus_off_q &
(~reset_mode))
    waiting_for_bus_free <=#Tp 1'b1;
end

assign tx_oen = node_bus_off;

assign set_reset_mode = node_bus_off & (~node_bus_off_q);
assign error_status = (~reset_mode) & extended_mode? ((rx_err_cnt >=
error_warning_limit) | (tx_err_cnt >= error_warning_limit))    :
                                                    ((rx_err_cnt >=
96) | (tx_err_cnt >= 96))                                    ;

assign transmit_status = transmitting                  | (extended_mode
& waiting_for_bus_free);
assign receive_status  = (~rx_idle) & (~transmitting) | (extended_mode
& waiting_for_bus_free);
```

```
/* Error code capture register */
always @ (posedge clk or posedge rst)
begin
  if (rst)
    error_capture_code <= 8'h0;
  else if (read_error_code_capture_reg)
    error_capture_code <=#Tp 8'h0;
  else if (set_bus_error_irq)
    error_capture_code <=#Tp {error_capture_code_type[7:6],
error_capture_code_direction, error_capture_code_segment[4:0]};
end

assign error_capture_code_segment[0] = rx_idle | rx_ide | (rx_id2 &
(bit_cnt<13)) | rx_r1 | rx_r0 | rx_dlc | rx_ack | rx_ack_lim |
error_frame & node_error_active;
assign error_capture_code_segment[1] = rx_idle | rx_id1 | rx_id2 |
rx_dlc | rx_data | rx_ack_lim | rx_eof | rx_inter | error_frame &
node_error_passive;
assign error_capture_code_segment[2] = (rx_id1 & (bit_cnt>7)) | rx_rtr1
| rx_ide | rx_id2 | rx_rtr2 | rx_r1 | error_frame & node_error_passive
| overload_frame;
assign error_capture_code_segment[3] = (rx_id2 & (bit_cnt>4)) | rx_rtr2
| rx_r1 | rx_r0 | rx_dlc | rx_data | rx_crc | rx_crc_lim | rx_ack |
rx_ack_lim | rx_eof | overload_frame;
assign error_capture_code_segment[4] = rx_crc_lim | rx_ack | rx_ack_lim
| rx_eof | rx_inter | error_frame | overload_frame;
assign error_capture_code_direction  = ~transmitting;

always @ (bit_err or form_err or stuff_err)
begin
  if (bit_err)
    error_capture_code_type[7:6] <= 2'b00;
  else if (form_err)
    error_capture_code_type[7:6] <= 2'b01;
  else if (stuff_err)
    error_capture_code_type[7:6] <= 2'b10;
  else
    error_capture_code_type[7:6] <= 2'b11;
end

assign set_bus_error_irq = go_error_frame &
(~error_capture_code_blocked);


always @ (posedge clk or posedge rst)
begin
  if (rst)
    error_capture_code_blocked <= 1'b0;
  else if (read_error_code_capture_reg)
    error_capture_code_blocked <=#Tp 1'b0;
  else if (set_bus_error_irq)
    error_capture_code_blocked <=#Tp 1'b1;
end

endmodule
```

A.4: CRC HDL Code

The following code is the cyclic reducany check (CRC) module. This code generates the CRC sequence to be transmitted and checks the CRC sequence of incoming messages.

```
// can_crc.v
// synopsys translate_off
//`include "timescale.v"
`timescale 1ns/10ps
// synopsys translate_on
//`include "can_defines.v"

// Uncomment following line if you want to use WISHBONE interface.
Otherwise
// 8051 interface is used.
//`define   CAN_WISHBONE_IF
//`define Intel_mode

// Uncomment following line if you want to use CAN in Actel APA devices
(embedded memory used)
//define   ACTEL_APA_RAM

// Uncomment following line if you want to use CAN in Xilinx devices
(embedded memory used)
`define   XILINX_RAM

module can_crc (clk, data, enable, initialize, crc);

parameter Tp = 1;

input         clk;
input         data;
input         enable;
input         initialize;

output [14:0] crc;

reg    [14:0] crc;
wire          crc_next;
wire   [14:0] crc_tmp;

assign crc_next = data ^ crc[14];
assign crc_tmp = {crc[13:0], 1'b0};

always @ (posedge clk)
begin
  if(initialize)
    crc <= #Tp 0;
  else if (enable)
    begin
```

```
      if (crc_next)
        crc <= #Tp crc_tmp ^ 15'h4599;
      else
        crc <= #Tp crc_tmp;
    end
end

endmodule
```

A.5: ACF HDL Code

This code is the acceptance filtering component of the CAN controller. The acceptance filtering mechanism ensures that the host microcontroller in only informed of a message that is relevant for it.

```
// can_acf.v
// synopsys translate_off
//`include "timescale.v"
`timescale 1ns/10ps
// synopsys translate_on
//`include "can_defines.v"

// Uncomment following line if you want to use WISHBONE interface.
Otherwise
// 8051 interface is used.
//`define   CAN_WISHBONE_IF
//`define Intel_mode

// Uncomment following line if you want to use CAN in Actel APA devices
(embedded memory used)
//define   ACTEL_APA_RAM

// Uncomment following line if you want to use CAN in Xilinx devices
(embedded memory used)
`define   XILINX_RAM

module can_acf
(
  clk,
  rst,

  id,

  /* Mode register */
  reset_mode,
  acceptance_filter_mode,

  extended_mode,

  acceptance_code_0,
  acceptance_code_1,
  acceptance_code_2,
  acceptance_code_3,
  acceptance_mask_0,
  acceptance_mask_1,
  acceptance_mask_2,
  acceptance_mask_3,

  go_rx_crc_lim,
  go_rx_inter,
```

```
  go_error_frame,

  data0,
  data1,
  rtr1,
  rtr2,
  ide,
  no_byte0,
  no_byte1,

  id_ok
);

parameter Tp = 1;

input         clk;
input         rst;
input  [28:0] id;
input         reset_mode;
input         acceptance_filter_mode;
input         extended_mode;

input   [7:0] acceptance_code_0;
input   [7:0] acceptance_code_1;
input   [7:0] acceptance_code_2;
input   [7:0] acceptance_code_3;
input   [7:0] acceptance_mask_0;
input   [7:0] acceptance_mask_1;
input   [7:0] acceptance_mask_2;
input   [7:0] acceptance_mask_3;
input         go_rx_crc_lim;
input         go_rx_inter;
input         go_error_frame;
input   [7:0] data0;
input   [7:0] data1;
input         rtr1;
input         rtr2;
input         ide;
input         no_byte0;
input         no_byte1;

output        id_ok;

reg           id_ok;

wire          match;
wire          match_sf_std;
wire          match_sf_ext;
wire          match_df_std;
wire          match_df_ext;

// Working in basic mode. ID match for standard format (11-bit ID).
assign match =        ( (id[3]  == acceptance_code_0[0] |
acceptance_mask_0[0] ) &
                        (id[4]  == acceptance_code_0[1] |
acceptance_mask_0[1] ) &
                        (id[5]  == acceptance_code_0[2] |
```

```
acceptance_mask_0[2] ) &
                        (id[6]  == acceptance_code_0[3] |
acceptance_mask_0[3] ) &
                        (id[7]  == acceptance_code_0[4] |
acceptance_mask_0[4] ) &
                        (id[8]  == acceptance_code_0[5] |
acceptance_mask_0[5] ) &
                        (id[9]  == acceptance_code_0[6] |
acceptance_mask_0[6] ) &
                        (id[10] == acceptance_code_0[7] |
acceptance_mask_0[7] )
                      );

// Working in extended mode. ID match for standard format (11-bit ID).
Using single filter.
assign match_sf_std = ( (id[3]  == acceptance_code_0[0] |
acceptance_mask_0[0] ) &
                        (id[4]  == acceptance_code_0[1] |
acceptance_mask_0[1] ) &
                        (id[5]  == acceptance_code_0[2] |
acceptance_mask_0[2] ) &
                        (id[6]  == acceptance_code_0[3] |
acceptance_mask_0[3] ) &
                        (id[7]  == acceptance_code_0[4] |
acceptance_mask_0[4] ) &
                        (id[8]  == acceptance_code_0[5] |
acceptance_mask_0[5] ) &
                        (id[9]  == acceptance_code_0[6] |
acceptance_mask_0[6] ) &
                        (id[10] == acceptance_code_0[7] |
acceptance_mask_0[7] ) &

                        (rtr1   == acceptance_code_1[4] |
acceptance_mask_1[4] ) &
                        (id[0]  == acceptance_code_1[5] |
acceptance_mask_1[5] ) &
                        (id[1]  == acceptance_code_1[6] |
acceptance_mask_1[6] ) &
                        (id[2]  == acceptance_code_1[7] |
acceptance_mask_1[7] ) &

                        (data0[0]  == acceptance_code_2[0] |
acceptance_mask_2[0] | no_byte0) &
                        (data0[1]  == acceptance_code_2[1] |
acceptance_mask_2[1] | no_byte0) &
                        (data0[2]  == acceptance_code_2[2] |
acceptance_mask_2[2] | no_byte0) &
                        (data0[3]  == acceptance_code_2[3] |
acceptance_mask_2[3] | no_byte0) &
                        (data0[4]  == acceptance_code_2[4] |
acceptance_mask_2[4] | no_byte0) &
                        (data0[5]  == acceptance_code_2[5] |
acceptance_mask_2[5] | no_byte0) &
                        (data0[6]  == acceptance_code_2[6] |
acceptance_mask_2[6] | no_byte0) &
                        (data0[7]  == acceptance_code_2[7] |
acceptance_mask_2[7] | no_byte0) &
```

```
                        (data1[0]  == acceptance_code_3[0] |
acceptance_mask_3[0] | no_byte1) &
                        (data1[1]  == acceptance_code_3[1] |
acceptance_mask_3[1] | no_byte1) &
                        (data1[2]  == acceptance_code_3[2] |
acceptance_mask_3[2] | no_byte1) &
                        (data1[3]  == acceptance_code_3[3] |
acceptance_mask_3[3] | no_byte1) &
                        (data1[4]  == acceptance_code_3[4] |
acceptance_mask_3[4] | no_byte1) &
                        (data1[5]  == acceptance_code_3[5] |
acceptance_mask_3[5] | no_byte1) &
                        (data1[6]  == acceptance_code_3[6] |
acceptance_mask_3[6] | no_byte1) &
                        (data1[7]  == acceptance_code_3[7] |
acceptance_mask_3[7] | no_byte1)
                       );

// Working in extended mode. ID match for extended format (29-bit ID).
Using single filter.
assign match_sf_ext = ( (id[21]  == acceptance_code_0[0] |
acceptance_mask_0[0] ) &
                        (id[22]  == acceptance_code_0[1] |
acceptance_mask_0[1] ) &
                        (id[23]  == acceptance_code_0[2] |
acceptance_mask_0[2] ) &
                        (id[24]  == acceptance_code_0[3] |
acceptance_mask_0[3] ) &
                        (id[25]  == acceptance_code_0[4] |
acceptance_mask_0[4] ) &
                        (id[26]  == acceptance_code_0[5] |
acceptance_mask_0[5] ) &
                        (id[27]  == acceptance_code_0[6] |
acceptance_mask_0[6] ) &
                        (id[28]  == acceptance_code_0[7] |
acceptance_mask_0[7] ) &

                        (id[13]  == acceptance_code_1[0] |
acceptance_mask_1[0] ) &
                        (id[14]  == acceptance_code_1[1] |
acceptance_mask_1[1] ) &
                        (id[15]  == acceptance_code_1[2] |
acceptance_mask_1[2] ) &
                        (id[16]  == acceptance_code_1[3] |
acceptance_mask_1[3] ) &
                        (id[17]  == acceptance_code_1[4] |
acceptance_mask_1[4] ) &
                        (id[18]  == acceptance_code_1[5] |
acceptance_mask_1[5] ) &
                        (id[19]  == acceptance_code_1[6] |
acceptance_mask_1[6] ) &
                        (id[20]  == acceptance_code_1[7] |
acceptance_mask_1[7] ) &

                        (id[5]  == acceptance_code_2[0] |
acceptance_mask_2[0] ) &
```

```
                          (id[6]  == acceptance_code_2[1] |
acceptance_mask_2[1] ) &
                          (id[7]  == acceptance_code_2[2] |
acceptance_mask_2[2] ) &
                          (id[8]  == acceptance_code_2[3] |
acceptance_mask_2[3] ) &
                          (id[9]  == acceptance_code_2[4] |
acceptance_mask_2[4] ) &
                          (id[10] == acceptance_code_2[5] |
acceptance_mask_2[5] ) &
                          (id[11] == acceptance_code_2[6] |
acceptance_mask_2[6] ) &
                          (id[12] == acceptance_code_2[7] |
acceptance_mask_2[7] ) &

                          (rtr2   == acceptance_code_3[2] |
acceptance_mask_3[2] ) &
                          (id[0]  == acceptance_code_3[3] |
acceptance_mask_3[3] ) &
                          (id[1]  == acceptance_code_3[4] |
acceptance_mask_3[4] ) &
                          (id[2]  == acceptance_code_3[5] |
acceptance_mask_3[5] ) &
                          (id[3]  == acceptance_code_3[6] |
acceptance_mask_3[6] ) &
                          (id[4]  == acceptance_code_3[7] |
acceptance_mask_3[7] )
);

// Working in extended mode. ID match for standard format (11-bit ID).
Using double filter.
assign match_df_std = (((id[3]  == acceptance_code_0[0] |
acceptance_mask_0[0] ) &
                          (id[4]  == acceptance_code_0[1] |
acceptance_mask_0[1] ) &
                          (id[5]  == acceptance_code_0[2] |
acceptance_mask_0[2] ) &
                          (id[6]  == acceptance_code_0[3] |
acceptance_mask_0[3] ) &
                          (id[7]  == acceptance_code_0[4] |
acceptance_mask_0[4] ) &
                          (id[8]  == acceptance_code_0[5] |
acceptance_mask_0[5] ) &
                          (id[9]  == acceptance_code_0[6] |
acceptance_mask_0[6] ) &
                          (id[10] == acceptance_code_0[7] |
acceptance_mask_0[7] ) &

                          (rtr1   == acceptance_code_1[4] |
acceptance_mask_1[4] ) &
                          (id[0]  == acceptance_code_1[5] |
acceptance_mask_1[5] ) &
                          (id[1]  == acceptance_code_1[6] |
acceptance_mask_1[6] ) &
                          (id[2]  == acceptance_code_1[7] |
acceptance_mask_1[7] ) &
```

```
                        (data0[0] == acceptance_code_3[0] |
acceptance_mask_3[0] | no_byte0) &
                        (data0[1] == acceptance_code_3[1] |
acceptance_mask_3[1] | no_byte0) &
                        (data0[2] == acceptance_code_3[2] |
acceptance_mask_3[2] | no_byte0) &
                        (data0[3] == acceptance_code_3[3] |
acceptance_mask_3[3] | no_byte0) &
                        (data0[4] == acceptance_code_1[4] |
acceptance_mask_1[4] | no_byte0) &
                        (data0[5] == acceptance_code_1[5] |
acceptance_mask_1[5] | no_byte0) &
                        (data0[6] == acceptance_code_1[6] |
acceptance_mask_1[6] | no_byte0) &
                        (data0[7] == acceptance_code_1[7] |
acceptance_mask_1[7] | no_byte0) )
                       |

                       ((id[3]  == acceptance_code_2[0] |
acceptance_mask_2[0] ) &
                        (id[4]  == acceptance_code_2[1] |
acceptance_mask_2[1] ) &
                        (id[5]  == acceptance_code_2[2] |
acceptance_mask_2[2] ) &
                        (id[6]  == acceptance_code_2[3] |
acceptance_mask_2[3] ) &
                        (id[7]  == acceptance_code_2[4] |
acceptance_mask_2[4] ) &
                        (id[8]  == acceptance_code_2[5] |
acceptance_mask_2[5] ) &
                        (id[9]  == acceptance_code_2[6] |
acceptance_mask_2[6] ) &
                        (id[10] == acceptance_code_2[7] |
acceptance_mask_2[7] ) &

                        (rtr1   == acceptance_code_3[4] |
acceptance_mask_3[4] ) &
                        (id[0]  == acceptance_code_3[5] |
acceptance_mask_3[5] ) &
                        (id[1]  == acceptance_code_3[6] |
acceptance_mask_3[6] ) &
                        (id[2]  == acceptance_code_3[7] |
acceptance_mask_3[7] ) )
                      );

// Working in extended mode. ID match for extended format (29-bit ID).
Using double filter.
assign match_df_ext = (((id[21]  == acceptance_code_0[0] |
acceptance_mask_0[0] ) &
                        (id[22]  == acceptance_code_0[1] |
acceptance_mask_0[1] ) &
                        (id[23]  == acceptance_code_0[2] |
acceptance_mask_0[2] ) &
                        (id[24]  == acceptance_code_0[3] |
acceptance_mask_0[3] ) &
                        (id[25]  == acceptance_code_0[4] |
acceptance_mask_0[4] ) &
```

```
                        (id[26]  == acceptance_code_0[5] |
acceptance_mask_0[5] ) &
                        (id[27]  == acceptance_code_0[6] |
acceptance_mask_0[6] ) &
                        (id[28]  == acceptance_code_0[7] |
acceptance_mask_0[7] ) &

                        (id[13]  == acceptance_code_1[0] |
acceptance_mask_1[0] ) &
                        (id[14]  == acceptance_code_1[1] |
acceptance_mask_1[1] ) &
                        (id[15]  == acceptance_code_1[2] |
acceptance_mask_1[2] ) &
                        (id[16]  == acceptance_code_1[3] |
acceptance_mask_1[3] ) &
                        (id[17]  == acceptance_code_1[4] |
acceptance_mask_1[4] ) &
                        (id[18]  == acceptance_code_1[5] |
acceptance_mask_1[5] ) &
                        (id[19]  == acceptance_code_1[6] |
acceptance_mask_1[6] ) &
                        (id[20]  == acceptance_code_1[7] |
acceptance_mask_1[7] ) )

                       |

                       ((id[21]  == acceptance_code_2[0] |
acceptance_mask_2[0] ) &
                        (id[22]  == acceptance_code_2[1] |
acceptance_mask_2[1] ) &
                        (id[23]  == acceptance_code_2[2] |
acceptance_mask_2[2] ) &
                        (id[24]  == acceptance_code_2[3] |
acceptance_mask_2[3] ) &
                        (id[25]  == acceptance_code_2[4] |
acceptance_mask_2[4] ) &
                        (id[26]  == acceptance_code_2[5] |
acceptance_mask_2[5] ) &
                        (id[27]  == acceptance_code_2[6] |
acceptance_mask_2[6] ) &
                        (id[28]  == acceptance_code_2[7] |
acceptance_mask_2[7] ) &

                        (id[13]  == acceptance_code_3[0] |
acceptance_mask_3[0] ) &
                        (id[14]  == acceptance_code_3[1] |
acceptance_mask_3[1] ) &
                        (id[15]  == acceptance_code_3[2] |
acceptance_mask_3[2] ) &
                        (id[16]  == acceptance_code_3[3] |
acceptance_mask_3[3] ) &
                        (id[17]  == acceptance_code_3[4] |
acceptance_mask_3[4] ) &
                        (id[18]  == acceptance_code_3[5] |
acceptance_mask_3[5] ) &
                        (id[19]  == acceptance_code_3[6] |
acceptance_mask_3[6] ) &
```

```
                        (id[20]   == acceptance_code_3[7] |
acceptance_mask_3[7] ) )                 );
// ID ok signal generation
always @ (posedge clk or posedge rst)
begin
  if (rst)
    id_ok <= 0;
  else if (go_rx_crc_lim)                    // sample_point is already
included in go_rx_crc_lim
    begin
      if (extended_mode)
        begin
          if (acceptance_filter_mode)        // dual filter
            begin
              if (ide)                       // extended frame message
                id_ok <=#Tp match_df_ext;
              else                           // standard frame message
                id_ok <=#Tp match_df_std;
            end
          else                               // single filter
            begin
              if (ide)                       // extended frame message
                id_ok <=#Tp match_sf_ext;
              else                           // standard frame message
                id_ok <=#Tp match_sf_std;
            end
        end
      else
        id_ok <=#Tp match;
    end
  else if (reset_mode | go_rx_inter | go_error_frame)        //
sample_point is already included in go_rx_inter
    id_ok <=#Tp 0;
end

endmodule
```

A.6: Receive FIFO HDL Code

The following code is the receive FIFO of the CAN controller. This is where the messages are stored until they are read by the host microcontroller.

```
// can_fifo.v
// synopsys translate_off
//`include "timescale.v"
`timescale 1ns/10ps
// synopsys translate_on
//`include "can_defines.v"

// Uncomment following line if you want to use WISHBONE interface.
Otherwise
// 8051 interface is used.
//`define   CAN_WISHBONE_IF
//`define Intel_mode

// Uncomment following line if you want to use CAN in Actel APA devices
(embedded memory used)
//define   ACTEL_APA_RAM

// Uncomment following line if you want to use CAN in Xilinx devices
(embedded memory used)
`define   XILINX_RAM

module can_fifo
(
  clk,
  rst,
  wr,
  data_in,
  addr,
  data_out,
  fifo_selected,
  reset_mode,
  release_buffer,
  extended_mode,
  overrun,
  info_empty,
  info_cnt
);

parameter Tp = 1;

input         clk;
input         rst;
input         wr;
input   [7:0] data_in;
input   [7:0] addr;
input         reset_mode;
input         release_buffer;
```

```
input         extended_mode;
input         fifo_selected;

output  [7:0] data_out;
output        overrun;
output        info_empty;
output  [6:0] info_cnt;

`ifdef ACTEL_APA_RAM
`else
`ifdef XILINX_RAM
`else
  reg     [7:0] fifo [0:63];
  reg     [3:0] length_fifo[0:63];
  reg           overrun_info[0:63];
`endif
`endif

reg     [5:0] rd_pointer;
reg     [5:0] wr_pointer;
reg     [5:0] read_address;
reg     [5:0] wr_info_pointer;
reg     [5:0] rd_info_pointer;
reg           wr_q;
reg     [3:0] len_cnt;
reg     [6:0] fifo_cnt;
reg     [6:0] info_cnt;
reg           latch_overrun;

wire    [3:0] length_info;
wire          write_length_info;
wire          fifo_empty;
wire          fifo_full;
wire          info_full;

assign write_length_info = (~wr) & wr_q;

// Delayed write signal
always @ (posedge clk or posedge rst)
begin
  if (rst)
    wr_q <= 0;
  else if (reset_mode)
    wr_q <=#Tp 0;
  else
    wr_q <=#Tp wr;
end

// length counter
always @ (posedge clk or posedge rst)
begin
  if (rst)
    len_cnt <= 0;
  else if (reset_mode | write_length_info)
    len_cnt <=#Tp 1'b0;
  else if (wr & (~fifo_full))
    len_cnt <=#Tp len_cnt + 1'b1;
```

```
end

// wr_info_pointer
always @ (posedge clk or posedge rst)
begin
  if (rst)
    wr_info_pointer <= 0;
  else if (reset_mode)
    wr_info_pointer <=#Tp 0;
  else if (write_length_info & (~info_full))
    wr_info_pointer <=#Tp wr_info_pointer + 1'b1;
end

// rd_info_pointer
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rd_info_pointer <= 0;
  else if (reset_mode)
    rd_info_pointer <=#Tp 0;
  else if (release_buffer & (~fifo_empty))
    rd_info_pointer <=#Tp rd_info_pointer + 1'b1;
end

// rd_pointer
always @ (posedge clk or posedge rst)
begin
  if (rst)
    rd_pointer <= 0;
  else if (release_buffer & (~fifo_empty))
    rd_pointer <=#Tp rd_pointer + length_info;
  else if (reset_mode)
    rd_pointer <=#Tp 0;
end

// wr_pointer
always @ (posedge clk or posedge rst)
begin
  if (rst)
    wr_pointer <= 0;
  else if (wr & (~fifo_full))
    wr_pointer <=#Tp wr_pointer + 1'b1;
  else if (reset_mode)
    wr_pointer <=#Tp 0;
end

// latch_overrun
always @ (posedge clk or posedge rst)
begin
  if (rst)
    latch_overrun <= 0;
  else if (reset_mode | write_length_info)
    latch_overrun <=#Tp 0;
  else if (wr & fifo_full)
    latch_overrun <=#Tp 1'b1;
end
```

```
// Counting data in fifo
always @ (posedge clk or posedge rst)
begin
  if (rst)
    fifo_cnt <= 0;
  else if (wr & (~release_buffer) & (~fifo_full))
    fifo_cnt <=#Tp fifo_cnt + 1'b1;
  else if ((~wr) & release_buffer & (~fifo_empty))
    fifo_cnt <=#Tp fifo_cnt - length_info;
  else if (wr & release_buffer & (~fifo_full) & (~fifo_empty))
    fifo_cnt <=#Tp fifo_cnt - length_info + 1'b1;
  else if (reset_mode)
    fifo_cnt <=#Tp 0;
end

assign fifo_full = fifo_cnt == 64;
assign fifo_empty = fifo_cnt == 0;

// Counting data in length_fifo and overrun_info fifo
always @ (posedge clk or posedge rst)
begin
  if (rst)
    info_cnt <= 0;
  else if (write_length_info ^ release_buffer)
    begin
      if (release_buffer & (~info_empty))
        info_cnt <=#Tp info_cnt - 1'b1;
      else if (write_length_info & (~info_full))
        info_cnt <=#Tp info_cnt + 1'b1;
    end
end

assign info_full = info_cnt == 64;
assign info_empty = info_cnt == 0;

// Selecting which address will be used for reading data from rx fifo
always @ (extended_mode or rd_pointer or addr)
begin
  if (extended_mode)      // extended mode
    begin
      read_address <= rd_pointer + (addr - 8'd16);
    end
  else                    // normal mode
    begin
      read_address <= rd_pointer + (addr - 8'd20);
    end
end

`ifdef ACTEL_APA_RAM
  actel_ram_64x8_sync fifo
  (
    .DO       (data_out),
    .RCLOCK   (clk),
    .WCLOCK   (clk),
    .DI       (data_in),
    .PO       (),                    // parity not used
```

```
    .WRB      (~(wr & (~fifo_full))),
    .RDB      (~fifo_selected),
    .WADDR    (wr_pointer),
    .RADDR    (read_address)
  );

  actel_ram_64x4_sync info_fifo
  (
    .DO       (length_info),
    .RCLOCK   (clk),
    .WCLOCK   (clk),
    .DI       (len_cnt),
    .PO       (),                          // parity not used
    .WRB      (~(write_length_info & (~info_full))),
    .RDB      (1'b0),                      // always enabled
    .WADDR    (wr_info_pointer),
    .RADDR    (rd_info_pointer)
  );


  actel_ram_64x1_sync overrun_fifo
  (
    .DO       (overrun),
    .RCLOCK   (clk),
    .WCLOCK   (clk),
    .DI       (latch_overrun | (wr & fifo_full)),
    .PO       (),                          // parity not used
    .WRB      (~(write_length_info & (~info_full))),
    .RDB      (1'b0),                      // always enabled
    .WADDR    (wr_info_pointer),
    .RADDR    (rd_info_pointer)
  );
`else
`ifdef XILINX_RAM

/*
  ram_64x8_sync fifo
  (
    .addra(wr_pointer),
    .addrb(read_address),
    .clka(clk),
    .clkb(clk),
    .dina(data_in),
    .doutb(data_out),
    .wea(wr & (~fifo_full))
  );
*/

  RAMB4_S8_S8 fifo
  (
    .DOA(),
    .DOB(data_out),
    .ADDRA({3'h0, wr_pointer}),
    .CLKA(clk),
    .DIA(data_in),
    .ENA(1'b1),
    .RSTA(1'b0),
```

```
    .WEA(wr & (~fifo_full)),
    .ADDRB({3'h0, read_address}),
    .CLKB(clk),
    .DIB(8'h0),
    .ENB(1'b1),
    .RSTB(1'b0),
    .WEB(1'b0)
  );

/*
  ram_64x4_sync info_fifo
  (
    .addra(wr_info_pointer),
    .addrb(rd_info_pointer),
    .clka(clk),
    .clkb(clk),
    .dina(len_cnt),
    .doutb(length_info),
    .wea(write_length_info & (~info_full))
  );
*/
  RAMB4_S4_S4 info_fifo
  (
    .DOA(),
    .DOB(length_info),
    .ADDRA({4'h0, wr_info_pointer}),
    .CLKA(clk),
    .DIA(len_cnt),
    .ENA(1'b1),
    .RSTA(1'b0),
    .WEA(write_length_info & (~info_full)),
    .ADDRB({4'h0, rd_info_pointer}),
    .CLKB(clk),
    .DIB(4'h0),
    .ENB(1'b1),
    .RSTB(1'b0),
    .WEB(1'b0)
  );

/*
  ram_64x1_sync overrun_fifo
  (
    .addra(wr_info_pointer),
    .addrb(rd_info_pointer),
    .clka(clk),
    .clkb(clk),
    .dina(latch_overrun | (wr & fifo_full)),
    .doutb(overrun),
    .wea(write_length_info & (~info_full))
  );*/

  RAMB4_S1_S1 overrun_fifo
  (
    .DOA(),
    .DOB(overrun),
    .ADDRA({6'h0, wr_info_pointer}),
    .CLKA(clk),
```

```
    .DIA(latch_overrun | (wr & fifo_full)),
    .ENA(1'b1),
    .RSTA(1'b0),
    .WEA(write_length_info & (~info_full)),
    .ADDRB({6'h0, rd_info_pointer}),
    .CLKB(clk),
    .DIB(1'h0),
    .ENB(1'b1),
    .RSTB(1'b0),
    .WEB(1'b0)
  );

`else
  // writing data to fifo
  always @ (posedge clk)
  begin
    if (wr & (~fifo_full))
      fifo[wr_pointer] <=#Tp data_in;
  end

  // reading from fifo
  assign data_out = fifo[read_address];


  // writing length_fifo
  always @ (posedge clk)
  begin
    if (write_length_info & (~info_full))
      length_fifo[wr_info_pointer] <=#Tp len_cnt;
  end

  // reading length_fifo
  assign length_info = length_fifo[rd_info_pointer];

  // overrun_info
  always @ (posedge clk)
  begin
    if (write_length_info & (~info_full))
      overrun_info[wr_info_pointer] <=#Tp latch_overrun | (wr &
fifo_full);
  end

  // reading overrun
  assign overrun = overrun_info[rd_info_pointer];


`endif
`endif

endmodule
```

A.7: Register Set HDL Code

The following sections of code define the register set within the CAN controller and define the format of the registers.

```
// can_registers.v
// synopsys translate_off
//`include "timescale.v"
`timescale 1ns/10ps
// synopsys translate_on
//`include "can_defines.v"

// Uncomment following line if you want to use WISHBONE interface.
Otherwise
// 8051 interface is used.
//`define   CAN_WISHBONE_IF
//`define Intel_mode

// Uncomment following line if you want to use CAN in Actel APA devices
(embedded memory used)
//define   ACTEL_APA_RAM

// Uncomment following line if you want to use CAN in Xilinx devices
(embedded memory used)
`define   XILINX_RAM

module can_registers
(
  clk,
  rst,
  cs,
  we,
  addr,
  data_in,
  data_out,
  irq,

  sample_point,
  transmitting,
  set_reset_mode,
  node_bus_off,
  error_status,
  rx_err_cnt,
  tx_err_cnt,
  transmit_status,
  receive_status,
  tx_successful,
  need_to_tx,
  overrun,
  info_empty,
  set_bus_error_irq,
  set_arbitration_lost_irq,
  arbitration_lost_capture,
  node_error_passive,
```

```
node_error_active,
rx_message_counter,
/* Mode register */
reset_mode,
listen_only_mode,
acceptance_filter_mode,
self_test_mode,

/* Command register */
clear_data_overrun,
release_buffer,
abort_tx,
tx_request,
self_rx_request,
single_shot_transmission,

/* Arbitration Lost Capture Register */
read_arbitration_lost_capture_reg,

/* Error Code Capture Register */
read_error_code_capture_reg,
error_capture_code,

/* Bus Timing 0 register */
baud_r_presc,
sync_jump_width,

/* Bus Timing 1 register */
time_segment1,
time_segment2,
triple_sampling,

/* Error Warning Limit register */
error_warning_limit,

/* Rx Error Counter register */
we_rx_err_cnt,

/* Tx Error Counter register */
we_tx_err_cnt,

/* Clock Divider register */
extended_mode,
clkout,

/* This section is for BASIC and EXTENDED mode */
/* Acceptance code register */
acceptance_code_0,

/* Acceptance mask register */
acceptance_mask_0,
/* End: This section is for BASIC and EXTENDED mode */

/* This section is for EXTENDED mode */
/* Acceptance code register */
acceptance_code_1,
acceptance_code_2,
```

```
  acceptance_code_3,

  /* Acceptance mask register */
  acceptance_mask_1,
  acceptance_mask_2,
  acceptance_mask_3,
  /* End: This section is for EXTENDED mode */

  /* Tx data registers. Holding identifier (basic mode), tx frame
information (extended mode) and data */
  tx_data_0,
  tx_data_1,
  tx_data_2,
  tx_data_3,
  tx_data_4,
  tx_data_5,
  tx_data_6,
  tx_data_7,
  tx_data_8,
  tx_data_9,
  tx_data_10,
  tx_data_11,
  tx_data_12
  /* End: Tx data registers */
);

parameter Tp = 1;

input         clk;
input         rst;
input         cs;
input         we;
input   [7:0] addr;
input   [7:0] data_in;

output  [7:0] data_out;
reg     [7:0] data_out;

output        irq;

input         sample_point;
input         transmitting;
input         set_reset_mode;
input         node_bus_off;
input         error_status;
input   [7:0] rx_err_cnt;
input   [7:0] tx_err_cnt;
input         transmit_status;
input         receive_status;
input         tx_successful;
input         need_to_tx;
input         overrun;
input         info_empty;
input         set_bus_error_irq;
input         set_arbitration_lost_irq;
input   [4:0] arbitration_lost_capture;
input         node_error_passive;
```

```
input         node_error_active;
input   [6:0] rx_message_counter;
/* Mode register */
output        reset_mode;
output        listen_only_mode;
output        acceptance_filter_mode;
output        self_test_mode;

/* Command register */
output        clear_data_overrun;
output        release_buffer;
output        abort_tx;
output        tx_request;
output        self_rx_request;
output        single_shot_transmission;

/* Arbitration Lost Capture Register */
output        read_arbitration_lost_capture_reg;

/* Error Code Capture Register */
output        read_error_code_capture_reg;
input   [7:0] error_capture_code;

/* Bus Timing 0 register */
output  [5:0] baud_r_presc;
output  [1:0] sync_jump_width;

/* Bus Timing 1 register */
output  [3:0] time_segment1;
output  [2:0] time_segment2;
output        triple_sampling;

/* Error Warning Limit register */
output  [7:0] error_warning_limit;

/* Rx Error Counter register */
output        we_rx_err_cnt;

/* Tx Error Counter register */
output        we_tx_err_cnt;

/* Clock Divider register */
output        extended_mode;
output        clkout;

/* This section is for BASIC and EXTENDED mode */
/* Acceptance code register */
output  [7:0] acceptance_code_0;

/* Acceptance mask register */
output  [7:0] acceptance_mask_0;

/* End: This section is for BASIC and EXTENDED mode */

/* This section is for EXTENDED mode */
/* Acceptance code register */
output  [7:0] acceptance_code_1;
```

```
output  [7:0] acceptance_code_2;
output  [7:0] acceptance_code_3;
/* Acceptance mask register */
output  [7:0] acceptance_mask_1;
output  [7:0] acceptance_mask_2;
output  [7:0] acceptance_mask_3;

/* End: This section is for EXTENDED mode */

/* Tx data registers. Holding identifier (basic mode), tx frame
information (extended mode) and data */
output  [7:0] tx_data_0;
output  [7:0] tx_data_1;
output  [7:0] tx_data_2;
output  [7:0] tx_data_3;
output  [7:0] tx_data_4;
output  [7:0] tx_data_5;
output  [7:0] tx_data_6;
output  [7:0] tx_data_7;
output  [7:0] tx_data_8;
output  [7:0] tx_data_9;
output  [7:0] tx_data_10;
output  [7:0] tx_data_11;
output  [7:0] tx_data_12;
/* End: Tx data registers */

reg           tx_successful_q;
reg           overrun_q;
reg           overrun_status;
reg           transmission_complete;
reg           transmit_buffer_status_q;
reg           receive_buffer_status;
reg           info_empty_q;
reg           error_status_q;
reg           node_bus_off_q;
reg           node_error_passive_q;
reg           transmit_buffer_status;
reg           single_shot_transmission;

reg     [7:0] data_out_tmp;

// Some interrupts exist in basic mode and in extended mode. Since they
are in different registers they need to be multiplexed.
wire          data_overrun_irq_en;
wire          error_warning_irq_en;
wire          transmit_irq_en;
wire          receive_irq_en;

wire    [7:0] irq_reg;

wire we_mode                  = cs & we & (addr == 8'd0);
wire we_command               = cs & we & (addr == 8'd1);
wire we_bus_timing_0          = cs & we & (addr == 8'd6) & reset_mode;
wire we_bus_timing_1          = cs & we & (addr == 8'd7) & reset_mode;
wire we_clock_divider_low     = cs & we & (addr == 8'd31);
wire we_clock_divider_hi      = we_clock_divider_low & reset_mode;
```

```
wire read = cs & (~we);
wire read_irq_reg = read & (addr == 8'd3);
assign read_arbitration_lost_capture_reg = read & extended_mode & (addr
== 8'd11);
assign read_error_code_capture_reg = read & extended_mode & (addr ==
8'd12);

/* This section is for BASIC and EXTENDED mode */
wire we_acceptance_code_0       = cs & we &   reset_mode  &
((~extended_mode) & (addr == 8'd4)  | extended_mode & (addr == 8'd16));
wire we_acceptance_mask_0       = cs & we &   reset_mode  &
((~extended_mode) & (addr == 8'd5)  | extended_mode & (addr == 8'd20));
wire we_tx_data_0               = cs & we & (~reset_mode) &
((~extended_mode) & (addr == 8'd10) | extended_mode & (addr == 8'd16))
& transmit_buffer_status;
wire we_tx_data_1               = cs & we & (~reset_mode) &
((~extended_mode) & (addr == 8'd11) | extended_mode & (addr == 8'd17))
& transmit_buffer_status;
wire we_tx_data_2               = cs & we & (~reset_mode) &
((~extended_mode) & (addr == 8'd12) | extended_mode & (addr == 8'd18))
& transmit_buffer_status;
wire we_tx_data_3               = cs & we & (~reset_mode) &
((~extended_mode) & (addr == 8'd13) | extended_mode & (addr == 8'd19))
& transmit_buffer_status;
wire we_tx_data_4               = cs & we & (~reset_mode) &
((~extended_mode) & (addr == 8'd14) | extended_mode & (addr == 8'd20))
& transmit_buffer_status;
wire we_tx_data_5               = cs & we & (~reset_mode) &
((~extended_mode) & (addr == 8'd15) | extended_mode & (addr == 8'd21))
& transmit_buffer_status;
wire we_tx_data_6               = cs & we & (~reset_mode) &
((~extended_mode) & (addr == 8'd16) | extended_mode & (addr == 8'd22))
& transmit_buffer_status;
wire we_tx_data_7               = cs & we & (~reset_mode) &
((~extended_mode) & (addr == 8'd17) | extended_mode & (addr == 8'd23))
& transmit_buffer_status;
wire we_tx_data_8               = cs & we & (~reset_mode) &
((~extended_mode) & (addr == 8'd18) | extended_mode & (addr == 8'd24))
& transmit_buffer_status;
wire we_tx_data_9               = cs & we & (~reset_mode) &
((~extended_mode) & (addr == 8'd19) | extended_mode & (addr == 8'd25))
& transmit_buffer_status;
wire we_tx_data_10              = cs & we & (~reset_mode) & (
extended_mode & (addr == 8'd26)) & transmit_buffer_status;
wire we_tx_data_11              = cs & we & (~reset_mode) & (
extended_mode & (addr == 8'd27)) & transmit_buffer_status;
wire we_tx_data_12              = cs & we & (~reset_mode) & (
extended_mode & (addr == 8'd28)) & transmit_buffer_status;
/* End: This section is for BASIC and EXTENDED mode */

/* This section is for EXTENDED mode */
wire   we_interrupt_enable      = cs & we & (addr == 8'd4)  &
extended_mode;
wire   we_error_warning_limit   = cs & we & (addr == 8'd13) &
reset_mode & extended_mode;
assign we_rx_err_cnt            = cs & we & (addr == 8'd14) &
reset_mode & extended_mode;
```

```
assign we_tx_err_cnt            = cs & we & (addr == 8'd15) &
reset_mode & extended_mode;
wire   we_acceptance_code_1     = cs & we & (addr == 8'd17) &
reset_mode & extended_mode;
wire   we_acceptance_code_2     = cs & we & (addr == 8'd18) &
reset_mode & extended_mode;
wire   we_acceptance_code_3     = cs & we & (addr == 8'd19) &
reset_mode & extended_mode;
wire   we_acceptance_mask_1     = cs & we & (addr == 8'd21) &
reset_mode & extended_mode;
wire   we_acceptance_mask_2     = cs & we & (addr == 8'd22) &
reset_mode & extended_mode;
wire   we_acceptance_mask_3     = cs & we & (addr == 8'd23) &
reset_mode & extended_mode;
/* End: This section is for EXTENDED mode */

always @ (posedge clk)
begin
  tx_successful_q           <=#Tp tx_successful;
  overrun_q                 <=#Tp overrun;
  transmit_buffer_status_q  <=#Tp transmit_buffer_status;
  info_empty_q              <=#Tp info_empty;
  error_status_q            <=#Tp error_status;
  node_bus_off_q            <=#Tp node_bus_off;
  node_error_passive_q      <=#Tp node_error_passive;
end

/* Mode register */
wire   [0:0] mode;
wire   [4:1] mode_basic;
wire   [3:1] mode_ext;
wire         receive_irq_en_basic;
wire         transmit_irq_en_basic;
wire         error_irq_en_basic;
wire         overrun_irq_en_basic;

can_register_asyn_syn #(1, 1'h1) MODE_REG0
( .data_in(data_in[0]),
  .data_out(mode[0]),
  .we(we_mode),
  .clk(clk),
  .rst(rst),
  .rst_sync(set_reset_mode)
);

can_register_asyn #(4, 0) MODE_REG_BASIC
( .data_in(data_in[4:1]),
  .data_out(mode_basic[4:1]),
  .we(we_mode),
  .clk(clk),
  .rst(rst)
);

can_register_asyn #(3, 0) MODE_REG_EXT
( .data_in(data_in[3:1]),
  .data_out(mode_ext[3:1]),
  .we(we_mode & reset_mode),
```

```
  .clk(clk),
  .rst(rst));
assign reset_mode             = mode[0];
assign listen_only_mode       = extended_mode & mode_ext[1];
assign self_test_mode         = extended_mode & mode_ext[2];
assign acceptance_filter_mode = extended_mode & mode_ext[3];

assign receive_irq_en_basic  = mode_basic[1];
assign transmit_irq_en_basic = mode_basic[2];
assign error_irq_en_basic    = mode_basic[3];
assign overrun_irq_en_basic  = mode_basic[4];
/* End Mode register */

/* Command register */
wire   [4:0] command;
can_register_asyn_syn #(1, 1'h0) COMMAND_REG0
( .data_in(data_in[0]),
  .data_out(command[0]),
  .we(we_command),
  .clk(clk),
  .rst(rst),
  .rst_sync(tx_request & sample_point)
);

can_register_asyn_syn #(1, 1'h0) COMMAND_REG1
( .data_in(data_in[1]),
  .data_out(command[1]),
  .we(we_command),
  .clk(clk),
  .rst(rst),
  .rst_sync(abort_tx & ~transmitting)
);

can_register_asyn_syn #(2, 2'h0) COMMAND_REG
( .data_in(data_in[3:2]),
  .data_out(command[3:2]),
  .we(we_command),
  .clk(clk),
  .rst(rst),
  .rst_sync(|command[3:2])
);

can_register_asyn_syn #(1, 1'h0) COMMAND_REG4
( .data_in(data_in[4]),
  .data_out(command[4]),
  .we(we_command),
  .clk(clk),
  .rst(rst),
  .rst_sync(tx_successful & (~tx_successful_q) | abort_tx)
);

assign self_rx_request = command[4] & (~command[0]);
assign clear_data_overrun = command[3];
assign release_buffer = command[2];
assign abort_tx = command[1] & (~command[0]) & (~command[4]);
assign tx_request = command[0] | command[4];
```

```
always @ (posedge clk or posedge rst)
begin
  if (rst)
    single_shot_transmission <= 1'b0;
  else if (we_command & data_in[1] & (data_in[1] | data_in[4]))
    single_shot_transmission <=#Tp 1'b1;
  else if (tx_successful & (~tx_successful_q))
    single_shot_transmission <=#Tp 1'b0;
end

/* End Command register */

/* Status register */

wire   [7:0] status;

assign status[7] = node_bus_off;
assign status[6] = error_status;
assign status[5] = transmit_status;
assign status[4] = receive_status;
assign status[3] = transmission_complete;
assign status[2] = transmit_buffer_status;
assign status[1] = overrun_status;
assign status[0] = receive_buffer_status;

always @ (posedge clk or posedge rst)
begin
  if (rst)
    transmission_complete <= 1'b1;
  else if (tx_successful & (~tx_successful_q) | abort_tx)
    transmission_complete <=#Tp 1'b1;
  else if (tx_request)
    transmission_complete <=#Tp 1'b0;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    transmit_buffer_status <= 1'b1;
  else if (tx_request)
    transmit_buffer_status <=#Tp 1'b0;
  else if (~need_to_tx)
    transmit_buffer_status <=#Tp 1'b1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    overrun_status <= 1'b0;
  else if (overrun & (~overrun_q))
    overrun_status <=#Tp 1'b1;
  else if (clear_data_overrun)
    overrun_status <=#Tp 1'b0;
end

always @ (posedge clk or posedge rst)
begin
```

```
  if (rst)
    receive_buffer_status <= 1'b0;
  else if (release_buffer)
    receive_buffer_status <=#Tp 1'b0;
  else if (~info_empty)
    receive_buffer_status <=#Tp 1'b1;
end

/* End Status register */

/* Interrupt Enable register (extended mode) */
wire   [7:0] irq_en_ext;
wire         bus_error_irq_en;
wire         arbitration_lost_irq_en;
wire         error_passive_irq_en;
wire         data_overrun_irq_en_ext;
wire         error_warning_irq_en_ext;
wire         transmit_irq_en_ext;
wire         receive_irq_en_ext;

can_register #(8) IRQ_EN_REG
( .data_in(data_in),
  .data_out(irq_en_ext),
  .we(we_interrupt_enable),
  .clk(clk)
);

assign bus_error_irq_en             = irq_en_ext[7];
assign arbitration_lost_irq_en      = irq_en_ext[6];
assign error_passive_irq_en         = irq_en_ext[5];
assign data_overrun_irq_en_ext      = irq_en_ext[3];
assign error_warning_irq_en_ext     = irq_en_ext[2];
assign transmit_irq_en_ext          = irq_en_ext[1];
assign receive_irq_en_ext           = irq_en_ext[0];
/* End Bus Timing 0 register */

/* Bus Timing 0 register */
wire   [7:0] bus_timing_0;
can_register #(8) BUS_TIMING_0_REG
( .data_in(data_in),
  .data_out(bus_timing_0),
  .we(we_bus_timing_0),
  .clk(clk)
);

assign baud_r_presc = bus_timing_0[5:0];
assign sync_jump_width = bus_timing_0[7:6];
/* End Bus Timing 0 register */

/* Bus Timing 1 register */
wire   [7:0] bus_timing_1;
can_register #(8) BUS_TIMING_1_REG
( .data_in(data_in),
  .data_out(bus_timing_1),
  .we(we_bus_timing_1),
  .clk(clk)
);
```

```
assign time_segment1 = bus_timing_1[3:0];
assign time_segment2 = bus_timing_1[6:4];
assign triple_sampling = bus_timing_1[7];
/* End Bus Timing 1 register */

/* Error Warning Limit register */
can_register_asyn #(8, 96) ERROR_WARNING_REG
( .data_in(data_in),
  .data_out(error_warning_limit),
  .we(we_error_warning_limit),
  .clk(clk),
  .rst(rst)
);
/* End Error Warning Limit register */

/* Clock Divider register */
wire   [7:0] clock_divider;
wire         clock_off;
wire   [2:0] cd;
reg    [2:0] clkout_div;
reg    [2:0] clkout_cnt;
reg          clkout_tmp;
//reg          clkout;

can_register #(1) CLOCK_DIVIDER_REG_7
( .data_in(data_in[7]),
  .data_out(clock_divider[7]),
  .we(we_clock_divider_hi),
  .clk(clk)
);

assign clock_divider[6:4] = 3'h0;

can_register #(1) CLOCK_DIVIDER_REG_3
( .data_in(data_in[3]),
  .data_out(clock_divider[3]),
  .we(we_clock_divider_hi),
  .clk(clk)
);

can_register #(3) CLOCK_DIVIDER_REG_LOW
( .data_in(data_in[2:0]),
  .data_out(clock_divider[2:0]),
  .we(we_clock_divider_low),
  .clk(clk)
);

assign extended_mode = clock_divider[7];
assign clock_off     = clock_divider[3];
assign cd[2:0]       = clock_divider[2:0];

always @ (cd)
begin
  case (cd)                          // synopsys_full_case
synopsys_paralel_case
    3'b000 : clkout_div <= 0;
```

```
    3'b001 : clkout_div <= 1;
    3'b010 : clkout_div <= 2;
    3'b011 : clkout_div <= 3;
    3'b100 : clkout_div <= 4;
    3'b101 : clkout_div <= 5;
    3'b110 : clkout_div <= 6;
    3'b111 : clkout_div <= 0;
  endcase
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    clkout_cnt <= 3'h0;
  else if (clkout_cnt == clkout_div)
    clkout_cnt <=#Tp 3'h0;
  else
    clkout_cnt <= clkout_cnt + 1'b1;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    clkout_tmp <= 1'b0;
  else if (clkout_cnt == clkout_div)
    clkout_tmp <=#Tp ~clkout_tmp;
end

/*
//always @ (cd or clk or clkout_tmp or clock_off)
always @ (cd or clkout_tmp or clock_off)
begin
  if (clock_off)
    clkout <=#Tp 1'b1;
//  else if (&cd)
//    clkout <=#Tp clk;
  else
    clkout <=#Tp clkout_tmp;
end
*/
assign clkout = clock_off ? 1'b1 : ((&cd)? clk : clkout_tmp);

/* End Clock Divider register */

/* This section is for BASIC and EXTENDED mode */

/* Acceptance code register */
can_register #(8) ACCEPTANCE_CODE_REG0
( .data_in(data_in),
  .data_out(acceptance_code_0),
  .we(we_acceptance_code_0),
  .clk(clk)
);
/* End: Acceptance code register */

/* Acceptance mask register */
can_register #(8) ACCEPTANCE_MASK_REG0
```

```
( .data_in(data_in),
  .data_out(acceptance_mask_0),
  .we(we_acceptance_mask_0),
  .clk(clk)
);
/* End: Acceptance mask register */
/* End: This section is for BASIC and EXTENDED mode */

/* Tx data 0 register. */
can_register #(8) TX_DATA_REG0
( .data_in(data_in),
  .data_out(tx_data_0),
  .we(we_tx_data_0),
  .clk(clk)
);
/* End: Tx data 0 register. */

/* Tx data 1 register. */
can_register #(8) TX_DATA_REG1
( .data_in(data_in),
  .data_out(tx_data_1),
  .we(we_tx_data_1),
  .clk(clk)
);
/* End: Tx data 1 register. */

/* Tx data 2 register. */
can_register #(8) TX_DATA_REG2
( .data_in(data_in),
  .data_out(tx_data_2),
  .we(we_tx_data_2),
  .clk(clk)
);
/* End: Tx data 2 register. */

/* Tx data 3 register. */
can_register #(8) TX_DATA_REG3
( .data_in(data_in),
  .data_out(tx_data_3),
  .we(we_tx_data_3),
  .clk(clk)
);
/* End: Tx data 3 register. */

/* Tx data 4 register. */
can_register #(8) TX_DATA_REG4
( .data_in(data_in),
  .data_out(tx_data_4),
  .we(we_tx_data_4),
  .clk(clk)
);
/* End: Tx data 4 register. */

/* Tx data 5 register. */
can_register #(8) TX_DATA_REG5
( .data_in(data_in),
  .data_out(tx_data_5),
```

```
  .we(we_tx_data_5),
  .clk(clk));
/* End: Tx data 5 register. */

/* Tx data 6 register. */
can_register #(8) TX_DATA_REG6
( .data_in(data_in),
  .data_out(tx_data_6),
  .we(we_tx_data_6),
  .clk(clk)
);
/* End: Tx data 6 register. */

/* Tx data 7 register. */
can_register #(8) TX_DATA_REG7
( .data_in(data_in),
  .data_out(tx_data_7),
  .we(we_tx_data_7),
  .clk(clk)
);
/* End: Tx data 7 register. */

/* Tx data 8 register. */
can_register #(8) TX_DATA_REG8
( .data_in(data_in),
  .data_out(tx_data_8),
  .we(we_tx_data_8),
  .clk(clk)
);
/* End: Tx data 8 register. */

/* Tx data 9 register. */
can_register #(8) TX_DATA_REG9
( .data_in(data_in),
  .data_out(tx_data_9),
  .we(we_tx_data_9),
  .clk(clk)
);
/* End: Tx data 9 register. */

/* Tx data 10 register. */
can_register #(8) TX_DATA_REG10
( .data_in(data_in),
  .data_out(tx_data_10),
  .we(we_tx_data_10),
  .clk(clk)
);
/* End: Tx data 10 register. */

/* Tx data 11 register. */
can_register #(8) TX_DATA_REG11
( .data_in(data_in),
  .data_out(tx_data_11),
  .we(we_tx_data_11),
  .clk(clk)
);
/* End: Tx data 11 register. */
```

```
/* Tx data 12 register. */
can_register #(8) TX_DATA_REG12
( .data_in(data_in),
  .data_out(tx_data_12),
  .we(we_tx_data_12),
  .clk(clk)
);
/* End: Tx data 12 register. */

/* This section is for EXTENDED mode */

/* Acceptance code register 1 */
can_register #(8) ACCEPTANCE_CODE_REG1
( .data_in(data_in),
  .data_out(acceptance_code_1),
  .we(we_acceptance_code_1),
  .clk(clk)
);
/* End: Acceptance code register */

/* Acceptance code register 2 */
can_register #(8) ACCEPTANCE_CODE_REG2
( .data_in(data_in),
  .data_out(acceptance_code_2),
  .we(we_acceptance_code_2),
  .clk(clk)
);
/* End: Acceptance code register */

/* Acceptance code register 3 */
can_register #(8) ACCEPTANCE_CODE_REG3
( .data_in(data_in),
  .data_out(acceptance_code_3),
  .we(we_acceptance_code_3),
  .clk(clk)
);
/* End: Acceptance code register */

/* Acceptance mask register 1 */
can_register #(8) ACCEPTANCE_MASK_REG1
( .data_in(data_in),
  .data_out(acceptance_mask_1),
  .we(we_acceptance_mask_1),
  .clk(clk)
);
/* End: Acceptance code register */

/* Acceptance mask register 2 */
can_register #(8) ACCEPTANCE_MASK_REG2
( .data_in(data_in),
  .data_out(acceptance_mask_2),
  .we(we_acceptance_mask_2),
  .clk(clk)
);
/* End: Acceptance code register */
```

```
/* Acceptance mask register 3 */
can_register #(8) ACCEPTANCE_MASK_REG3
( .data_in(data_in),
  .data_out(acceptance_mask_3),
  .we(we_acceptance_mask_3),
  .clk(clk)
);
/* End: Acceptance code register */

/* End: This section is for EXTENDED mode */

// Reading data from registers
always @ ( addr or read or extended_mode or mode or bus_timing_0 or
bus_timing_1 or clock_divider or
           acceptance_code_0 or acceptance_code_1 or acceptance_code_2
or acceptance_code_3 or
           acceptance_mask_0 or acceptance_mask_1 or acceptance_mask_2
or acceptance_mask_3 or
           reset_mode or tx_data_0 or tx_data_1 or tx_data_2 or
tx_data_3 or tx_data_4 or
           tx_data_5 or tx_data_6 or tx_data_7 or tx_data_8 or
tx_data_9 or status or
           error_warning_limit or rx_err_cnt or tx_err_cnt or
irq_en_ext or irq_reg or mode_ext or
           arbitration_lost_capture or rx_message_counter or mode_basic
or error_capture_code
         )
begin
  if(read)  // read
    begin
      if (extended_mode)     // EXTENDED mode (Different register map
depends on mode)
        begin
          case(addr)
            8'd0  :  data_out_tmp <= {4'b0000, mode_ext[3:1], mode[0]};
            8'd1  :  data_out_tmp <= 8'h0;
            8'd2  :  data_out_tmp <= status;
            8'd3  :  data_out_tmp <= irq_reg;
            8'd4  :  data_out_tmp <= irq_en_ext;
            8'd6  :  data_out_tmp <= bus_timing_0;
            8'd7  :  data_out_tmp <= bus_timing_1;
            8'd11 :  data_out_tmp <= {3'h0,
arbitration_lost_capture[4:0]};
            8'd12 :  data_out_tmp <= error_capture_code;
            8'd13 :  data_out_tmp <= error_warning_limit;
            8'd14 :  data_out_tmp <= rx_err_cnt;
            8'd15 :  data_out_tmp <= tx_err_cnt;
            8'd16 :  data_out_tmp <= acceptance_code_0;
            8'd17 :  data_out_tmp <= acceptance_code_1;
            8'd18 :  data_out_tmp <= acceptance_code_2;
            8'd19 :  data_out_tmp <= acceptance_code_3;
            8'd20 :  data_out_tmp <= acceptance_mask_0;
            8'd21 :  data_out_tmp <= acceptance_mask_1;
            8'd22 :  data_out_tmp <= acceptance_mask_2;
            8'd23 :  data_out_tmp <= acceptance_mask_3;
            8'd24 :  data_out_tmp <= 8'h0;
            8'd25 :  data_out_tmp <= 8'h0;
```

```
            8'd26 :  data_out_tmp <= 8'h0;
            8'd27 :  data_out_tmp <= 8'h0;
            8'd28 :  data_out_tmp <= 8'h0;
            8'd29 :  data_out_tmp <= {1'b0, rx_message_counter};
            8'd31 :  data_out_tmp <= clock_divider;

            default: data_out_tmp <= 8'h0;
          endcase
        end
      else                    // BASIC mode
        begin
          case(addr)
            8'd0  :  data_out_tmp <= {3'b001, mode_basic[4:1],
mode[0]};
            8'd1  :  data_out_tmp <= 8'hff;
            8'd2  :  data_out_tmp <= status;
            8'd3  :  data_out_tmp <= {4'hf, irq_reg[3:0]};
            8'd4  :  data_out_tmp <= reset_mode? acceptance_code_0 :
8'hff;
            8'd5  :  data_out_tmp <= reset_mode? acceptance_mask_0 :
8'hff;
            8'd6  :  data_out_tmp <= reset_mode? bus_timing_0 : 8'hff;
            8'd7  :  data_out_tmp <= reset_mode? bus_timing_1 : 8'hff;
            8'd10 :  data_out_tmp <= reset_mode? 8'hff : tx_data_0;
            8'd11 :  data_out_tmp <= reset_mode? 8'hff : tx_data_1;
            8'd12 :  data_out_tmp <= reset_mode? 8'hff : tx_data_2;
            8'd13 :  data_out_tmp <= reset_mode? 8'hff : tx_data_3;
            8'd14 :  data_out_tmp <= reset_mode? 8'hff : tx_data_4;
            8'd15 :  data_out_tmp <= reset_mode? 8'hff : tx_data_5;
            8'd16 :  data_out_tmp <= reset_mode? 8'hff : tx_data_6;
            8'd17 :  data_out_tmp <= reset_mode? 8'hff : tx_data_7;
            8'd18 :  data_out_tmp <= reset_mode? 8'hff : tx_data_8;
            8'd19 :  data_out_tmp <= reset_mode? 8'hff : tx_data_9;
            8'd31 :  data_out_tmp <= clock_divider;

            default: data_out_tmp <= 8'h0;
          endcase
        end
    end
  else
    data_out_tmp <= 8'h0;
end

always @ (posedge clk or posedge rst)
begin
  if (rst)
    data_out <= 0;
  else if (read)
    data_out <=#Tp data_out_tmp;
end

// Some interrupts exist in basic mode and in extended mode. Since they
are in different registers they need to be multiplexed.
assign data_overrun_irq_en  = extended_mode ? data_overrun_irq_en_ext
: overrun_irq_en_basic;
assign error_warning_irq_en = extended_mode ? error_warning_irq_en_ext
: error_irq_en_basic;
```

```
assign transmit_irq_en      = extended_mode ? transmit_irq_en_ext
: transmit_irq_en_basic;
assign receive_irq_en       = extended_mode ? receive_irq_en_ext
: receive_irq_en_basic;

reg data_overrun_irq;
always @ (posedge clk or posedge rst)
begin
  if (rst)
    data_overrun_irq <= 1'b0;
  else if (overrun & (~overrun_q) & data_overrun_irq_en)
    data_overrun_irq <=#Tp 1'b1;
  else if (read_irq_reg)
    data_overrun_irq <=#Tp 1'b0;
end

reg transmit_irq;
always @ (posedge clk or posedge rst)
begin
  if (rst)
    transmit_irq <= 1'b0;
  else if (transmit_buffer_status & (~transmit_buffer_status_q) &
transmit_irq_en)
    transmit_irq <=#Tp 1'b1;
  else if (read_irq_reg)
    transmit_irq <=#Tp 1'b0;
end

reg receive_irq;
always @ (posedge clk or posedge rst)
begin
  if (rst)
    receive_irq <= 1'b0;
  else if (release_buffer)
    receive_irq <=#Tp 1'b0;
  else if ((~info_empty) & (~receive_irq) & receive_irq_en)
    receive_irq <=#Tp 1'b1;
end

reg error_irq;
always @ (posedge clk or posedge rst)
begin
  if (rst)
    error_irq <= 1'b0;
  else if (((error_status ^ error_status_q) | (node_bus_off ^
node_bus_off_q)) & error_warning_irq_en)
    error_irq <=#Tp 1'b1;
  else if (read_irq_reg)
    error_irq <=#Tp 1'b0;
end

reg bus_error_irq;
always @ (posedge clk or posedge rst)
begin
  if (rst)
    bus_error_irq <= 1'b0;
  else if (set_bus_error_irq & bus_error_irq_en)
```

```
    bus_error_irq <=#Tp 1'b1;
  else if (read_irq_reg)
    bus_error_irq <=#Tp 1'b0;
end

reg arbitration_lost_irq;
always @ (posedge clk or posedge rst)
begin
  if (rst)
    arbitration_lost_irq <= 1'b0;
  else if (set_arbitration_lost_irq & arbitration_lost_irq_en)
    arbitration_lost_irq <=#Tp 1'b1;
  else if (read_irq_reg)
    arbitration_lost_irq <=#Tp 1'b0;
end

reg error_passive_irq;
always @ (posedge clk or posedge rst)
begin
  if (rst)
    error_passive_irq <= 1'b0;
  else if ((node_error_passive & (~node_error_passive_q) |
(~node_error_passive) & node_error_passive_q & node_error_active) &
error_passive_irq_en)
    error_passive_irq <=#Tp 1'b1;
  else if (read_irq_reg)
    error_passive_irq <=#Tp 1'b0;
end

assign irq_reg = {bus_error_irq, arbitration_lost_irq,
error_passive_irq, 1'b0, data_overrun_irq, error_irq, transmit_irq,
receive_irq};

assign irq = data_overrun_irq | transmit_irq | receive_irq | error_irq
| bus_error_irq | arbitration_lost_irq | error_passive_irq;

endmodule
```

```
// can_register.v
// synopsys translate_off
//`include "timescale.v"
`timescale 1ns/10ps
// synopsys translate_on
//`include "can_defines.v"

// Uncomment following line if you want to use WISHBONE interface.
Otherwise
// 8051 interface is used.
//`define   CAN_WISHBONE_IF
//`define Intel_mode

// Uncomment following line if you want to use CAN in Actel APA devices
(embedded memory used)
//define   ACTEL_APA_RAM

// Uncomment following line if you want to use CAN in Xilinx devices
(embedded memory used)
`define   XILINX_RAM


module can_register
( data_in,
  data_out,
  we,
  clk
);

parameter WIDTH = 8; // default parameter of the register width

input [WIDTH-1:0] data_in;
input             we;
input             clk;

output [WIDTH-1:0] data_out;
reg    [WIDTH-1:0] data_out;

always @ (posedge clk)
begin
  if (we)                         // write
    data_out<=#1 data_in;
end

endmodule
```

```
// can_register_asyn.v
// synopsys translate_off
//`include "timescale.v"
`timescale 1ns/10ps
// synopsys translate_on
//`include "can_defines.v"

// Uncomment following line if you want to use WISHBONE interface.
Otherwise
// 8051 interface is used.
//`define   CAN_WISHBONE_IF
//`define Intel_mode

// Uncomment following line if you want to use CAN in Actel APA devices
(embedded memory used)
//define   ACTEL_APA_RAM

// Uncomment following line if you want to use CAN in Xilinx devices
(embedded memory used)
`define   XILINX_RAM

module can_register_asyn
( data_in,
  data_out,
  we,
  clk,
  rst
);

parameter WIDTH = 8; // default parameter of the register width
parameter RESET_VALUE = 0;

input [WIDTH-1:0] data_in;
input             we;
input             clk;
input             rst;

output [WIDTH-1:0] data_out;
reg    [WIDTH-1:0] data_out;


always @ (posedge clk or posedge rst)
begin
  if (rst)                          // asynchronous reset
    data_out<=#1 RESET_VALUE;
  else if (we)                      // write
    data_out<=#1 data_in;
end

endmodule
```

```
// can_register_asyn_syn.v
// synopsys translate_off
//`include "timescale.v"
`timescale 1ns/10ps
// synopsys translate_on
//`include "can_defines.v"

// Uncomment following line if you want to use WISHBONE interface.
Otherwise
// 8051 interface is used.
//`define   CAN_WISHBONE_IF
//`define Intel_mode

// Uncomment following line if you want to use CAN in Actel APA devices
(embedded memory used)
//define   ACTEL_APA_RAM

// Uncomment following line if you want to use CAN in Xilinx devices
(embedded memory used)
`define   XILINX_RAM


module can_register_asyn_syn
( data_in,
  data_out,
  we,
  clk,
  rst,
  rst_sync
);

parameter WIDTH = 8; // default parameter of the register width
parameter RESET_VALUE = 0;

input [WIDTH-1:0] data_in;
input             we;
input             clk;
input             rst;
input             rst_sync;

output [WIDTH-1:0] data_out;
reg    [WIDTH-1:0] data_out;

always @ (posedge clk or posedge rst)
begin
  if(rst)
    data_out<=#1 RESET_VALUE;
  else if (rst_sync)                  // synchronous reset
    data_out<=#1 RESET_VALUE;
  else if (we)                        // write
    data_out<=#1 data_in;
end

endmodule
```

```
// can_register_syn.v
// synopsys translate_off
//`include "timescale.v"
`timescale 1ns/10ps
// synopsys translate_on
//`include "can_defines.v"

// Uncomment following line if you want to use WISHBONE interface.
Otherwise
// 8051 interface is used.
//`define   CAN_WISHBONE_IF
//`define Intel_mode

// Uncomment following line if you want to use CAN in Actel APA devices
(embedded memory used)
//define   ACTEL_APA_RAM

// Uncomment following line if you want to use CAN in Xilinx devices
(embedded memory used)
`define   XILINX_RAM


module can_register_syn
( data_in,
  data_out,
  we,
  clk,
  rst_sync
);

parameter WIDTH = 8; // default parameter of the register width
parameter RESET_VALUE = 0;

input [WIDTH-1:0] data_in;
input             we;
input             clk;
input             rst_sync;

output [WIDTH-1:0] data_out;
reg    [WIDTH-1:0] data_out;



always @ (posedge clk)
begin
  if (rst_sync)                        // synchronous reset
    data_out<=#1 RESET_VALUE;
  else if (we)                         // write
    data_out<=#1 data_in;
end

endmodule
```

A.8: Inverse Bit Order HDL Code

The following code simply inverts the bit order of the input data stream.

```
// can_ibo.v
// This module only inverts bit order
module can_ibo
(
  di,
  do
);

input   [7:0] di;
output  [7:0] do;

assign do[0] = di[7];
assign do[1] = di[6];
assign do[2] = di[5];
assign do[3] = di[4];
assign do[4] = di[3];
assign do[5] = di[2];
assign do[6] = di[1];
assign do[7] = di[0];

endmodule
```

Appendix B: C Code

The following appendix contains all of the C code that describes the processes of initializing the CAN controller, writing a message into the the transmit buffer, and reading received messages from the receive buffer.

B.1: Initialization and Message Transmission C Code

The following code is the initialization and message transmission procedure.

```
/* Author: Anthony Marino
   Date: July 8, 2003
   Objective: This is a program that will initialize the CAN controller
   and tell the controller to send a message */

void WriteData(unsigned char , char);
char ReadExData(unsigned char);


//Control Lines for communication to SJA1000
#define ALE 1
#define RD 2
#define WR 0
#define CS 0
#define Low 0
#define High 1

//Clock Divider register
#define CDR 0x48 //BasicCAN mode,clock off,comparator bypassed

//Acceptance Code register
#define ACR 0x00 //Allow all identifiers

//Acceptance Mask register
#define AMR 0xFF //Allow all identifiers

//Bus timing values for 24 MHz clock, 1Mb/s bit rate
//Bus timing register 0
#define BTR_0 0x00

//Bus timing register 1
#define BTR_1 0x18

//Output control register
#define OCR 0x1A

// Control register
```

```
//#define CntrR 0x01  //Enables reset mode

//Clear a register
#define clearbyte 0x00

//Don't care
#define dontcare 0xFF

void main(void)
{

int i;
i=0;

while (i<2000) {

//Begin Initialization of the controller

//Enter reset mode
WriteData(0x00,0x01);

//Set clock divider register
WriteData(0x08,CDR);

//Set acceptance code and mask
WriteData(0x04,ACR); //acceptance code
WriteData(0x05,AMR); //acceptance mask

//Set bus timing registers
WriteData(0x06, BTR_0);  //Bus timing register 0
WriteData(0x07,BTR_1);  //Bus timing register 1

//Set output control register
WriteData(0x08,OCR);

//Switch off reset mode
WriteData(0x00,clearbyte);

//End of Initialization


//Begin Message Transmission

//Write message to be transmitted into the transmit buffer
WriteData(0x0A,0xE8);  //identifier (10 to 3)
WriteData(0x0B,0x21);  //identifier (2 to 0), RTR, and DLC
WriteData(0x0C,0x12);  //data byte 1 = 0x12
WriteData(0x0D,clearbyte);
WriteData(0x0E,clearbyte);
WriteData(0x0F,clearbyte);
WriteData(0x10,clearbyte);
WriteData(0x11,clearbyte);
WriteData(0x12,clearbyte);
WriteData(0x13,clearbyte);

//Set transmit request bit in command register
WriteData(0x01,0x01);
```

```
i++;
      } //end of loop

//End of message transmission
}

//Read and Write function definitions

void WriteData(unsigned char Address, char Data)
{
WrPortI(SPCR, &SPCRShadow, 0x84);          // setup parallel port A as
an Output
BitWrPortI(PCDR, &PCDRShadow, High, RD);  // Set the RD line High
BitWrPortI(PCDR, &PCDRShadow, Low, CS);   // Set the CS low
BitWrPortI(PEDR, &PEDRShadow, High, ALE); // Set the ALE line high
WrPortI(PADR, &PADRShadow, Address);        // Set AD0 to AD7 to the
value stored in Address
BitWrPortI(PEDR, &PEDRShadow, Low, ALE);  // Set the ALE line low
BitWrPortI(PEDR, &PEDRShadow, Low, WR);   // Set the WR line Low
WrPortI(PADR, &PADRShadow, Data);           // Set AD0 to AD7 to the
value stored in Data
BitWrPortI(PEDR, &PEDRShadow, High, WR);  // Set the WR line high
BitWrPortI(PCDR, &PCDRShadow, High, CS);  // Set the CS High
}


char ReadExData(unsigned char Address)
{
char Data;
WrPortI(SPCR, &SPCRShadow, 0x84);          // setup parallel port A as
an Output
BitWrPortI(PEDR, &PEDRShadow, High, WR);  // Set the WR line high
BitWrPortI(PCDR, &PCDRShadow, Low, CS);   // Set the CS low
BitWrPortI(PEDR, &PEDRShadow, High, ALE); // Set the ALE line high
WrPortI( PADR, &PADRShadow, Address);       // Set AD0 to AD7 to the
value stored in Address
BitWrPortI(PEDR, &PEDRShadow, Low, ALE);  // Set the ALE line low
BitWrPortI(PCDR, &PCDRShadow, Low, RD);   // Set the RD line Low
WrPortI( SPCR, &SPCRShadow, 0x80);         // setup parallel port A as
an input
Data = RdPortI(PADR);                      // Read into PortA values at
AD0 to AD7
BitWrPortI(PCDR, &PCDRShadow, High, RD);  // Set the RD line high
BitWrPortI(PCDR, &PCDRShadow, High, CS);  // Set the CS High
return Data;
}
```

B.2: Initialization and Message Reception C Code

The following code is the initialization and message reception procedure. This program also turns on an LED if the data byte received is 01h which relates to light 1 turning on.

```
/* Author: Anthony Marino
   Date: July 8, 2003
   Objective: This is a program that will check the receive buffer and
   read messages from it */

void WriteData(unsigned char , char);
char ReadExData(unsigned char);


//Control Lines for communication to SJA1000
#define ALE 1
#define RD 2
#define WR 0
#define CS 0
#define Low 0
#define High 1

//Clock Divider register
#define CDR 0x48 //BasicCAN mode,clock off,comparator bypassed

//Acceptance Code register
#define ACR 0x00 //Allow all identifiers

//Acceptance Mask register
#define AMR 0xFF //Allow all identifiers

//Bus timing values for 24 MHz clock, 1Mb/s bit rate
//Bus timing register 0
#define BTR_0 0x00

//Bus timing register 1
#define BTR_1 0x18

//Output control register
#define OCR 0x1A

// Control register
//#define CntrR 0x01  //Enables reset mode

//Clear a register
#define clearbyte 0x00

//Don't care
#define dontcare 0xFF
```

```
void main(void)
{

//Begin Initialization of the controller

//Enter reset mode
WriteData(0x00,0x01);

//Set clock divider register
WriteData(0x08,CDR);

//Set acceptance code and mask
WriteData(0x04,ACR); //acceptance code
WriteData(0x05,AMR); //acceptance mask

//Set bus timing registers
WriteData(0x06, BTR_0);  //Bus timing register 0
WriteData(0x07,BTR_1);  //Bus timing register 1

//Set output control register
WriteData(0x08,OCR);

//Switch off reset mode
WriteData(0x00,clearbyte);

//End of Initialization


//Begin Message Reception

//Check status register

int i;
i=0;

while (i<1)
 {
   data = ReadExData(0x02);

   if (data(0)=1)  //if RBS high read from receive buffer
   {
     message = ReadExData(0x16);
     i++;
   }
 }

if (message = 0x01) //Turn on LED if data byte is 0x01 (light 1 on)
{
      int j;      // define an integer j to serve as a loop counter

      // Write 84 hex to slave port control register which initializes
parallel
      // port A as an output port (port A drives the LED's on the
development board).

      WrPortI(SPCR, &SPCRShadow, 0x84);
```

```
      // now write all ones to port A which sets outputs high and LED's
off

      WrPortI(PADR, &PADRShadow, 0xff);

      while(1) {  // begin an endless loop

            BitWrPortI(PADR, &PADRShadow, 1, 1);  // turn LED DS3 off

            for(j=0; j<32000; j++)
                  ; // time delay loop

            BitWrPortI(PADR, &PADRShadow, 0, 1);  // turn LED DS3 on

            for(j=0; j<25000; j++)
                  ; // time delay loop

      }     // end while loop
}

//Release receive buffer
WriteData(0x01,0x04);

//End of message reception
}

//Read and Write function definitions

void WriteData(unsigned char Address, char Data)
{
WrPortI(SPCR, &SPCRShadow, 0x84);          // setup parallel port A as
an Output
BitWrPortI(PCDR, &PCDRShadow, High, RD);  // Set the RD line High
BitWrPortI(PCDR, &PCDRShadow, Low, CS);   // Set the CS low
BitWrPortI(PEDR, &PEDRShadow, High, ALE); // Set the ALE line high
WrPortI(PADR, &PADRShadow, Address);        // Set AD0 to AD7 to the
value stored in Address
BitWrPortI(PEDR, &PEDRShadow, Low, ALE);  // Set the ALE line low
BitWrPortI(PEDR, &PEDRShadow, Low, WR);   // Set the WR line Low
WrPortI(PADR, &PADRShadow, Data);           // Set AD0 to AD7 to the
value stored in Data
BitWrPortI(PEDR, &PEDRShadow, High, WR);  // Set the WR line high
BitWrPortI(PCDR, &PCDRShadow, High, CS);  // Set the CS High
}


char ReadExData(unsigned char Address)
{
char Data;
WrPortI(SPCR, &SPCRShadow, 0x84);          // setup parallel port A as
an Output
BitWrPortI(PEDR, &PEDRShadow, High, WR);  // Set the WR line high
BitWrPortI(PCDR, &PCDRShadow, Low, CS);   // Set the CS low
BitWrPortI(PEDR, &PEDRShadow, High, ALE); // Set the ALE line high
WrPortI( PADR, &PADRShadow, Address);       // Set AD0 to AD7 to the
value stored in Address
```

```
BitWrPortI(PEDR, &PEDRShadow, Low, ALE); // Set the ALE line low
BitWrPortI(PCDR, &PCDRShadow, Low, RD);  // Set the RD line Low
WrPortI( SPCR, &SPCRShadow, 0x80);        // setup parallel port A as
an input
Data = RdPortI(PADR);                     // Read into PortA values at
AD0 to AD7
BitWrPortI(PCDR, &PCDRShadow, High, RD); // Set the RD line high
BitWrPortI(PCDR, &PCDRShadow, High, CS); // Set the CS High
return Data;
}
```