DESIGN OF AN ASIC CHIP FOR A CONTROLLER AREA NETWORK (CAN) PROTOCOL CONTROLLER

by

SREERAM KRISHNAMOORTHY, B.E.

A THESIS

IN

ELECTRICAL ENGINEERING

Submitted to the Graduate Faculty
of Texas Tech University in
Partial Fulfillment of
the Requirements for
the Degree of

MASTER OF SCIENCE

IN

ELECTRICAL ENGINEERING

Approved

Jon Gustav Bredeson
Chairperson of the Committee

Micheal Eugene Parten

Accepted

John Borrelli
Dean of the Graduate School

August, 2006

ACKNOWLEDGEMENTS

First and foremost I would like to express my sincere gratitude to my thesis advisor, Dr. Jon G. Bredeson for his support, patience, and encouragement throughout my graduate studies. This thesis work was enabled and sustained by his vision and ideas. Dr. Micheal E. Parten is due a special note of thanks for his unique way of inspiring students through clarity of thought, enthusiasm and caring. His technical excellence, unwavering faith and constant encouragement were very helpful and made this effort an enjoyable one. I consider it a privilege to have worked under his guidance.

I would like to thank the constant support I received from Mr. Sundar Raman and my colleagues at Quantum Think Pvt. Technologies Ltd. Bangalore. My interactions with them have always been of great value and have oriented my research in innovative and challenging directions. I would like to thank them for their confidence in my abilities and for the many opportunities they have offered me.

Special thanks are due to my friends for their emotional support and encouragement provided literally on a daily basis. The friendship of Karthik Ranganathan is much appreciated and has led to many interesting and good-spirited discussions relating to this research.

There are no words that adequately express my appreciation and gratitude to my parents, Bhagirathi and Krishnamoorthy for the wealth of love, support, guidance and practical assistance they have given me life-long. I am deeply indebted to my sisters and their family, for their constant encouragement and unflinching support. I owe them respect and thankfulness for the love and happiness they have always given me.

Last, but not the least, I would like to express my gratitude to everyone who, knowingly or otherwise, have provided support, encouragement and assistance and paved the way for my success so far.

TABLE OF CONTENTS

ABSTRACT

This thesis describes the design, simulation and ASIC implementation of a protocol controller for the Controller Area Network (CAN) 2.0A multi-master serial communication protocol. The CAN Controller designed will function as the interface between an application and the actual CAN bus. The RTL based design is implemented using Verilog HDL. Physical realizations of the design are obtained with Magma tools. Logic Equivalence is verified using Cadence Verplex. Simulations are made at each level to verify the implementations. The system was implemented in TSMC 0.13µm CMOS six metal process.

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

1.1 CAN History

Controller Area Network (CAN) is a shared serial bus communication protocol, originally developed in 1986 by Robert Bosch GmbH. The increasing number of distributed control systems in cars and the increasing wiring costs of car body electronics led to the birth of the "Automotive Serial Controller Area Network" protocol. Although initially developed for use in the automotive industry, its use quickly spread to a wide variety of embedded systems applications like industrial control where high-speed communication is required. With growing acceptance in various industries not necessarily related to the automotive industry, the protocol was renamed the Controller Area Network (CAN) [3].

1.2 CAN Overview

A CAN system sends messages using a serial bus network. With every node connected to every other node in the network, the need for a central controller for the entire network is made redundant. A block diagram of a typical CAN network used for communication is shown in Fig 1.1.

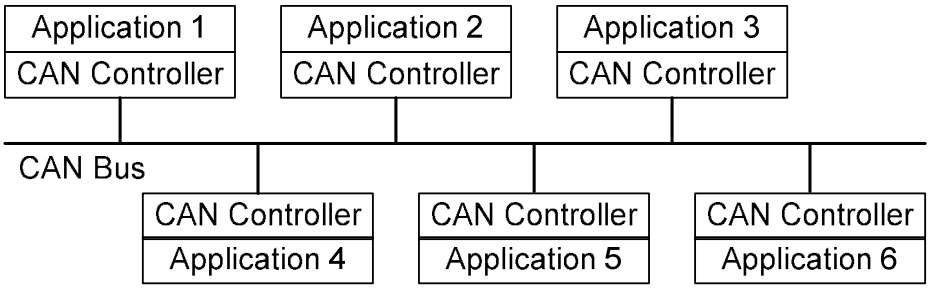

Fig 1.1 Communication using CAN

CAN being a multi-master communication protocol; every node in the system is equal to every other node. Any processor can send a message to any other processor. Even if some processor fails, the other systems in the machine will continue to work properly

and communicate with each other. Any node on the network that wants to transmit a message waits until the bus is free. Every message has an identifier, and every message is available to every other node in the network. The node selects those messages that are relevant and ignores the rest [3].

The CAN protocol was designed for short messages, no more than eight bytes long. The protocol never interrupts an ongoing transmission, but it assigns priorities to messages to prevent conflicts and to make sure that urgent messages are delivered first. The CAN protocol includes robust error detection and fault confinement mechanisms to make the traffic highly reliable [3]. The robustness of the network is further augmented by the fact that any faulty node can be removed from the network without affecting the rest of the network [2]. Unlike a traditional network such as USB or Ethernet, CAN does not send large blocks of data point-to-point from node A to node B under the supervision of a central bus master. In a CAN network, many short messages are broadcast to the entire network, which allows for data consistency in every node of the system [4].

The communication across a CAN bus starts with the application providing the CAN controller with the data to be transmitted. The CAN controller provides an interface between the application and the CAN bus. The function of the CAN controller is to convert the data provided by the application into a CAN message frame fit to be transmitted across the bus. A transceiver receives the serial input stream from the controller and converts it into a differential signal. These differential signals are transmitted physically across the CAN bus.

CAN transmits signals on the CAN bus which consists of a CAN-High and CAN-Low. These 2 buses carry signals of opposite polarity to overcome noise interruption. CAN uses a bit arbitration technique in which the priority of accessing the bus is determined by the 11-bit identifier. Due to the architecture a dominant bit will always override a recessive bit; the node with a lower identifier will have higher priority.

Every CAN Controller in a network will receive any message transmitted on the bus. Based on a filtering mechanism, the controller decides if the received information is relevant to the interfacing application and then proceeds to process the information [5].

1.3 Problem Being Addressed

The ultimate problem being addressed is the design and development of a one-chip solution for a CAN controller. The CAN controller chip would be interfaced to an application with a sensor and an actuator. The sensors provide analog signals for transmission across the CAN network. The analog signals sent by the application's sensor are received by an Analog-to-Digital Converter which performs the conversion. The digital output of the ADC is given to the microprocessor. The Microprocessor is an individual unit which processes the data and decides upon the transmission of a message through the CAN network. It also processes any messages received from the CAN network [5]. A block diagram of the CAN Controller is shown in Fig. 1.2.

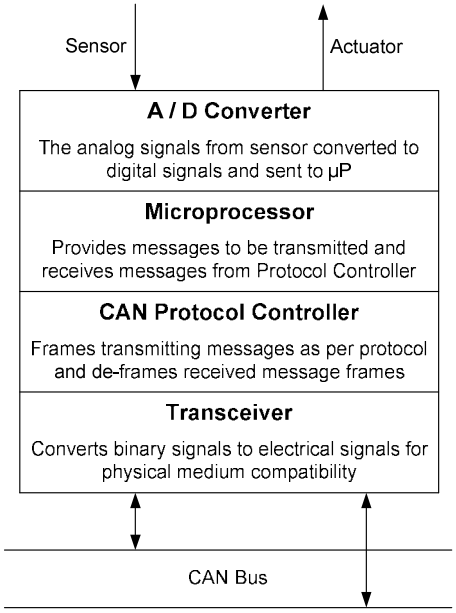

Fig. 1.2 CAN Controller On-Chip Solution

The CAN Protocol Controller receives unformatted message from the microprocessor, frames the messages as per the protocol specifications and also de-frames the received CAN message frames. The digital signals transmitted by the protocol controller are converted into electrical signals compatible with the CAN differential transmission medium by the CAN Transceiver which is also designed as a separate entity. The integration of these individual blocks would constitute the entire CAN Controller [5].

1.4 ASIC Implementation of CAN Protocol Controller

The interest in CAN is rapidly increasing due to an increase in the number of diverse applications being foreseen and also the availability of a number of devices in the market with the ability to interface to a CAN system. The increasing requirements of these applications, with respect to the levels of integration, re-usability of the code and the final price of the product, bring about the need for the development of systems-on-chip [6, 7]. The objective of this thesis is to design, implement and demonstrate the working of CAN protocol controller.

1.5 Motivation

The objective of this thesis is the ASIC implementation of the CAN Protocol Controller. The Register Transfer Level (RTL) model of the protocol controller is developed using Verilog HDL and the functional simulation of the model is obtained. The transmission and reception of the various CAN message frames are tested and verified with the modeling of a CAN network consisting of four CAN protocol controllers. The accurate behavior of the CAN protocol controller is further verified by inducing errors in the transmitted frames and checking the error detection mechanism of the protocol controllers.

The synthesized model of the CAN protocol controller is obtained and the Logic Equivalence verified. Dynamic simulations are performed on the Gate Level Netlist to further verify the functionality of the CAN Protocol Controller. Static Timing Analysis is performed to verify if the timing constraints specified for the system are satisfied. The design is placed and routed without any Design Rule Violations and the GDSII (Graphic Data System II) was extracted.

1.6 Organization of the Thesis

The remainder of the thesis is organized as follows. Chapter II gives an introduction to the various communication protocols and discusses CAN 2.0 A protocol at length. Chapter III provides a basic overview of the ASIC design process and the tools

made use of in this design. Chapter IV provides an introduction to the different implementations of a CAN controller and goes on to depict the basic building blocks of the CAN protocol controller developed for this thesis. It also provides a brief overview of the design constraints and the physical implementation of the CAN Protocol Controller. Chapter V covers the results obtained from the functional and Gate Level Simulation of the behavioral model, the logic equivalence verification, the timing analysis, the physical design and the DRC check of the CAN protocol controller. This chapter also covers the verification methods used to test the functionality of the CAN protocol controller. Chapter VI contains conclusions as well as considerations of ideas for further improvement.

CHAPTER 2
SERIAL COMMUNICATION USING CONTROLLER AREA NETWORK PROTOCOL

2.1 An Introduction to Communication Protocols

The factory floor is becoming an increasingly Internet-connected and networked environment. The increasing demand for communication has led to the development of various communication protocols. These protocols differ from one another based on the application for which they have been designed. In particular, the communication protocols addressing industrial applications comprise the "higher end" and the "lower end" protocols. The higher end protocols, also called factory bus, address the overall factory system information, while the lower end protocols, called the fieldbus, address the inter-processor communication as well as the sensor/actuator communication [2].

The cost advantage and advanced functionality offered by silicon technology coupled with the growing dependence on distributed systems and networking, have resulted in the demand for newer means of highly standardized communication in the fieldbus application area. The industry requires flexible control systems characterized by a high degree of standardization. The high degree of standardization leads to reusable solutions in terms of hardware and software modules that are easy to adapt to the varying needs and solutions in each individual field of application [2]. The number of fieldbus protocols has increased steadily over the last two decades, with Profibus, Interbus-S, P-Net, LON and FIP being among the earlier standards to be accepted.

The automobile industry developed several electronic systems to meet the growing demand for greater safety, comfort, convenience and compliance requirements for improved pollution control and reduced fuel consumption. As a result the complexity of these control systems and the need to exchange data among them required more and more hard-wired, dedicated signal lines. This prompted the replacement of the existing mode of wiring by a network architecture where all the nodes in the network communicate through a common bus. The Controller Area Network protocol developed by R. Bosch GmbH, Germany is ideally suited for communication through the above

network. Utilizing CAN, controllers, sensors, and actuators communicate with each other, in real-time, at speed of up to 1 MBits/s, over a two wire serial data bus [10].

The generic communication interface architecture essential to CAN protocol, models the Virtual Leveled Systems Architecture (VLSA) which is based on the concept of "Shared Variables". In this model, the individual tasks reside in the distributed controllers, each being responsible for a part of the overall control program [5]. An example of such a distributed system is illustrated in Fig. 2.1.

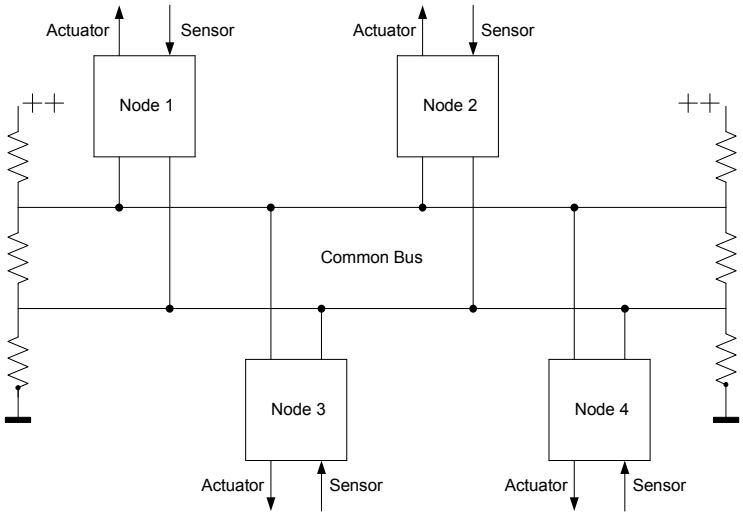

Fig. 2.1 Simplistic Representation of a Distributed Network

In the above illustrated figure all the nodes are connected to a common CAN bus. The nodes interact with the actual process through their sensors and actuators. To transmit messages on the bus the nodes make use of a dynamic, priority based arbitration mechanism. Every node in the network will receive any message transmitted on the bus, the nodes filter out the relevant messages based on a message filtering algorithm. Based on the received message the application might send out control signals to the device through the actuators.

2.2 CAN Protocol

The CAN protocol is an international standard defined in the ISO 11898 and ISO 11519. The difference between the two is in the physical layer, where ISO 11898 handles

high speed applications up to 1Mbit/second. ISO 11519 has an upper limit of 125kbit/second. The CAN protocol only comprises of the Data Link - composed of the Logical Link Control (LLC) sub layer and the Media Access Control (MAC) sub layer and the Physical Layer. ISO/OSI Layered Model for CAN Protocol is shown in Fig. 2.2.

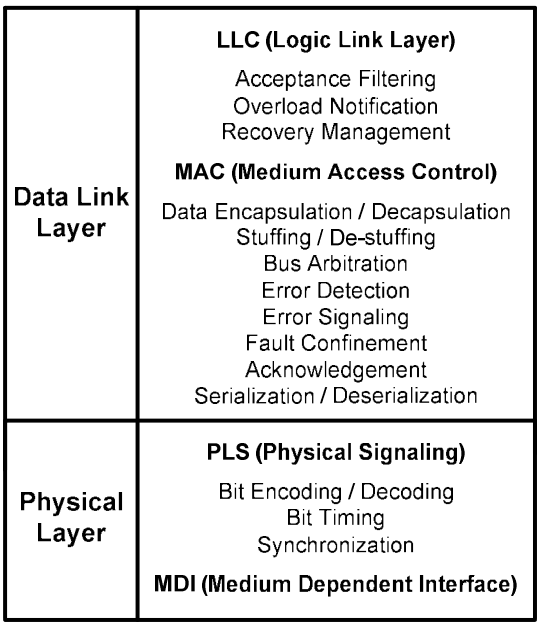

Fig. 2.2 CAN Protocol ISO/OSI Layered Model

The CAN Data Link Layer controls the message communication. The Data Link Layer builds data frames to hold data and control information. It also provides other services such as frame identification, bus arbitration, bit stuffing, error detection, error signaling, fault confinement and automatic retransmission of erroneous frames.

The CAN Physical Layer is responsible for transfer of data between different nodes in a given network; it defines how signals are transmitted and therefore deals with issues like encoding, timing and synchronization of the bit stream to be transferred [14].

The application layer is specified by higher-layer protocols such as CAL/CANOpen and CAN Kingdom, DeviceNet [7].

The CAN ISO standard discussed in this thesis is specified by ISO 11898-1 [6] which gives the data link layer and the physical signaling, and ISO 11898-2, which specifies the high-speed physical layer characteristics [6].

2.2.1 CAN Data Link Layer

2.2.1.1 Bus Arbitration

CAN is a protocol for short messages. Each transmission can carry 0 - 8 bytes of data. It uses CSMA/CD+AMP (Carrier Sense Multiple Access/Collision Detection with Arbitration on Message Priority). Thus the protocol is message oriented and each message has a specific priority according to which it gains access to the bus in case of simultaneous transmission. An ongoing transmission is never interrupted. During the bus idle state, any node can access the CAN bus. In the event of multiple accesses, the priority is decided by a method called "Non-Destructive Bit-wise Arbitration". Nondestructive means that the winner of the arbitration - the message with the higher priority - must continue transmitting the message without having to restart the message transmission from the beginning [9].

Arbitration is performed during the transmission of the identifier field. Each CAN message has an identifier which is of 11 bits. This identifier is the principle part of the CAN arbitration field, which is immediately after the Start bit. The identifier not only identifies the type of message but also indicates the priority of the message. During arbitration, each transmitting node monitors the bus state and compares the received bit with the transmitted bit. If a dominant bit is received when a recessive bit is transmitted then the node has lost arbitration. As soon as a node has lost the arbitration the node ceases to transmit, becoming a receiver of the ongoing message. At the end of transmission of the Arbitration Field, all nodes but one would have lost arbitration, and the highest priority message will get through unimpeded.

When the bus is free again the CAN Controller automatically makes a new attempt to transmit its message. Another round of arbitration is performed and the message with the highest priority gets through again leaving the messages with lower priority to content for the bus in the subsequent cycles. The CAN protocol requires that a specific identifier is sent only by one node this ensures that no two messages with the same identifier contend for bus access.

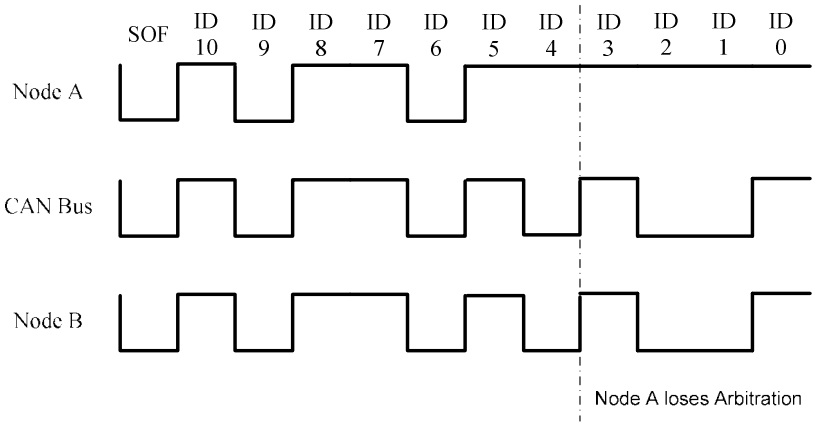

Fig. 2.3 CAN Bus Arbitration

Fig. 2.3 illustrates the method of non-destructive bit-wise arbitration of CAN. Two nodes A and B have a transmission request. The CAN bus bit indicates the bit value sampled from the bus, and the value is the Wired-AND value of the bits transmitted by nodes A and B [5].

In the above example both nodes start transmission as soon as the bus is free. Both Node A and Node B transmits the same bits till the 4th bit of the Identifier frame and neither node loses arbitration. During the 4th bit time Node A transmits a dominant bit, Node B transmits a recessive level. The recessive bit level is overwritten by the dominant bit level. At this point of time Node A loses the arbitration. Node B stops the transmission of any further bits and switches to receive mode, because the message that has won the arbitration must possibly be processed by this node [10].

2.2.1.2 CAN Message Transfer

CAN uses short fixed format messages of different but limited length - the maximum utility load is 94 bits. There is no explicit address in the messages; instead, the messages can be said to be content-addressed, that is, their content implicitly determines their address. The CAN protocol specifies four different frame types for communication: Data Frame, to transfer data; Remote Frame, to request data; Error Frame, to globally signal a detected error condition; Overload Frame, to extend delay time between subsequent frames [11].

2.2.1.3 Data Frame

The Data Frames is the most common message type, CAN systems, uses Data Frames to transmit data over the network. A Data frame is of fixed format with varying but limited data length. A Data Frame can hold up to eight bytes of data. A Data Frame is composed of eight different bit fields: Start of Frame; Arbitration Field; Control Field; Data Field; CRC field; Acknowledgement Field; End of Frame field and the Inter Frame Space.

The CAN protocol specifies two versions of the Frame, the Base Format and the Extended Format. The CAN specification 2.0A defines the Base Format CAN systems where the frames have standard 11 bit identifiers while the CAN specification 2.0B defines the Extended Format CAN systems where frames have 29 bit identifiers. The extended format is used in complex systems with heavy traffic where the number of messages created by transmitters on the network is greater than the number of possible ID codes that the CAN system could assign to them to make sure that each message is unique [3]. The Standard CAN 11-bit identifier provides for 2^{11}, or 2048 different message identifiers, while the Extended CAN 29-bit identifier provides for 2^{29}, or 537 million identifiers [4]. This design incorporates the CAN 2.0A specification and the following discussion will explain it in more detail.

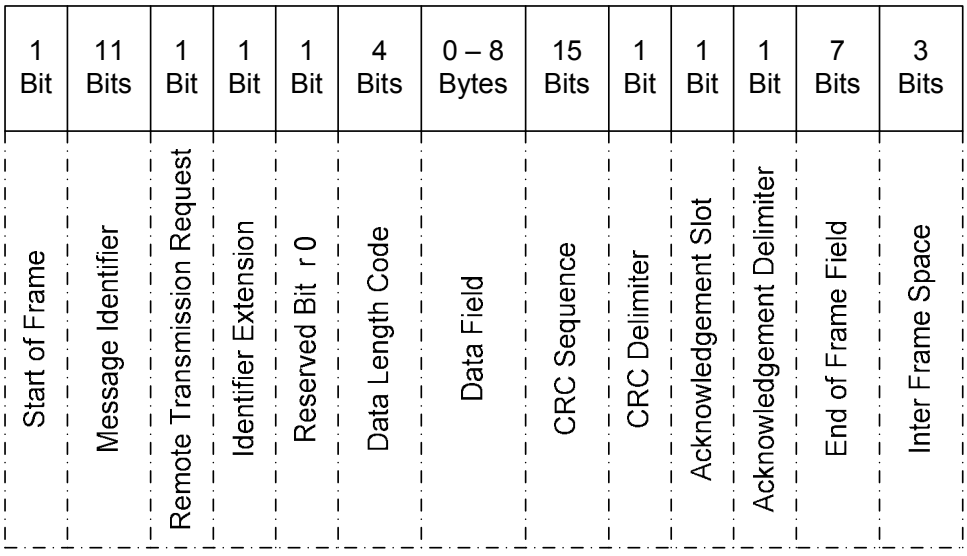

Fig. 2.4 CAN Data Frame

The various fields of the data as shown in Fig. 2.4 are explained as follows:

Start of Frame: A single dominant bit (logic 0) marks the start of the Data Frame. The CAN bus is in an idle (recessive) state prior to the transmission of this bit. All receivers on the bus use this bit to synchronize their clocks to the transmitter's clock.

Arbitration Field: The Arbitration Field is made up of 12 bits, the 11 bit Message Identifier and the Remote Transfer Request (RTR) bit. This field serves a dual purpose; it is used to indicate the logical address of the message and to determine which node has access to the CAN Bus.

The Message Identifier's bits are transferred in the order from ID10 to ID0 to satisfy the Non-Destructive bit-wise arbitration method of CAN. The lower the value of this field, the higher is the priority of the message. A message with all the bits of the identifier set to logic 0 will have priority over any other message in the network.

The RTR bit indicates whether the frame is a Data Frame or a Remote Frame. Within a Data Frame the RTR bit must be dominant. The RTR Bit is used by a receiver to request a remote transmitter to send its information. If this bit is set to 1 (= recessive) the frame contains no data field even if the data length code defines something other. A corresponding transmitter on receiving a Remote Transfer Request will initiate the transmission of a Data Frame. The request and the possible answer are two completely different frames on the bus. This means the response can be delayed due to messages with higher priorities [10].

In the event of a data frame and a remote frame being transmitted simultaneously with identical message identifiers, the RTR bit decides the priority. As a data frame has a dominant RTR bit, it is given priority over a remote frame with a recessive RTR bit [5].

Control Field: The control field is made up of 6 bits, the Identifier Extension (IE) bit, r0 and the 4 bit Data Length Code (DLC).

The first bit of the Control Filed is the IE; this bit is transmitted as a dominant bit in the standard format message to indicate that there are no more identifier bits in the message. The second bit is a reserved bit, transmitted as a dominant bit. The last four bits contain the data length code for the following data field. The DLC indicates the number

of bytes in the Data Field. The Admissible values of the DLC field are zero to eight. Since the field is four bits long values greater than eight can be indicated. If the value of the Data Length Code is greater than eight then it is assumed that the frame contains eight bytes [10].

Data Field: This field holds the application data to be transferred. This field is of variable length. The Data Field can contain zero to eight bytes of data. The data is transmitted with the MSB first.

CRC Field: The CRC Field is made up of the 15 bit CRC sequence and the recessive CRC Delimiter bit. The receiver uses the CRC sequence to check if the data bit sequence in the frame was corrupted during delivery.

Acknowledgement Field: This field is of 2 bits and contains the Acknowledgement Slot bit and the recessive Acknowledgement Delimiter bit. A transmitter transmits a recessive Acknowledgement bit. Any receiver on successful reception of the message sends out a dominant bit during this slot to acknowledge the successful reception of the message. A transmitter on reading back a dominant bit in the Acknowledgement Slot understands that at least one node if not all has received a complete and error free message.

End of Frame Field: The EOF marks the end of the CAN Frame [10]. The End of Frame is made up of a sequence of seven recessive bits.

Inter Frame Space: Every Data or Remote Frame is separated from the preceding frames by the Inter Frame Space. The IFS is made up of a sequence of recessive bits which extends for at least 3 bit durations. The bus may continue to remain idle after this, or a new frame will be indicated with the dominant Start-of-Frame bit. If one of the first two bits of this field is dominant then it is assumed an Overload Frame has been initiated.

2.2.1.4 Remote Frame

In a CAN network a node acting as a receiver for certain information may initiate transfer of the respective data using a Remote Transmission Request message. Remote requests are sent out on a regular basis to get updates from other nodes. This feature is very useful in situations where a network node that was temporarily off-line wishes to

reconnect to the network. The node might have information that is not up to date. In such a case the node need not wait until the corresponding transmitters send messages. The node can update the information that was previously available to it by scanning all necessary messages after sending out Remote Frames to the corresponding transmitters on the network.

The receiver via the Message Identifier of the Remote Frame addresses the node from which it needs the data. Every node in the network receives the RTR frame. Based on the identifier the application detects if it is the corresponding transmitter of the message. A corresponding transmitter on receiving a Remote Transfer Request will initiate the transmission of a Data Frame. The request and the possible answer are two completely different frames on the bus. This means the answer can be delayed due to messages with higher priorities in addition to the possible delay by the application. An advantage of this feature is that the message by the transmitter containing the application data is not only received by the requesting receiver but also by possible other receivers which are interested in this message. This mechanism ensures the data consistency of the network [10].

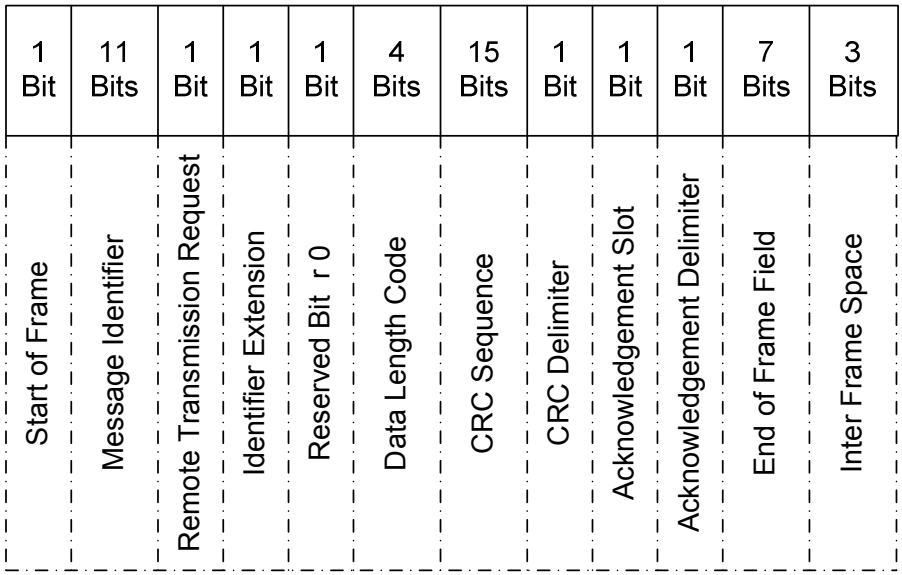

Fig. 2.5 CAN Remote Frame

The structure of the Remote Frame is similar to the Data Frame but for a few differences. Contrary to Data Frames, the RTR bit of Remote Frames is 'recessive'. There is no Data Field, independent of the values of the Data Length Code which may be signed any value within the admissible range 0 to 8 Bytes. The value is the Data Length Code of the corresponding Data Frame [1]. The structure of a RTR messages can be seen in Fig. 2.5. The descriptions of the various fields in a Remote Frame are the same as in a Data Frame, but for the differences explained earlier.

2.2.1.5 Error Frame

A CAN node on detecting an error, signals the presence of the error by sending out an Error Frame. The Error Frame can be sent during any point in a transmission and is always sent before a Data Frame or Remote Frame has completed successfully. The transmitter constantly monitors the bus while it is transmitting. When the transmitter detects the Error Frame, it aborts the current frame and prepares to resend the message once the bus becomes idle again [3]. This ensures data consistency throughout the network.

The error frame consists of 2 different fields. The first field is the Error Flag given by the superposition of Error Flags contributed by different CAN nodes. The second field is the Error Delimiter.

2.2.1.5.1 Error Flag

Error Flags are made up of a sequence of six consecutive bits of the same polarity. This sequence of bits violates the rule of bit stuffing or destroys a bit field requiring fixed form. As a consequence all other nodes also detect an error condition and likewise start transmission of an error flag.

The Error Flag is of two types:

- Active Error Flag: An Active Error flag is transmitted by an Error Active node. Active Error Flags are made up of a sequence of six consecutive dominant bits. This sequence violates the bit stuffing rule or destroys a bit field requiring fixed form. The total length of the Active Error Flag varies between a minimum of 6

and a maximum of 12 bits. This is due to the superposition of different error flags transmitted by individual nodes. Fig 2.6 gives the structure of Active Error Frame.

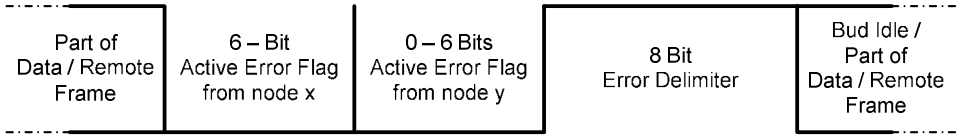

Fig. 2.6 Active Error Frame

- Passive Error Flag: A Passive Error Flag is transmitted by an Error Passive node. Passive Error Flags are made up of a sequence of six consecutive recessive bits. An Error Passive station waits for six consecutive bits of equal polarity, beginning at the start of the Passive Error Flag. The Passive Error Flag is complete when these six equal bits have been detected. Passive error flags initiated by a transmitter cause errors at the receivers when they start in a frame field which is encoded by the method of bit stuffing, because they then lead to stuff errors detected by the receivers. Passive error flags initiated by receivers are not able to prevail in any activity on the bus line [10]. Fig. 2.7 gives the structure of a Passive Error Frame.

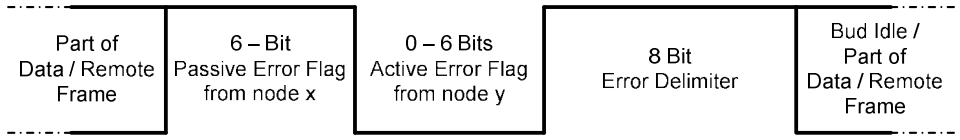

Fig. 2.7 Passive Error Frame

2.2.1.5.2 Error Delimiter

The Error Delimiter consists of 8 recessive bits. After transmission of an error flag, each node sends recessive bits and monitors the bus until it detects a recessive bit. It then starts transmitting 7 more recessive bits [1].

2.2.1.6 Overload Frame

Overload Frames are generated when a CAN controller encounters a situation in which it will not able to process a received message. The structure of the Overload frame is similar to that of an Active Error Flag. The only difference is that an Overload Frame is

initiated at the last bit of the EOF field or in the IFS field. To delay the transmission of the next message, a CAN node sends out an Overload Frame, all of the nodes on the network detect it and send out their own Frames, effectively stopping all message traffic on the CAN system. Then, all of the nodes listen for a sequence of eight recessive bits before they contend for Bus access. The maximum amount of time needed to recover in a CAN system after an Overload Frame is 31 bit times [3]. The structure of an Overload Frame is as shown in Fig. 2.8.

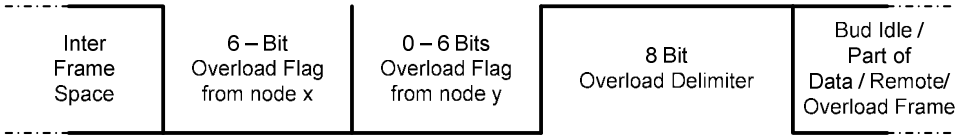

Fig 2.8 Overload Frame

2.2.1.7 Acknowledgement

All receivers check for the consistency of the message being received. On receiving an error free message till the end of the CRC Sequence the CAN controller sends out a dominant bit in the Acknowledgement slot. A transmitter on reading back a dominant bit in the Acknowledgement Slot understands that at least one node if not all has received a complete and error free message.

2.2.1.8 Bit Stuffing

The CAN protocol specifies the use of NRZ encoding, this might cause the networks individual clocks to go out of sync. To enable synchronization even in the NRZ format, a bit of complementary polarity is inserted after every five consecutive bits of the same polarity in the bit stream to be transmitted. This practice is called bit stuffing.

The frame segments Start of Frame, Arbitration Field, Control Field and CRC sequence are coded by the method of bit stuffing. The remaining bit fields of the data frame or remote frame the CRC delimiter, ACK field and EOF are fixed form and are not stuffed. The Error Frame and the Overload Frame are of fixed form as well, and are not coded with bit stuffing [10].

This method of bit stuffing is also significant keeping in mind the error signaling mechanism of the CAN protocol. The error frame transmitted to signal the presence of errors consists of an error of 6 consecutive bits of the same polarity [1]. Since bit stuffing is used, six consecutive bits of the same polarity appearing in the received bit stream is considered an error. Whenever an error frame is transmitted a stuff error or a form error is detected in all the participating nodes causing the transmitter to abort transmission and retransmit the message once the bus is idle. The bit stuffing mechanism is demonstrated in Fig. 2.9.

Original Bit Stream to be transmitted – 010111111011

Stuffed Bit Stream transmitted on bus – 0101111101011

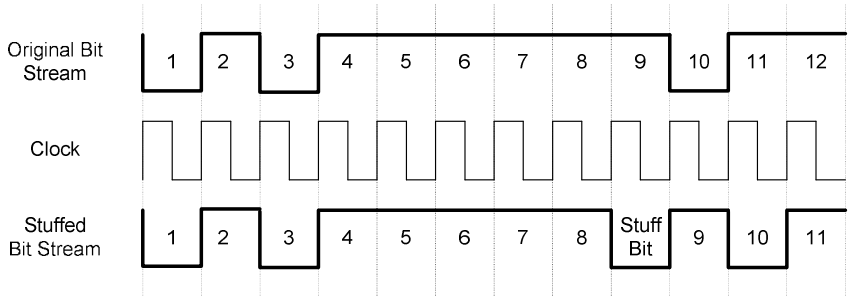

Fig. 2.9 Bit Stuffing

The receivers use the stuff bit to synchronize their clocks in case there is a long sequence of similar bits. During reception, the corresponding node automatically disregards the stuffed bit in the received bit stream to extract the information.

2.2.1.9 Error Process

The robustness of CAN may be attributed in part to its abundant error checking procedures. Every error that is detected by a network node will be notified to the rest of the network immediately. Unlike other field bus protocols which apply the principle of message confirmation, in the CAN protocol only a corrupted message is signaled by means of an error frame.

The robustness of CAN may be attributed in part to its extensive and sophisticated error checking procedures. Every error that is detected by a network node will be notified

to the rest of the network immediately. Unlike other field bus protocols, the CAN protocol does not apply the principle of message confirmation but instead signals errors immediately as they occur by means of an Error Frame [13]. There are several mechanisms in the CAN protocol, to detect errors and to prevent erroneous nodes from disabling all bus activities.

The CAN error process is divided into three parts: Error Detection, Error Handling, and Error Confinement.

2.2.1.9.1 Error Detection

The CAN protocol implements five mechanisms for error detection; three at the message level, two at the bit level.

Message level:

- Cyclic Redundancy Check (CRC): The CRC Field of every message holds the checksum of the preceding application data. This CRC sequence is transmitted in the CRC Field of the CAN frame. The receiving node performs a similar checksum of the received application data and performs a comparison to the received sequence. If the receiver detects a mismatch between the calculated and the received CRC sequence, then a CRC error has occurred. The receiver discards the message and transmits an Error Frame to request retransmission of the frame.
- Form Check: There are certain predefined bit values that must be transmitted recessive within a message. If a receiver detects a dominant bit in one of these positions a Form Error will be flagged. The bits checked are the CRC delimiter, ACK delimiter, and the EOF bits.
- Acknowledgment Check: Every node that transmits a message listens for an acknowledgement in the ACK slot. An Acknowledgement Error is flagged if a transmitter does not monitor a dominant bit during the ACK.

Bit level:

- Bit Monitoring: Every transmitting node monitors the transmitted bit level. If the

monitored bit level is different from the transmitted bit level, a Bit Error is flagged.

- Bit Stuff Monitoring: The bit stuffing rule specifies that a bit of opposite polarity is inserted after every five consecutive bits of the same polarity. If any receiving node detects six consecutive bits of the same polarity between Start of Frame and the CRC Delimiter, the bit stuffing rule has been violated. A stuff error occurs and an Error Frame is generated. The message is then repeated.

2.2.1.9.2 Error Handling

A CAN node on detecting an error condition, using any and all of the five mechanisms described above, signals the presence of an error by transmitting an Error Frame. The Error Frame transmitted by the node that detects the error, induces a Stuff or Form Error in other receiving nodes and a Bit Error in the transmitting node. This causes the transmitting node to abort transmission of the message. Since all the participating nodes receive the Error Frame the erroneous message is discarded thus ensuring the consistency of data throughout the network. The transmitter of the message will retransmit the message when the bus is free again. The retransmitted message still has to contend for arbitration on the bus.

2.2.1.9.3 Error Confinement

If a message fails an error frame is generated from the receiving nodes, causing the transmitting node to resend the message until it is received correctly. The probability of a network breakdown because of a local disturbance of one or a group of network nodes is very high. The CAN protocol makes use of a unique algorithm to prevent such a scenario. The algorithm is designed to automatically detect a faulty node and disconnect it from the network by removing the nodes transmit capability on reaching an error limit.

To implement the Error Confinement mechanism, CAN makes use of two error counters, one to keep track of transmit errors (Transmit Error Counter) and the other to keep track of receive errors (Receive Error Counter). A non-proportional method of

counting is implemented in CAN. The counter increment value in case of a transmission or reception error is higher than the counter decrement value for every successful transmission or reception. The state diagram for Error Confinement is given in Fig. 2.10.

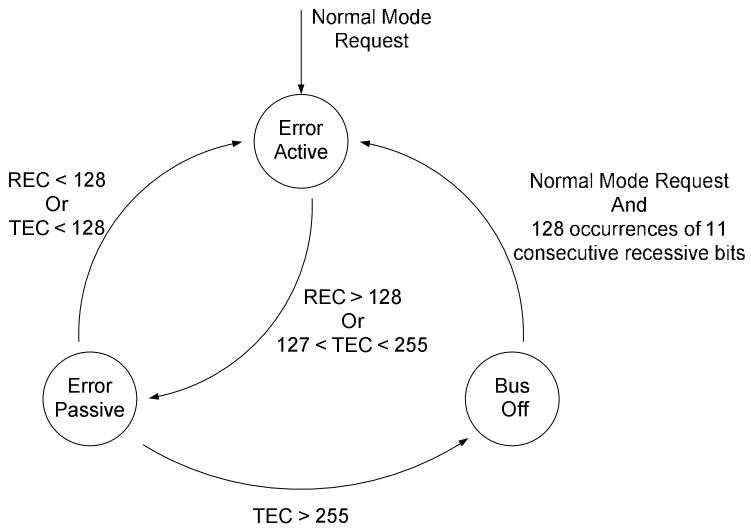

Fig. 2.10 CAN Error States

Based on these counters, a node can be in one of three states:

- Error Active: This is the normal operating state of the CAN node. In this state the node can transmit and receive messages. In the event of an error the node transmits an Active Error Frame. The node is in Error Active state if the values of both the REC and TEC are less than or equal to 127.
- Error Passive: In this state the node can take part in normal bus communication. The node transmits a Passive Error Frame to signal the presence of errors. A transmitting node in Error Passive state has to wait for an extra 8 bit bus idle period following the Inter Frame Space before it contends for Bus access. A node enters the Error Passive state when one of its counter values exceeds 127. The node can still return to the Error Active state when both the counters value is less than 128.
- Bus Off: In the Bus Off state the nodes participation is restricted to receiving messages. The node can no longer transmit messages. A node enters the Bus Off

state if the Transmit Error Count of a node exceeds 255. No messages can be transmitted until the error counters are reset by the host microcontroller or processor.

The Error Counts are modified as per the rules specified in the BOSCH, CAN Specification Version 2.0. More than one rule may apply during a given message transfer

1. When a receiver detects an error, the REC will be incremented by 1, except when the detected error is a Bit Error during the transmission of an Active Error Flag or an Overload Flag. When a transceiver detects an error, the REC will be increased by 1, except when the detected error is a bit error during the sending of an active error flag.
2. A receiver on detecting a dominant bit as the first bit after sending an Error Flag increments the REC by 8.
3. When a transmitter sends an Error Flag the TEC is incremented by 8. The TEC remains unchanged, under the following exceptions:
   - Exception 1:

     If an Error Passive transmitter detects an Acknowledgement Error because of not detecting an Acknowledgement and does not detect a dominant bit while sending its Passive Error Flag.
   - Exception 2:

     If the transmitter detects a Stuff Error and sends an error flag because a recessive stuff bit transmitted recessive is monitored as a dominant bit during arbitration.
4. The TEC is incremented by 8 if a transmitter detects a Bit Error while sending an Active Error Flag or an Overload Flag.
5. The REC is incremented by 8 if a receiver detects a Bit Error while sending an Active Error Flag or an Overload Flag.
6. A node can tolerate up to 7 consecutive dominant bits after transmitting an Error Flag or Overload Flag. Every Transmitter increments its TEC by 8 and every receiver increments its REC by 8, after detecting the 14th consecutive dominant

bit (in case of an Active Error Flag or an Overload Flag) or after detecting the 8th consecutive dominant bit following a Passive Error Flag, and after each sequence of additional eight consecutive dominant bits.

7. The TEC is decremented by 1 (unless it is already 0) after the successful transmission of a message.
8. After the successful reception of a message and the successful transmission of the Acknowledgement, the REC is decremented by 1, if its value is between 1 and 127. If the REC is 0, it stays 0, and if it is greater than 127 it is set to a value between 119 and 127.

2.2.1.10 Acceptance Filtering and Message Validation

Data messages transmitted from any node on a CAN bus do not contain addresses of either the transmitting node, or of any intended receiving node. Instead, the content of the message is labeled by an identifier that is unique throughout the network. The content of the Arbitration Field is commonly used as a message identifier. All other nodes on the network receive the message and each performs an acceptance test on the identifier to determine if the message, and thus its content, is relevant to that particular node. The Acceptance Filter, based on the mask and code parameters of the host application, filters the received messages and stores only those required by the host application. Thus the host application ensures only relevant messages are processed and the rest are ignored.

2.2.1.11 Overload Notification

A CAN controller has two Receive Message Buffers to store the received data. This enables the host application to process one message while it is receiving another message. In case both the buffers are full, the controller will not accept any further messages until one of the buffers is cleared by the host microcontroller. To ensure that none of the messages transmitted during this period is lost, the transmission of the next message is delayed by sending out an Overload Frame. All the nodes on the network detect the Overload Frame and send out their own frame, effectively stopping all message

traffic on the CAN system. Then, all of the nodes listen for a sequence of eight recessive bits before they contend for Bus access. The delay produced by the Overload Frame is normally sufficient for the host application to clear one of the buffers. A maximum of two Overload Frames can be sent in succession to delay the transmission of a message on the Bus. An Overload Frame is initiated only at the last bit of the EOF field or in the IFS field.

2.3 CAN Physical Layer

The CAN Physical Layer is responsible for the physical connection between two nodes in a network and the actual transmission of electrical impulses. The CAN Physical Layer defines the bit representation, bit encoding, bit timing and synchronization and the bit rate and bus length scheme on the CAN bus.

The CAN Physical Layer translates the data provided by the transmitter's Data Link layer into an electronic signal. On the receiving end, the Physical Layer translates the electronic signals back into a data format and passes it on to the Data Link layer. There are several different physical layers, the most common type of is the one defined by the CAN standard, part ISO 11898-2. It is also referred to as "high-speed CAN". A simplistic representation of the CAN wiring diagram is shown in Fig. 2.11.

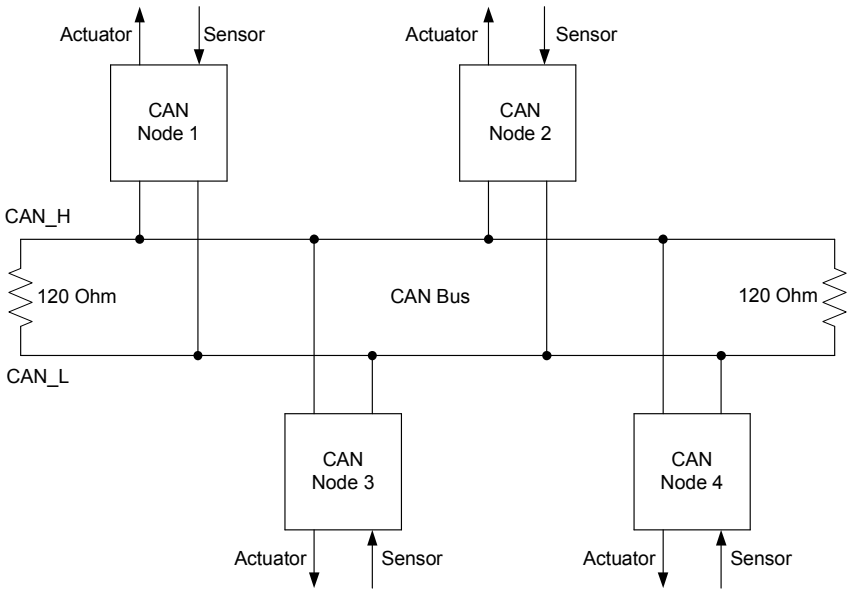

Fig. 2.11 CAN Wiring Diagram

High-speed CAN is made up of two wires arranged as a differential pair. The two wires are named CAN_H and CAN_L. Since the wires operate in the differential mode they carry inverted voltages, this provides excellent immunity from external electrical interference. The Bus termination is done using a resistor of 120 Ohms at each end of the bus. The termination removes the signal reflections at the end of the bus and ensures the bus gets correct DC levels.

2.3.1 Bit Representation

There are two logical bit representations used in the CAN protocol. CAN signals use the terms recessive and dominant to describe the state of the bus. The CAN protocol defines logic '0' as a dominant bus state and logic '1' as a recessive bus state. These correspond to certain electrical levels which depend on the Physical Layer used.

The modules are connected to the CAN bus in a wired-AND fashion. A recessive bit appears on the CAN Bus only if all the nodes on the network transmit a recessive bit during that bit-time. Since dominant bits always overwrite recessive bits, even if one node were to transmit a dominant bit during that bit time, then the whole bus is in the dominant state regardless of the number of nodes transmitting a recessive state. The concept of dominant and recessive bus states is an important concept in the discussion of CAN Bus Arbitration.

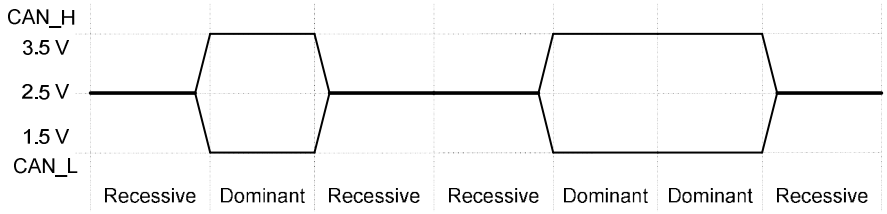

Fig. 2.12 CAN Dominant & Recessive States

The bit representation followed in CAN is shown in Fig. 2.12. For a two wire High-speed CAN Bus the recessive bus state "logic 1"occurs when the CAN_L and CAN_H lines are at the same potential (CAN_L = CAN_H = 2.5V), and the dominant bus state "logic 0" occurs when there is a difference in potential (CAN_L = 1.5V and CAN_H = 3.5V). The voltage level on the CAN bus is recessive when the bus is idle [3].

2.3.2 Bit Encoding

The CAN protocol specifies the use of Non-Return-to-Zero (NRZ) encoding. In the Non-return to zero encoding the logic level of the bit remains the same throughout the bit duration. If a frame has a string of '1's or '0's, the signal will remain constant for as many bit times as necessary. The disadvantage of NRZ is that there is no easy way to tell where each bit starts or ends when there are two or more '1's or '0's in a row. The only way to know where a bit starts or ends is for the receiver to have a clocking source that is identical to the transmitter so that it can decipher the bit stream. To enable clock synchronization among nodes in a CAN network, CAN uses the method of bit-stuffing [3].

2.3.3 Bit Timing and Synchronization

The Controller Area Network protocol uses synchronous data transmission to make the data transmissions more efficient. A major drawback of the synchronous transmission system is the absence of regular reference points for the node to perform bit synchronization. In a synchronous transmission system only one reference point is available at the start of the message. As it is difficult to keep any two clocks synchronized over a long period of time without some sort of reference point, CAN makes use of a sophisticated method of bit synchronization to overcome this drawback. In this method of synchronization every node in a network is continuously resynchronized. This ensures that all the nodes in a network are synchronized to function at the same transmission rate.

Bit rate is given by the number of bits that pass a given point in a network per second. The nominal bit rate is the number of bits per second transmitted by an ideal transmitter with no resynchronization. Whether transmitting or receiving, all nodes on the network must have the same nominal bit rate.

The nominal bit time is given by the inverse of the nominal bit rate. The nominal bit time is the amount of time needed to transmit a single bit across the network. CAN systems use the bit time to make sure that the nodes sample the bus at the appropriate time to determine whether the bus is in a recessive or dominant state. To accomplish this, the nominal bit time is broken into segments of four non-overlapping time segments - the

synchronization segment, the propagation segment and the phase segments one and two. Each segment consists of one or more time quanta [3]. The partition of the bit time is shown in Fig. 2.13.

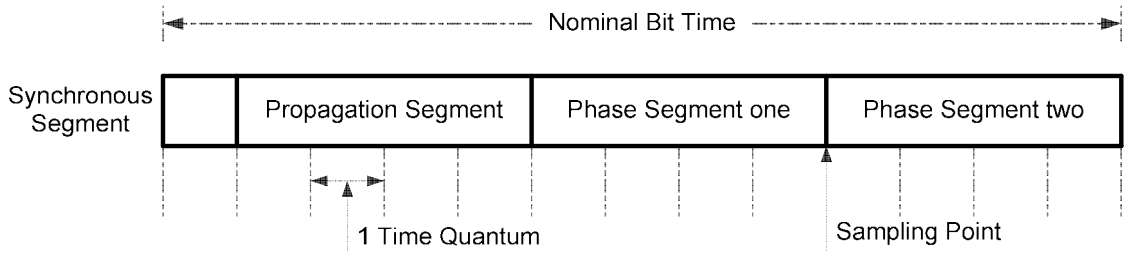

Fig. 2.13 CAN Bit Time

- The Synchronization Segment (Sync_Seg) is that part of the bit time where the signal transition is expected to occur. The Synchronization Segment is always fixed at one time quanta.
- Propagation Segment (Prop_Seg) is that part of the bit time intended to compensate for the physical delay in the bus lines. The Propagation Segment size should at least be equal to twice the time required for a bit to reach the farthest node. This is programmed to be 1-8 time quanta long.
- Phase segment one (Ph_Seg1) is a buffer segment that is lengthened if a recessive to dominant transition occurs during the Propagation Segment. The Phase segment one is lengthened such that the distance from the edge to the sampling point is the same as it would have been from the Synchronous Segment to the sample point if no edge had occurred. This is programmed to be 1-8 time quanta long.
- Phase segment two (Ph_Seg2) is also a buffer segment that is shortened if a recessive to dominant transition occurs during the Phase segment two. The Phase segment two is shortened such that the distance from the edge to the sampling point is the same as it would have been from the Synchronous Segment to the sample point if no edge had occurred. This is programmed to be the maximum of the Phase Buffer Segment 1 and the Information Processing Time.

The Information processing time is the time segment from the sampling point reserved for sampling the next bit level and is less than or equal to 2 time quanta [5].

- The Sampling Point is always at the end of Ph_Seg1 and is the time at which the bus level is read and interpreted as the value of the current bit.

CAN uses two types of synchronization, Hard Synchronization and Resynchronization.

- Hard synchronization occurs only once during a message transmission, on the recessive to dominant transition of the Start of Frame bit. The bit time is restarted at the end of the synchronization segment after a hard synchronization.
- Resynchronization is subsequently performed during the remainder of the message frame, when a recessive to dominant transition occurs outside of the expected synchronization segment.

The distance between an edge that occurs outside of Sync_Seg and the Sync_Seg is called the phase error of that edge. The re-Synchronization Jump Width (SJW) defines how far a resynchronization may move the Sample Point to compensate for edge phase errors [23]. The maximum change that can be effected on the Phase Buffer Segments is given by the Min [magnitude of phase error, SJW].

2.3.4 Bit Rate and Bus Length

CAN belongs to the class of "In-Bit-Response Protocols"[2], where a wave traveling from a node at one end of the network to a node at the other end of the network should not consume more than approx. 2/3 of a bit time. This constraint implies that the higher the bus length the lower the maximum allowed bit rate becomes.

CHAPTER 3
ASIC DESIGN FLOW AND IMPLEMENTATION

3.1 ASIC Design Flow

Application Specific Integrated Circuits are used to design entire systems on a single chip. ASIC's are inter connect of standard cells which have been standardized by fabrication houses. With the integration of more and more system components on single IC, the complexity of IC fabrication has increased. Modern day system design involves complex layout issues. Specifications of cells are provided by the vendors in form of Technology library which contains information about geometry, delay and power characteristics of cells. Fig. 3.1 shows the sequence of steps to design an ASIC.

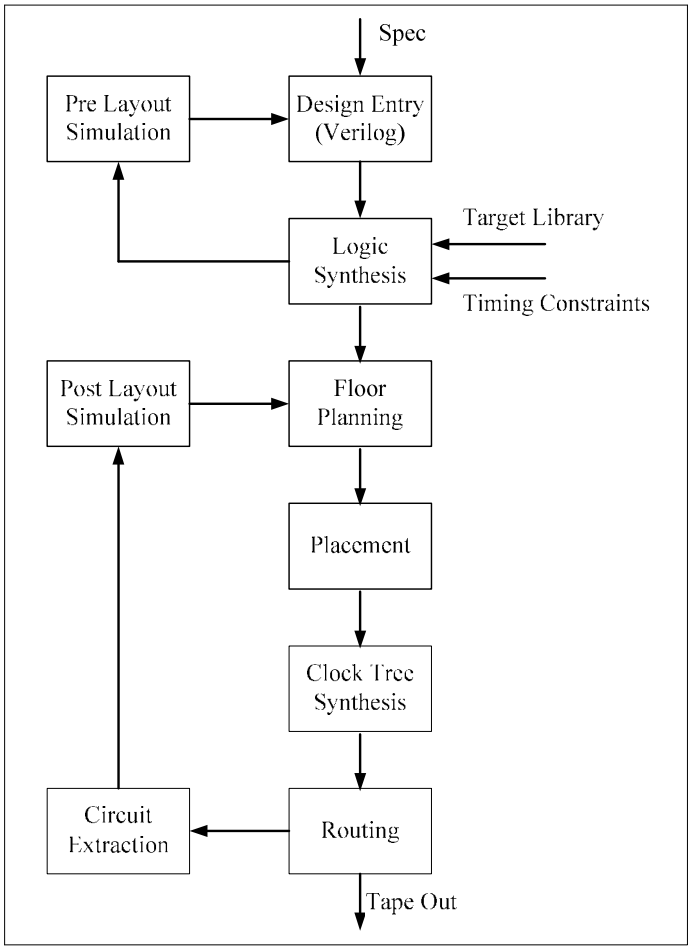

Fig. 3.1 ASIC Design Flow

The steps of the ASIC flow are listed below with a brief description of the function of each step.

- Design entry: Enter the design into an ASIC design system using a hardware description language (HDL).
- Logic synthesis: Use an HDL (VHDL or Verilog) and a logic synthesis tool to produce a netlist - a description of the logic cells and their connections.
- Pre layout simulation: Check to see if the design functions correctly.
- Floor planning: Arrange the blocks of the netlist on the chip.
- Placement: Decide the locations of cells in a block.
- Clock tree synthesis: Insert and optimize clock tree and logic.
- Routing: Make the connections between cells and blocks.
- Circuit Extraction: Determine the resistance and capacitance of the inter-connect wire.
- Post layout simulation: Check to see the design still works with the added loads of the inter-connect wire [16].

3.2 ASIC Implementation

The implementation of a digital ASIC begins soon after the definition of the architecture. A Hardware Description Language (HDL), such as Verilog or VHDL is used to develop the Register Transfer Level (RTL) model of the overall system architecture. The RTL code emulates the process of transferring data between registers through combinational logic. Memory and other macro cells included on the chip are also modeled. The RTL model is validated by the application of test vectors, to ensure correct functionality. Test vectors refer to a sequence of inputs applied to the model, for which an expected response is known. By comparing the expected and actual response of the model, the validity of the RTL model can be determined and corrected as needed. The tool used to simulate this design is ModelSim, a HDL compiler and simulator. The compiled source code will be simulated using it.

After validating the RTL model, logic synthesis is performed to map the RTL code to logic gates in the targeted foundry library. Logic synthesis is the process of automatically converting a given RTL Hardware Descriptive Language of a design to technology gates based on some design constraints. The result of the synthesis process is a Netlist composed of registers, combinational logic, interconnects, and macro cell instantiations. The tool used in this step is Magma Blast Create. Blast Create is a gain-based RTL synthesis tool that provides fast, high-capacity synthesis, integrated into an RTL-to-GDSII design flow.

The synthesized netlist should be simulated using the same test vectors used to validate the RTL. In addition, Formal Verification software can be used to compare two given circuit descriptions. Formal Verification offers an alternative to exhaustive simulation. There is no need to create test vectors and thorough verification is achieved with in a short amount of time. The goal of formal verification is to ensure the equivalence of two given circuit descriptions. These circuits might be given on different levels of abstraction, i.e. register transfer level or gate level. Any simulation or formal verification errors need to be resolved since this may indicate a problem with the library of hardware components used by the synthesizer, a synthesis error, or improperly written RTL. The tool used to verify Logic Equivalence in this design is Verplex.

Synthesis tools typically have little access to place and route data for the design. With little access to physical design data, synthesis tools can miss timing goals by as much as 100% based on the estimated versus the actual interconnect. Hence it is important that the synthesized design exceeds the final timing requirements. A good strategy is to use floor planning tools to perform preliminary placement of logic on a chip and to estimate interconnect delays based on that placement. This timing information can then be used to drive the front-end synthesis, simulation, and timing tools for more accurate results.

Once the design has been synthesized, the physical design process can begin. The first step in the physical design process is floor planning. The floorplan is a physical description of an ASIC. Floorplanning is a mapping between the logical description and

the physical description of an ASIC. Floor planning involves partitioning the design into groupings of cells and placement of macro cells contained in the design on the chip area. Other floor planning considerations are the creation of I/O macros and the power bussing structure. In order to meet the design objectives such as die-size and performance the cells/structures that need to be placed close together are identified in the placement stage.

The next step in the ASIC design flow is the Clock Tree Synthesis. Clock tree synthesis is performed in order to minimize space and power consumed by the clock signal. In this step the clock tree and clock logic is inserted and optimized. Once the clock tree is placed and optimized the place and route can begin. The interconnect wires are routed between the library components in the netlist. After the place and route is completed the design is checked for DRC, LVS, Antenna violations and Crosstalk violations. Finally the parasitic data gathered from the place and routed netlist is used to check if the design meets timing. The tool used to layout the design is Magmas' Blast Fusion. Blast Fusion is a physical design system that provides a complete netlist-to-GDSII flow for cell-based ICs. Blast Fusion includes optimization, place and route, useful skew clock generation; floorplanning and power planning; RC extraction; and a single, built-in incremental timing analyzer.

CHAPTER 4
DESIGN OF THE CAN CONTROLLER

4.1 CAN Controller Implementations

Communication is identical for all implementations of CAN. However, there are two principal hardware implementations of CAN; the Basic CAN and the Full CAN. The major difference between the two is in the interface between the CAN protocol controller and the host CPU. Both the Basic CAN and the Full CAN devices can be used in the same network.

In Basic CAN configurations there exists a tight coupling between the host controller and the CAN controller. Only basic functions concerning the filtering and management of CAN messages are implemented in hardware. A Basic CAN controller typically provides one transmit buffer for outgoing messages and one or two receive buffers for incoming messages. Because there are only two buffers for the reception of messages the host controller will be interrupted to deal with every CAN message before they get overwritten by the following messages. The host controller has to check in software if a message can be ignored or needs to be processed. The host controller also handles the answering of Remote Frames with the corresponding Data Frame. This places a huge burden on the CPU and will take up much of the CPU time leaving it little time to take care of other system related functions. Hence Basic CAN devices should only be used at low baud rates and low bus loads with only a few different messages.

In Full CAN configurations the whole hardware for convenient acceptance filtering and message management is provided. The host controller defines which messages are to be sent and which are to be received. The acceptance filtering mechanism of a Full CAN masks out the irrelevant messages, using the message identifiers and presents the host controller with only those messages that are of interest. Another advantage is that incoming Remote Frames can be answered automatically by the Full CAN controller with the corresponding Data Frame. These features strongly reduce the load on the microcontroller facilitating the use of Full CAN controllers in networks with high baud rates and high bus loads with many messages. However the

implementation of Full CAN is not as simple as Basic CAN. Since Full CAN can manipulate and update the data in the buffer by itself, the concurrency may be difficult to maintain. The data written to the buffer by the host controller may be overwritten by the CAN controller and hence appears to be invalid.

4.2 Architecture Overview

The CAN controller designed for this thesis has been implemented on the lines of a Basic CAN Controller, with a few modifications to it. The functional block diagram for the controller is as shown in the following Fig. 4.1.

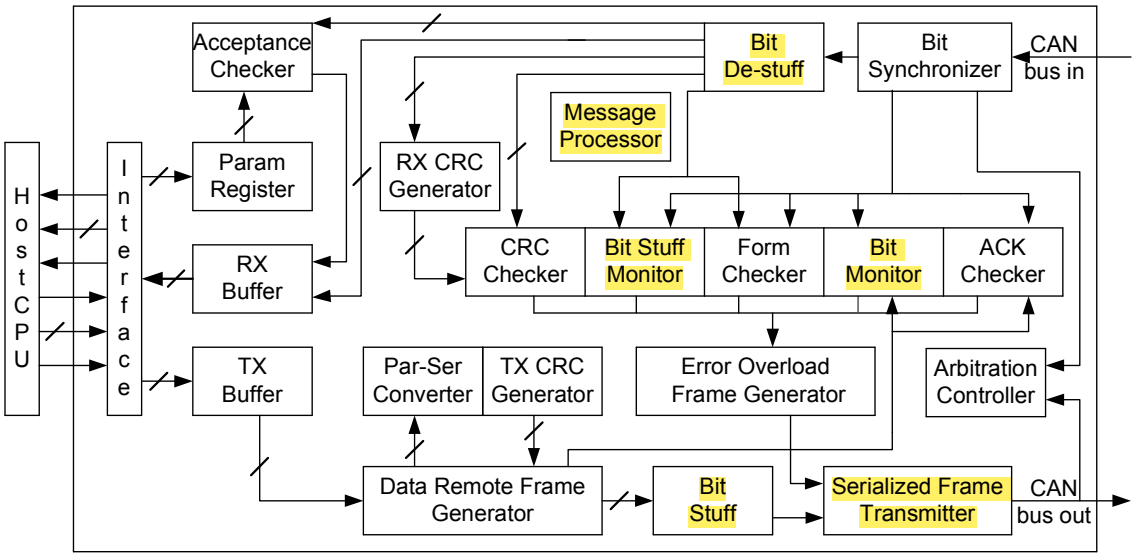

Fig. 4.1 Functional Block Diagram of the CAN Protocol Controller

The various functional blocks in the diagram are described as follows:

- Host CPU: This is the interfacing application which provides the CAN controller with the data to be transmitted across the CAN bus and also reads the received messages from the controller
- Interface: The interface between the host application and the CAN controller consists of an 8-bit data bus to transfer the message to the controller's transmit buffer, an 8-bit data read bus which reads the messages received from the controller's receive buffers and status signals to perform the requisite handshaking for these operations [5].

- Parameter Registers: The code parameter, mask parameter and the Re-Synchronous Jump Width (SJW) specified for the CAN node are stored in this register.
- TX Buffers: There are ten transmit buffers. Each buffer can hold one byte of data. The controller receives the message to be transmitted from the host CPU and stores the message in the transmit buffer before further message processing takes place.
- Data / Remote Frame Generator: Data / Remote Frame Generator is responsible for generating the message frame as specified by the CAN protocol.
- Par-Ser Converter: This unit serializes the message to facilitate the CRC computation.
- TX CRC Generator: Before transmission, this unit computes the CRC for the message to be transmitted. The generated CRC frame is appended to the message being transmitted before bit-stuffing is performed.
- Bit Stuff Unit: This unit performs bit-stuffing as specified by the CAN protocol, making the message suitable for transmission across the CAN network.
- Overload / Error Frame Generator: Generates Error or Overload frame whenever error or overload condition occurs. Error containment measures are also taken care of to ensure the accuracy of the controller's performance and its further participation in the CAN network.
- Serialized Frame Transmitter: This unit transmits the data/ remote frame or the error / overload frame or a dominant bit during the acknowledgment slot based on the prevalent conditions.
- Message Processor: This is the central unit which provides all the control and the status signals to the various other blocks in the controller. This unit routes the different signals generated in various blocks to the necessary target blocks.
- Arbitration Controller: The arbitration controller is responsible for indicating the arbitration status of the node.
- Synchronizer: This unit performs the bit timing logic necessary for synchronizing the CAN controller to the bit stream on the CAN bus. The recessive to dominant

transition edges present on the received bit stream are used for synchronization and re-synchronization.

- Bit De-stuff Unit: This unit performs the de-stuffing of the messages received from the CAN network. This unit also extracts the relevant information from the received message.
- RX CRC Generator: After reception, this unit computes the CRC for the message received.
- Cyclic Redundancy Checker: This unit compares the generated CRC for the received message with the CRC frame received by the node. An error is generated if the two CRC values do not match.
- Bit Stuff Monitor: This unit signals a stuff error when six consecutive bits of equal polarity are detected in the received message.
- Form Checker: A form error is generated if any of the fixed-form fields in a received CAN message is violated. The fixed form fields include the CRC delimiter, ACK delimiter and the EOF field.
- Bit Monitor: A CAN node acting as the transmitter of a message, samples back the bit from the CAN bus after putting out its own bit. If the bit transmitted and the bit sampled by the transmitter are not the same, a bit error is generated.
- Acknowledgment Checker: During the transmission of the acknowledgement slot a transmitter transmits a recessive bit and expects to receive a dominant bit. If the node receives a recessive bit in the acknowledgement slot an ACK error is signaled.
- Acceptance Checker: This unit checks the incoming message ID and determines if the received frame is valid.
- Receive Buffer: There are two 10 byte buffers that are used alternatively to store the messages received from the CAN bus. This enables the host CPU to process a message while another message is being received by the controller.

4.2.1 Building Blocks of the CAN Controller

Each block of the CAN controller performs a specific operation. The functionality of the basic building blocks of the CAN Controller along with its operation is described below.

4.2.1.1 Interface

The interface between the host application and the CAN controller consists of an 8-bit data bus to transfer the message to the controller's transmit buffer, an 8-bit data read bus which reads the messages received from the controller's receive buffers and status signals to perform the requisite handshaking for these operations.

4.2.1.2 Parameter Registers

The controller receives the code parameter, the mask parameter and the Re-Synchronous Jump Width (SJW) specified for the CAN node from the host CPU and stores them in the parameter registers. The content of the code parameter, mask parameter registers are used to determine the acceptance of the message. The Synchronous Jump Width register content is used for bit synchronization. The state diagram for the Parameter Registers is shown in Fig. 4.2.

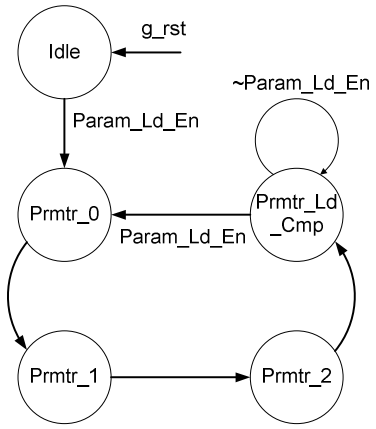

Fig. 4.2 Parameter Registers

Initially the mask parameter and the code parameter need to be loaded into the CAN controller. The *param_ld* signal is asserted by the host controller when it needs to

load new values in the mask and code parameter registers. The host controller accomplishes this by asserting the *param_ld* signal high. This indicates to the CAN controller that the host controller wishes to load the parameters into the corresponding registers. Following the high assertion of the *param_ld* signal the CPU proceeds to send the data on the 8 bit *data_in* bus in bytes. The 8 least significant bits of the mask parameter are transferred first. This is followed by the transfer of the 8 bit data formed by the concatenation of the 5 least significant bits of the code parameter and the 3 most significant bits of the mask parameter. This is followed by the transfer of the 8 bit data formed by the concatenation of the 2 bit SJW and the 6 most significant bits of the code parameter. The *param_ld* signal is asserted only for one clock cycle.

Once the transmission is complete the registers retain the values stored in them. A new value is loaded only when the host controller initializes another parameter load by asserting the *param_ld* signal high. A global reset to the system removes the parameters stored in these registers.

4.2.1.3 Tx_Buffer

There are ten transmit buffers. Each buffer can hold one byte of data. The controller receives the message to be transmitted from the host CPU and stores the message in the transmit buffer before further message processing takes place.

By asserting the *tx_buff_ld* signal high the host controller indicates to the CAN controller that it wishes to transmit a message over the CAN bus. Following the high assertion of the *tx_buff_ld* signal, the host controller sends out the message on the 8 bit data bus in bytes. The controller loads the message on the data bus onto the transmit buffer at the positive edge of the clock. The host controller sends the message in the order of the message identifier first, followed by the control bits, and then the data bytes with the most significant byte of the data being sent first [5]. The signal *tx_buffer_busy* goes high when the controller completes loading the transmit buffer and stays high till the message has been transmitted successfully. On successful transmission the buffers are reset, the *tx_buffer_busy* signal goes low and the controller is ready to receive the next message.

4.2.1.4 Data / Remote Frame Generation

Data / Remote Frame Generator is responsible for generating the message frame as specified by the CAN protocol. The state diagram for the Data / Remote Frame Generation is shown in Fig. 4.3.

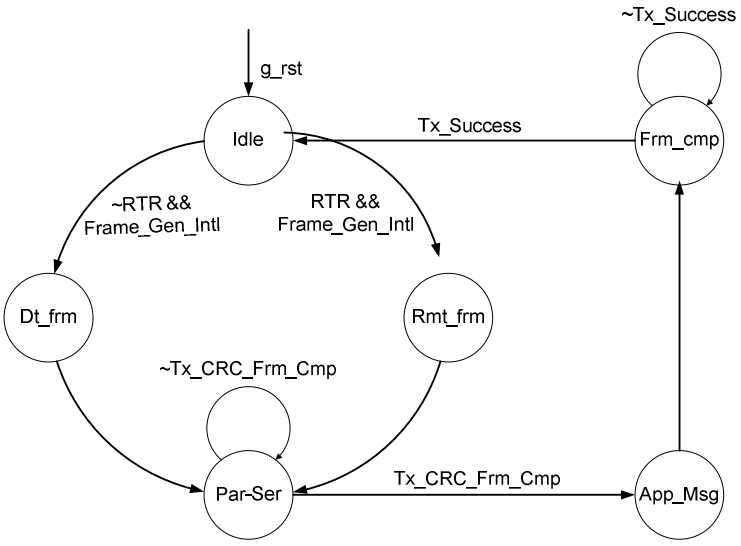

Fig. 4.3 CAN Data / Remote Frame Generation

On loading the final transmit buffer the data / remote frame generation is initialized. This is accomplished by asserting the *frame_gen_intl* high. Based on the Data Length Code (DLC) and the Remote Transfer Request (RTR) bit the *par_ser_data* frame is generated. If the RTR bit is recessive, the message to be transmitted is a Remote. In this case the *par_ser_data* frame does not have any Data Field and will be formed by the concatenation of the dominant Start Bit, the Message Identifier and the Control Field. In case the RTR bit is dominant, the message to be transmitted is a Data Frame and hence will contain a Data Field. In this case the *par_ser_data* frame is formed by the concatenation of the dominant Start Bit, the Message Identifier, the Control Field and the Data Field. The Data Field is of variable length given by the DLC. The Data Field can contain zero to eight bytes of data. The data is transmitted with the MSB first.

In case of a Remote Frame or a Data Frame, with DLC less than 8 the frame is appended with dominant bits to counter for the trailing bits which are not defined by the message. The *par_ser_data* frame is serialized using a Parallel to Series converter and

fed as input to a CRC generator. The high assertion of the *tx_crc_frm_cmp* signal indicates the completion of the CRC calculation.

The generated CRC frame is then appended to the end of the Data Field in a Data Frame or to the end of the Control Field in case of a Remote Frame. The message is appended with recessive bits to counter for the trailing bits which are not defined by the message. The message frame generated after appending the CRC frame is the *dt_rm_frm*. The *dt_rm_frm* is then bit stuffed before transmission. The bit stuffing is initialized by asserting the *bit_stf_intl* signal high.

4.2.1.5 Par-Ser Converter

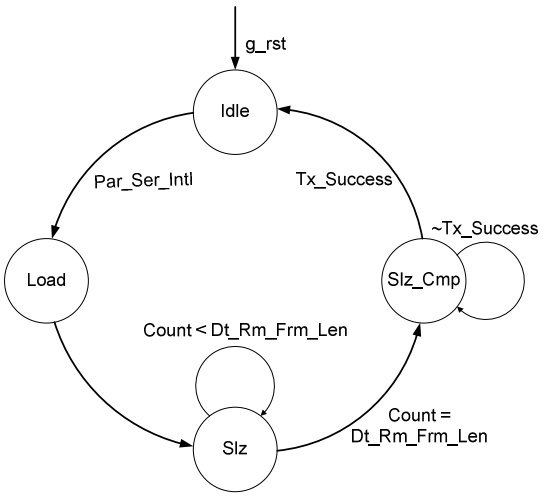

Fig. 4.4 Parallel – Serial Conversion

This unit serializes the message to facilitate the CRC computation. On generating the CRC input frame the *par_ser_intl* signal is asserted high. The high assertion of this signal indicates to the parallel to serial converter block to start the parallel to serial conversion. The message is loaded into another register *temp_data* and the CRC calculation is enabled by asserting the *tx_crc_enable* signal high. The content of *temp_data* is left shifted out to the serial input of the CRC generation module. A count is incremented for every bit shifted out. As soon as the count reaches the frame length of the CRC input frame the *tx_crc_frm_cmp* signal is asserted high and the parallel to serial conversion is stopped. The state diagram for the Par-Ser Converter is shown in Fig. 4.4.

4.2.1.6 CRC Generator

The CRC frame calculation commences with the high assertion of the CRC enable signal. The CRC frame is initialized to 15'h0 with the high assertion of the CRC initialization signal. The CRC frame for CAN is calculated by using a generator polynomial called CAN Polynomial and is represented in hexadecimal by 15'h4599. The CRC input message stream is divided by the generator polynomial given by

$$\text{CAN Polynomial: } x^{15} + x^{14} + x^{10} + x^{8} + x^{7} + x^{4} + x^{3} + 1$$

The remainder of this polynomial division gives the CRC Sequence transmitted as part of the message frame. This function can be implemented using a 15 bit shift register *crc_frm [14:0]*. The *data* bit denotes the next bit of the CRC input bit stream from dominant Start Bit until the end of the Data Field for a Data Frame or until the end of the Control Field for a Remote Frame.

The pseudo code is given as follows [1]:

```
crc_frm = 0
repeat
crc_next = data XOR crc_frm [14];
crc_frm [14:1] = crc_frm [13:0];
crc_frm [0] = 0;
if crc_next then
    crc_frm [14:0] = crc_frm [14:0] XOR (4599hex);
endif;
until (CRC sequence starts).
```

4.2.1.7 Bit Stuff Unit

This unit performs the bit-stuffing mechanism as specified by the CAN protocol, making the message suitable for transmission across the CAN network. As per specifications, bit stuffing is done only on Data or Remote Frames. The input to the stuffing unit is the 98 bit *dt_rm_frm*. The concatenated 98 bit register, *dt_rm_frm* contains the input to the stuffing input and the bits are in the order [97:0] == [Start bit, message

identifier [10:0], RTR bit, Control Field [5:0], Data Field [MSB:LSB], CRC Frame[14:0]]. The register may not always be full due to the variation in the data length, to ensure that the remaining bits do not contain junk data they are padded with recessive bits, 1's in this case. The bit stuffing is initialized by the *bit_stf_intl* signal that goes high on appending the CRC frame to the message and stays high for one clock cycle.

To bit stuff the message stream, the message stream is serialized and then checked for the bit stuffing condition. The stuffed bit stream being put out on the CAN bus has the Start Bit being transmitted first, followed by the message identifier's most significant bit and so on [5].

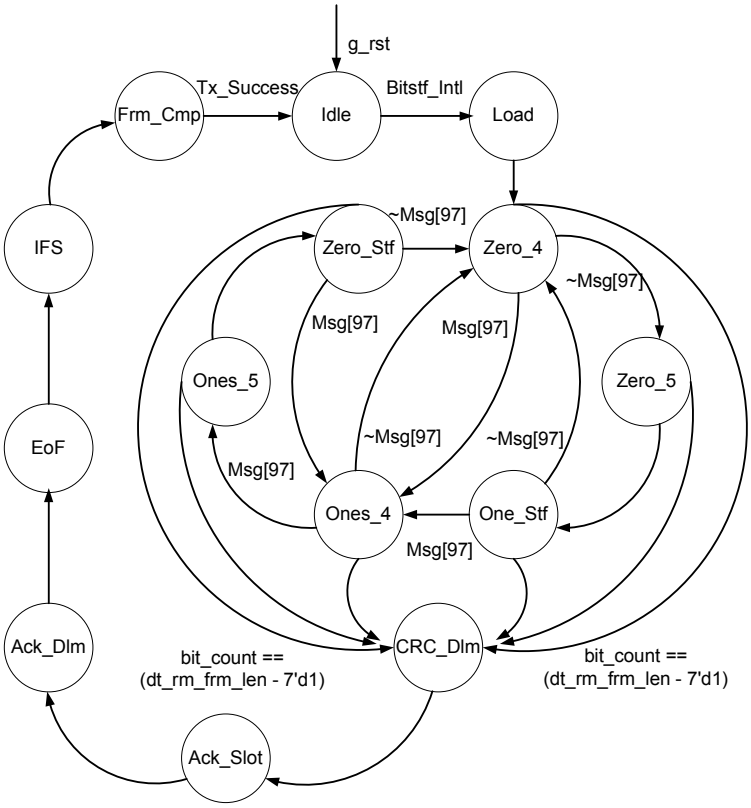

Fig. 4.5 CAN Bit Stuffing

The state machine shown in Fig. 4.5 is used to realize the bit stuffing mechanism. On receiving the *bit_stf_intl* the message stream to be stuffed, *dt_rm_frm* is stored in a temporary register *msg*. Then the 98th bit of *msg* is checked during each clock cycle. If it is a 1, a counter variable for one, *one_count* is incremented or if it is a 0, a counter variable

for *zero_count* is incremented. The states machine stays in state *ones_4* and outputs logic 1, if the sequence of ones is less than five. If the sequence of ones is equal to five then it enters state *ones_5* and outputs a logic 1, the next state would be the *zero_stuff* state as the protocol requires a stuff bit of opposite polarity to be transmitted after every sequence of five similar bits. In the *zero_stuff* state the output is logic 0. Similarly when a sequence of five zero's is detected the state machine enters the *one_stuff* state and outputs a logic 1.

As long as the sequence of ones or zeros is less than five the contents of *msg* are shifted by 1 to the left, thus discarding the bit already transmitted. The bit count is also incremented. However if there is a sequence of five consecutive ones or zeros then the *msg* register remains the same, without being shifted. The bit count is also not incremented and remains the same. The stuffing process is stopped when the bit count equals the length of the frame to be transmitted given by *dt_rm_frm_len.*

The CRC Delimiter, ACK slot, ACK delimiter and EOF bits are appended to the bit stuffed message to form the Data / Remote Frame; these bits are transmitted as recessive bits.

4.2.1.8 Overload / Error Frame Generation

On detecting an overload or error condition, an Overload Frame or an Error Frame is transmitted. A message received when both the buffers are full cannot be stored in the receive buffers and will be lost. To avoid this situation an Overload Frame is generated by the receiver to indicate an overload condition to the other participating nodes. The transmission of the next message on the Bus is delayed by the transmission of the Overload Frame.

Similarly when an error is detected the node sends out an Error Frame. The transmission of the Error Frame produces an error condition in the other participating nodes and causes the message to be retransmitted.

The Error Confinement process is also taken care of by the Overload / Error Frame Generation unit. It is designed as per the specifications in the Data Link Layer of the CAN protocol.

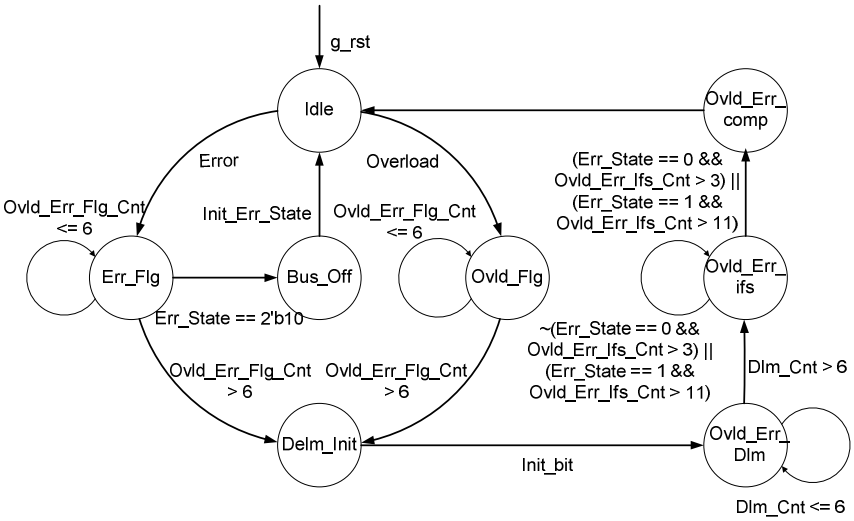

Fig. 4.6 CAN Error / Overload Frame Generation

The state machine shown in Fig. 4.6 is used to realize the error signaling mechanism.

4.2.1.9 Serialized Frame Transmitter

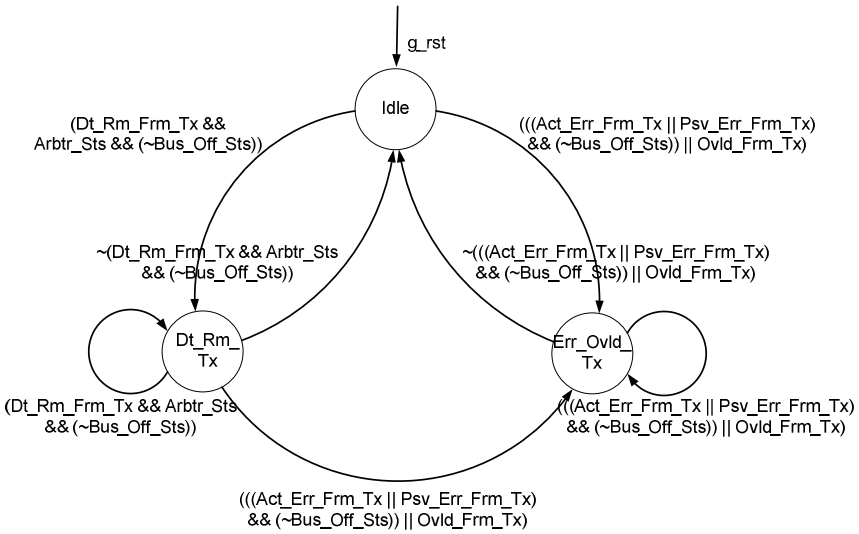

Fig. 4.7 Serialized Frame Transmitter

This unit transmits the Data / Remote Frame or the Error / Overload Frame or a dominant bit during the acknowledgment slot based on the prevalent conditions. The Data / Remote Frame transmission is initialized by asserting the *dt_rm_frm_tx* signal

high. The transmitting node continues to transmit the message until the last bit, provided there is no error condition encountered during the transmission. In the event of an error the node starts transmitting the Error / Overload Frame. The presence of an error or overload condition is indicated by the *act_err_frm_tx* or *psv_err_frm_tx* or the *ovld_frm_tx* signals going high. The node does not transmit an Error Frame when the node is in Bus Off state. Once the transmission is complete the node returns to the idle state. The node transmits a dominant bit during the acknowledge slot, when functioning as a receiver. The node transmits recessive bits during the idle state. The state diagram for the Serialized Frame Transmitter is shown in Fig. 4.7.

4.2.1.10 Message Processor

The message processor is the central unit which provides all the control and the status signals to the various other blocks in the controller. This unit routes the different signals generated in various blocks to the necessary target blocks.

The success of a transmission or reception is indicated by this block. A successful transmission is indicated by the high assertion of the *tx_success* signal similarly the successful reception is signaled by the high assertion of the *rx_success* signals. These two signals facilitate the registers to be reset.

During arbitration messages if a node loses arbitration it has to contend for bus access only after the completion of the current transmission. The high assertion of the *re_tran* signal initializes the retransmission of the message that lost arbitration.

The overload condition is also indicated by the message processor. If both the receive buffers are full and the *rd_en* signal of the node is not low the node signals an overload condition by asserting the *over_ld* signal high.

The message processor also provides information to other modules if an error occurred during the current transmission or reception. In case of an error it ensures that the error is recorded for further use. This module also acknowledges the successful reception of a message till the end of the CRC Field by asserting the *send_ack* signal high. This ensures that a dominant bit is transmitted during the acknowledgement slot.

4.2.1.11 Arbitration Controller

The arbitration controller is responsible for indicating the arbitration status of the node. The *arbtr_sts* signal indicates the arbitration status of the node. If the *arbtr_sts* of the node is logic 1 then the node is a transmitter if it is logic 0 then the node has lost arbitration and functions as a receiver of the ongoing message. The node which loses arbitration asserts a signal *msg_due_tx* to indicate that a message is due for transmission.

4.2.1.12 Synchronizer

This unit configures the timing parameters of the bit time for the CAN node. Each CAN node is configured individually to create a common bit rate for all the nodes on the network even though the CAN nodes' oscillator periods may be different. Synchronizations and re-synchronizations are performed on the recessive to dominant transition edges. The purpose is to control the distance between edges and Sample Points. The specifications for bit timing and synchronization as specified by the CAN protocols have already been discussed in Chapter II.

The specifications used for the design have been obtained from [8] are given below:

- Bit Rate for CAN transmission: 1Megabits/second
- CAN bus length: 20 meters
- Bus Propagation Delay: $5 * 10^{-9}$ seconds/meter
- Physical Interface (PCA82C250) transmitter plus receiver propagation delay = 150 nanoseconds
- Main oscillator frequency: 8 MHz

The design for the bit timing and synchronization unit involves the calculation of the time quanta required for the Bit timing Parameters, which are the Synchronization Segment, Propagation Segment, Phase Segments 1 and 2 and the Synchronization Jump Width (SJW).

Calculation of minimum permissible time for the Propagation Segment

Physical delay of the CAN bus = Propagation delay * length of CAN bus

= 5 * 10^-9 * 20 = 100 nanoseconds

Total propagation delay involved in back and forth bit transmission:

= (Interface delay + physical bus delay) * 2

= (150 + 100) nanoseconds * 2

Total propagation delay = 500 nanoseconds

With a baud rate prescaler (BRP) value of 1, the CAN system clock frequency is calculated to be 8 MHz. The value of a time quantum is given by the CAN system clock period and is given to be 125 nanoseconds.

With the CAN bit rate being 1 Mega bits/second, and the time quanta/bit being 125 nanoseconds, the total number of time quanta per unit bit time is

=1 Mega bits per seconds / 125 nanoseconds

= 8 time quanta per unit bit time.

Number of time quanta for Propagation Segment

= (Total propagation delay) / (One time quanta)

= 500/ 125

= 4 time quanta

With the total time quanta in a bit time being 8, 1 time quantum is reserved for Synchronization Segment and 4 time quanta are reserved for the Propagation Segment. This leaves (8-5) = 3 time quanta being reserved for the Phase Segments 1 and 2. As per the rule for designation of phase segments [1] which states that the value for phase segment1 must be equal to phase segment2, unless Phase Segment1 is less than the Information Processing Time (IPT). The value of IPT is always 2 time quanta. From this, the value of Phase Segment1 is determined to be 1 time quanta and the value for Phase Segment2 is determined to be 2 time quanta.

The value of Re-Synchronization Jump Width (SJW) is calculated as the smallest of 4 time quanta and Phase Segment1 and is thus calculated as 1 time quantum.

The values for all the segments of the nominal bit time and SJW are as follows:

Synchronization Segment = 1 time quantum

Propagation Segment = 4 time quanta

Phase Segment1 = 1 time quantum

Phase Segment2 = 2 time quanta

SJW = 1 time quantum

In this design, the CAN controller is configured to operate on the CAN bus at 1Mbits/s. This is done by setting the bit time at 8TQ and the baud-rate prescaler equivalent to divide by one. The state diagram for the Synchronizer is shown in Fig. 4.8.

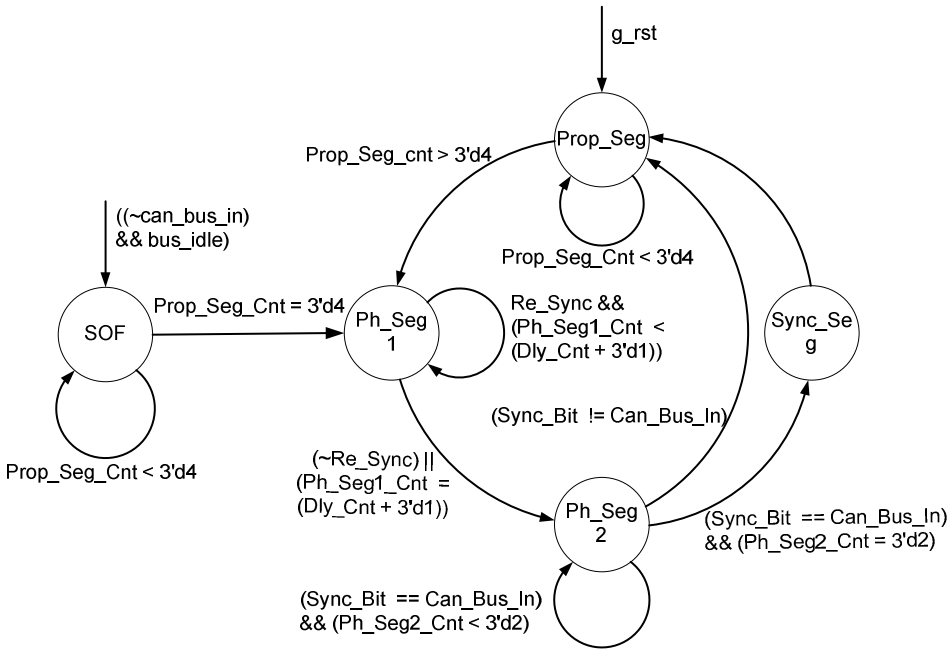

Fig. 4.8 Bit Synchronization

4.2.1.13 Bit De-stuff Unit

This unit performs the de-stuffing of the messages received from the CAN network. This unit also extracts the relevant information from the received message.

The CAN bus bit stream is sampled by the Synchronizer of the CAN controller. This sampled bit stream is then de-stuffed before the relevant information is extracted from the received message. Due to the bit stuffing process of the CAN protocol a stuff bit of opposite polarity follows a sequence of 5 consecutive bits of the same polarity. The function of the de-stuffing unit is to remove the stuffed bits from the received message. The state machine for the Bit De-stuff Unit is shown in Fig. 4.9.

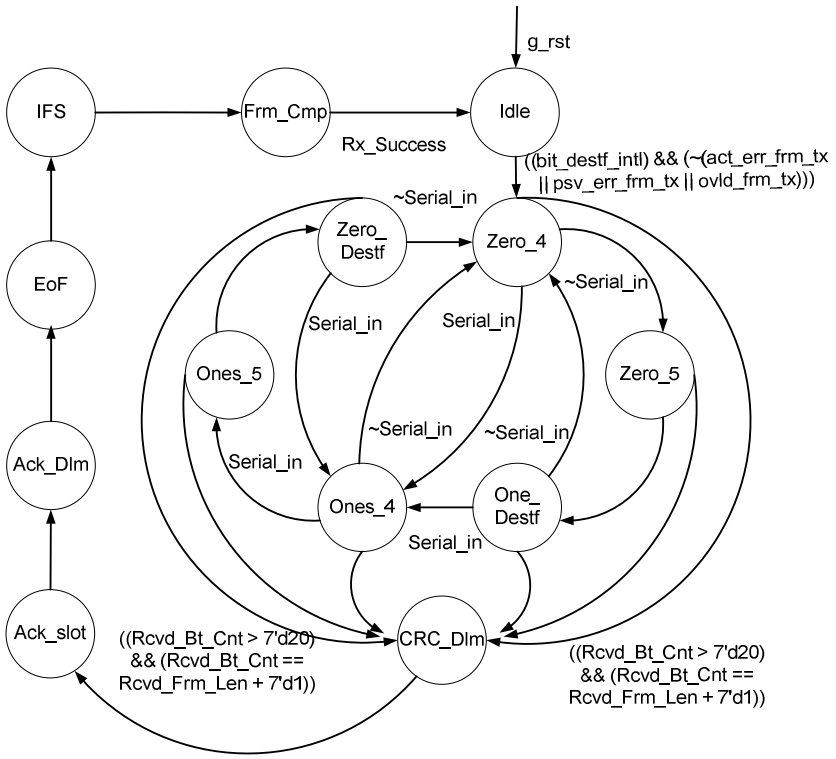

Fig. 4.9 CAN Bit De-Stuffing

The de-stuffing process is initialized by the high assertion of the *bit_destuff_intl* signal. As soon as the de-stuffing process is initialized the CRC calculation of the received bit stream is enabled by asserting the *rx_crc_enable* signal high. A 64 bit temporary register *destf_out* stores the received and de-stuffed bits. The temporary register is shifted to the left by one bit position for every de-stuffed bit and the incoming de-stuffed bit is moved into the 0^{th} bit position of the temporary register. Fig. 4.10 demonstrates the method of De-stuffing.

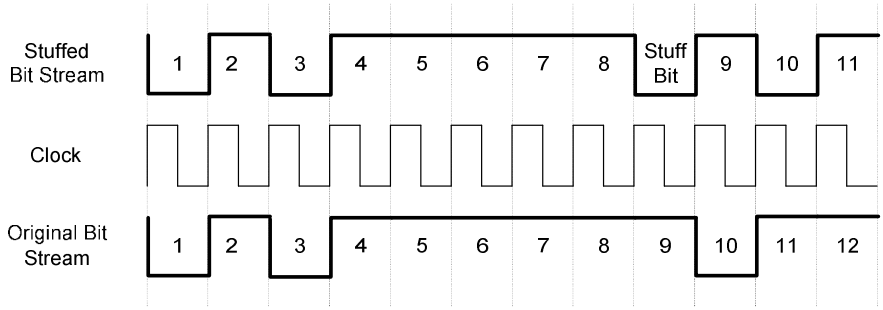

Fig. 4.10 Bit De-stuff

The incoming bit is checked during every clock cycle. If it is a 1, a counter variable for one, *one_count* is incremented or if it is a 0, a counter variable for *zero_count* is incremented. These counters are incremented even if a stuff bit is encountered. After coming across a sequence of five ones or five zeros the next bit is removed from the bit stream and it is not stored in the destf_out register.

The *rcvd_bt_cnt* is incremented at the positive edge of every clock after the initialization of the de-stuff process. However if a stuff bit is encountered in the bit stream then the *rcvd_bt_cnt* is not incremented and remains the same. The information from the received message is extracted when the received bit count *rcvd_bt_cnt* reaches a particular value. There is a possibility of a stuff bit occurring at a particular instance where the *rcvd_bt_cnt* equals the value at which the register is to be loaded. This would result in the storage of an unintended data. To overcome this problem signal *de_stuff* is asserted high to indicate the presence of a stuff bit. This would make sure the register is loaded with the relevant data.

The CRC generation is also disabled on detecting a stuff bit to make sure the CRC calculation is performed on the un-stuffed message bit stream. This is accomplished by asserting the *stuff_bit* signal high which in turn asserts the *rx_crc_enable* signal low on detecting a sequence of five ones or zeros in the bit stream.

The de-stuffing process is stopped when the bit count equals the frame given by *rcvd_frm_len.* All the receive registers are reset with the high assertion of the *rx_success* signal.

4.2.1.14 Error checkers

These consist of checking blocks for CRC, Stuff, Form, Bit and Acknowledgement errors according to the CAN specification.

4.2.1.14.1 Cyclic Redundancy Checker

The CAN controller on receiving the *rcvd_crc_flg* performs the CRC sequence comparison. The *rcvd_crc_flg* register holds the received CRC Frame and the *rx_crc_frm*

register holds the generated CRC for the received application data. A CRC Error is flagged by asserting the *crc_err* signal high if the received CRC frame and the generated CRC do not match.

4.2.1.14.2 Bit Stuff Monitor

This unit signals a stuff error when six consecutive bits of equal polarity are detected in between Start of Frame and the CRC Delimiter of the received message. The *one_count* and *zero_count* are fed as input to the bit stuff monitor module. A stuff error is flagged if the *one_count* is equal to five and the serial input is equal to logic 1 or if the *zero_count* is equal to five and the serial input is equal to logic 0. The occurrence of a stuff error is signaled by asserting the *stf_err* signal high.

4.2.1.14.3 Form Checker

This unit checks for the serial input at the fixed from fields which are the CRC Delimiter bit, Acknowledge Delimiter bit and the End of Frame Space bits. If the receiver detects a dominant bit in any of these fields a Form Error is signaled by asserting the *frm_err* signal high.

4.2.1.14.4 Bit Monitor

A CAN node acting as the transmitter of a message, samples back the bit from the CAN bus after putting out its own bit. A Bit Error occurs if a transmitter sends a dominant bit but detects a recessive bit on the bus line or, sends a recessive bit but detects a dominant bit on the bus line. When a dominant bit is detected instead of a recessive bit, no error occurs during the Arbitration Field or the Acknowledge Slot because these fields may be overwritten by a dominant bit in order to achieve arbitration and acknowledge functionality. A Bit Error is signaled by asserting the *bt_err* signal high.

4.2.1.14.5 Acknowledgment Checker

During the transmission of the acknowledgement slot a transmitter transmits a recessive bit and expects to receive a dominant bit. If the transmitting node receives a recessive bit in the acknowledgement slot it is understood that none of the nodes in the network received the message correctly and an ACK error is signaled. If the node receives a dominant bit in the acknowledgement slot then it is understood that at least one other node, has received the frame correctly. The presence of an acknowledgement error is signaled by asserting the *ack_err* signal high. Fig. 4.11 demonstrates the CAN acknowledgement process.

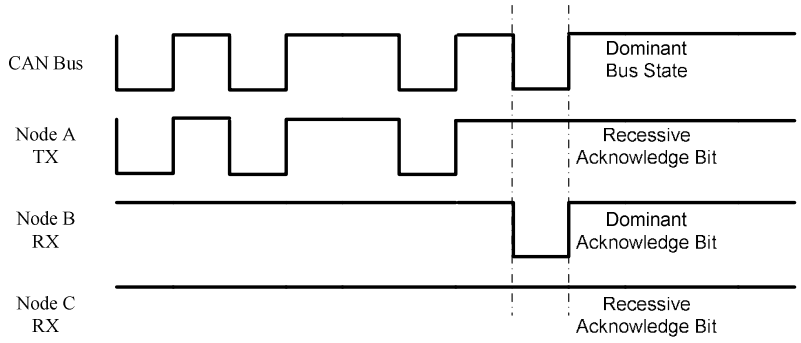

Fig. 4.11 CAN Acknowledgement

4.2.1.15 Acceptance Checker

This unit checks the incoming message ID and determines valid frame. The Acceptance Filter, based on the Mask and Code Parameter of the host application, filters the received messages and stores only those required by the host application. Thus the filtering mechanism ensures only relevant messages are processed and the rest are ignored. All messages that are let through the filter must be read and checked by the microcontroller. This means that the final filtering is done in software.

The design of the Acceptance Filter is defined by two parameters, acceptance Mask Parameter and acceptance Code Parameter. The acceptance Mask Parameter specifies which particular bits to compare in the acceptance Code Parameter with the identifier of the message. The set bits in the Mask Parameter indicate the bits that are relevant in the acceptance Code Parameter. The acceptance Code Parameter and the

inbound message ID must be equal in those bit positions which are marked relevant by the acceptance Mask Parameter bits. For standard identifiers, the acceptance Mask Parameter and the acceptance Code Parameter must be both, in the range of 0 to 2047. The acceptance Mask Parameter and the acceptance Code Parameter are user defined. These values can be changed using the registers available for the same.

The filtering process compares the Message Identifier with the hardware filters, composed of the acceptance Code and Mask registers. The filtering process evaluates the following binary expression to check the acceptance of the message

AND (XOR (*code_param*, *rcvd_msg_id*), *mask_param*) = 11'b00000000000

If the equation evaluates to zero, the received message is accepted. Even if one of the 11 bits is logic 1, the received message is deemed to have failed the filtering mechanism and is rejected.

The *rcvd_lst_bit_eof* is asserted high to indicate the reception of the last bit of the End of Frame. The filtering process is initialized only on the error free reception of the message till the last bit of the End of Frame.

In the event of an error during the reception of the message the *stf_frm_crc_err_pre* signal is asserted high. The acceptance unit checks if the *stf_frm_crc_err_pre* signal is asserted high. If it is asserted high the filtering mechanism is not initialized and the message is rejected.

If the message was received without any errors and if the filtering equation given above evaluates to zero, the received message is accepted. The acceptance of a message is indicated by asserting the *acpt_sts* signal high.

4.2.1.16 Receive Buffer

There are two 10 byte buffers, *rx_buff0* and *rx_buff1* that are used alternatively to store the messages received from the CAN bus. This enables the host CPU to process a message while another message is being received by the controller.

If the *rx_buff0* is not written into the *rx_buff_0_wr_stat* signal is asserted high. If *rx_buff0* is written into *rx_buff_0_wrtn* is asserted high. Similarly the status of *rx_buff1* is

also indicated. The received message is stored into the first buffer that is free on passing the acceptance filtering. The CAN controller checks the status of *rx_buff0*, if *rx_buff0* is written then *rx_buff1* is checked and written into. As soon as the data is written into the buffers the corresponding buffer written signal is asserted high.

The data is written into the buffer in the following order the received message Id given by *rcvd_msg_id*, *rcvd_rtr*, the *rcvd_dlc* and the *rcvd_data_frm* with the MSB first. The data is read from the receive buffers by the host application by asserting the *rd_en* signal low. This initializes the read operation.

Two signals *rx_buff_0_active* and *rx_buff_1_active* ensure that the data is read in the correct order. These signals go high as soon as the read operation for a particular buffer is initialized. At end of the read operation of a particular buffer the controller checks if the other buffer has been written into. If the other buffer is written into the controller asserts the active signal of the other buffer high, if not the active signal of the other buffer is asserted low. This ensures that the controller reads the data in the order in which the data arrives.

Once a buffer is read the *rx_buff_0_read* or the *rx_buff_1_read* signal is asserted high for the corresponding buffer. With the high assertion of the buffer read signal the contents of the corresponding buffer is reset and the buffer is ready to take in a new message. The data is sent out on the *data_out* bus one byte at a time.

4.3 Logical Synthesis

Logical synthesis includes the following:

- High-level optimization, including expression synthesis, resource sharing, and automatic clock gating.
- Area optimization, including mapping to SuperCell models.
- Constraint-driven optimization also called timing-driven optimization, including scan insertion, gain trimming, and arithmetic architecture selection

Magma blast Create is used in this design. Magma Blast Create is a front-end tool for RTL synthesis, design-for-test (DFT), and physical synthesis. Blast Create outputs a

design that is a placed, timing-correct physical design, with DFT structures inserted and that is ready for routing.

Magma RTL synthesis uses gain-based synthesis. The gain of a cell is the ratio between the output capacitance and input capacitance of the cell. Mapping to SuperCell models enables use of this ratio for balancing cell sizes and delays to maintain constant timing. By using SuperCell models and gain-based synthesis, Blast Create can defer cell sizing until later in the design flow, during placement.

A SuperCell model is a technology-specific scalable timing model created during library preparation. The SuperCell model represents a range of cell sizes, rather than a single size. The model contains timing and area descriptions for a collection of cell drive strengths of a particular function.

4.4 Design Constraints

The design constraints can be defined either by using a Synopsys Design Constraints (SDC) file or using M-Tcl constraint file. The constraints are mostly related to timing. The constraints defined include definition of clock domains, I/O timing, constants, and timing exceptions.

This design is constrained by specifying the master clock frequency of 8 MHz with a duty cycle of 50%. A clock circuit generated from the master clock is specified to run at 1MHz. The clock uncertainty is specified to be 0.5ns for both the clocks. The source and network latency have been set as 2ns and 1ns respectively for the master clock. The maximum clock transitions are set at 0.2ns.

A false path is set for the asynchronous reset signal. The input and output delays on ports relative to the system clock, have been set at 400ns and 100ns respectively. The input delay for an input operated upon on the oscillator clock transition is set at 50ns. The input delay has been specified to be approximately 40% of the clock period and the output at approximately 10%. In addition to setting the input and output delays on ports relative to a clock, set a driving cell on the inputs and a capacitive load on the outputs to provide a completely constrained design.

4.5 Floor Planning and Power Routing

Floorplanning is the process of:

- Positioning blocks on the die or within another block, thereby defining routing areas between them.
- Creating and developing a physical model of the design in the form of an initial optimized layout.

Power planning is the process of defining the power and ground nets of your design and specifying their structures, which distribute power through your design:

- Core and macro rings
- Power and ground mesh
- Rails
- Pin taps and pin ties
- I/O pad rings for chips and I/O pin rings for macros and other blocks

Magma Blast Fusion is used for the floor planning power routing and the physical synthesis sections of the design flow. In the Blast Fusion design flow, the floorplanning and power routing come after the completion of the constraint based optimization phase and before the detailed placement.

The floorplanning is done by specifying the width, height and the spacing between the inner core and the outer box of the design. In this design the width and height of the core is set at 325μm. The row height is set at 2.87μm; this value is obtained from the standard technology library file.

The target utilization is set at 60%, the power nets will consume a major portion of the core area hence the target utilization is not set at 100%. The aspect ratio for this design is '1'; this will produce square shaped floorplan. The physical library is bound to the netlist at the floorplan stage. The pins are placed along the periphery of the floorplan in the outer box. The pins are ordered as required.

The core rings are routed using Metal 5 horizontally and Metal 6 vertically. The core rings run around the periphery of the core. They are placed 3μm away from each other and from the core. A Vertical mesh is created to take power to the rails. They are

created using Metal 4 and a total of a 5 groups are created each group has a VDD and VSS mesh. From the mesh the power is routed to the rails through vias.

4.6 Physical Synthesis

Physical Synthesis is the process of

- Defining the preliminary standard-cell placement (global placement). This step distributes standard cells evenly over the available core area. After this step, the cells usually overlap and are not placed precisely in rows.
- Generating a legal placement (detailed placement).
- Distributing cells evenly and minimizing congestion (incremental global placement).

Physical synthesis and optimization of the floorplan, including standard cell placement, occur after floorplanning and power planning. In this stage Blast Fusion does global placement and incrementally optimizes mapping, structuring, and buffering. It converts the design from SuperCell models to standard cells.

4.7 Clock tree Synthesis

Clock Synthesis is the process of

- Inserting clock trees with the lowest insertion delay possible to replace high fan out clock nets.
- Balancing the clusters based on a specified constraint.
- Adjusting cell sizing, based on the new clock timing and loads associated with hold buffering.
- Optimizing the timing by pin swapping.

At the clock Synthesis stage Blast Fusion inserts and optimizes the clock logic and trees. Blast Fusion distributes the clock nets using H-tree clusters, balances the clusters, does placement, and routes most of the H-tree clusters. Additionally, it does buffering and further optimization that should be done only after the effects of clock networks can be taken into account.

4.8 Routing

Routing is the process of

- Routing most of the short, one layer, cell-to-cell routes.
- Fixing slew violations by adjusting buffering, sizing, placement, and routing.
- Running final track and power routing.
- Performing routing refinement to resolve miscellaneous routing rule violations.
- Performing incremental final routing.
- Fixing antenna violations so all antenna rules are upheld.
- Checking the full design one last time for DRC violations.

In this stage, Blast Fusion routes all signal nets, and finishes clock net routing and all power routing not previously done. This stage also contains optimization stages that can add buffers and resize cells. This fixes any slew and noise violations it identifies at this stage of the flow.

This stage completes the physical implementation of a design. After this stage is finished, all routes are DRC and LVS clean, and free of any antenna violations.

The script for logical synthesis, Floor Planning, Power Routing, Clock Tree Synthesis and Routing of the design is provided in the appendix. The Synopsys Design Constraint file for the design is also provided.

4.9 Formal Verification

Formal Verification offers an alternative to exhaustive simulation. There is no need to create test vectors and thorough verification is achieved with in a short amount of time. The goal of formal verification is to ensure the equivalence of two given circuit descriptions. These circuits might be given on different levels of abstraction, i.e. register transfer level or gate level. Formal verification is performed at every level of the design flow from the logic synthesis stage to the final stage. The tool used to establish Logic Equivalence is Cadence Verplex.

To establish logic equivalence the tools needs the two circuit descriptions to be compared. The corresponding technology library has to be imported into the tool. For RTL

to gate formal equivalence checking, use simulation libraries instead of synthesis libraries because design verification signoff happens for simulation libraries and not for synthesis libraries.

The comparison can be between a RTL and a netlist or between different netlist. The initial set ups include defining how the tool handles undefined cells it encounters in the design, defining how the tool treats un-driven signals in the designs, it can either be treated as a logic '0' or logic '1', or don't care or high impedance.

Once the designs have been read in a rule check can be performed on the golden design to display the list of rule violations. The mapping method is also specified in the initial set up. This specifies the phase, case sensitivity, and handling for unreachable points and black boxes when tool maps the key points. Once the designs have been mapped the tool adds mapped points to the compare list and compares the mapped points of the golden design and the revised design. After the comparison reports can be generated to view the equivalent points, non equivalent points, un-driven nets, floating nets and black boxes in the design.

CHAPTER 5
RESULTS

5.1 Simulation Results

5.1.1 Verification Methodology

A simulation model for verifying the CAN controller has been designed with four CAN nodes instantiated on a network. The four CAN nodes have been instantiated as *can_0*, *can_1*, *can_2* and *can_3*. In this simulation model, each CAN controller is capable of transmitting and receiving messages. This means that any of CAN controllers can be a master. The clock frequency used in this system is 8MHz and the CAN transmission rate is 1Mbit/s. The top module provides the nodes with the messages to be transmitted across the CAN network. The test bench module also provides the oscillator clock of 8 Mega Hz and the data at the CAN system clock of 1 Mega Hz for the CAN nodes. Apart from the above functions the top module Wire-AND's the bit stuffed output streams of the CAN nodes.

The CAN message communication consist of error-free transmission and reception of data frames, remote frames and also error frames. The functioning of the CAN network has been demonstrated by verifying each of the above cases. Since the verification model is written in Verilog HDL, the CAN controller can be tested at any time during the design phase.

5.1.2 Functional Verification

5.1.2.1 Wire-ANDing of the CAN Bus

The Wire-ANDing mechanism of the CAN Bus is taken care of by the top module. This can be seen in the Fig. 5.1. During the Bus idle state the CAN nodes transmit their respective outputs onto the CAN Bus. The arbitration process is carried out by sampling back the bit stream from the CAN bus. The nodes continue to put out their respective bit streams as long as the message identifier bits being transmitted are the same. The nodes *can_0*, *can_1* and *can_3* compete for access simultaneously. The Wired-AND result of these outputs is put out on the CAN bus.

The figure illustrates the Wire-ANDing mechanism of the CAN Bus.

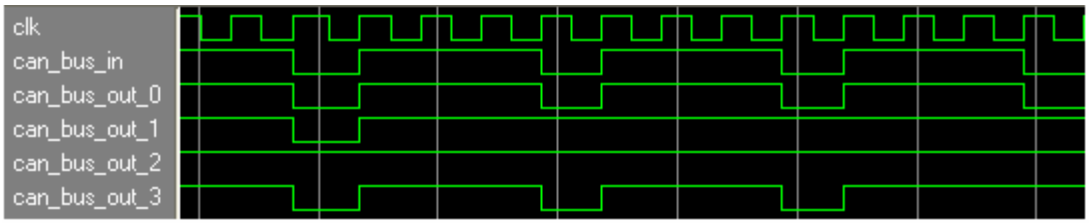

Fig. 5.1 Wire-ANDing of the CAN Bus

5.1.2.2 Host Interface for Data Transmission

Fig. 5.2 and Fig 5.3 illustrate the interface between the host controller, which provides the CAN node with data to be transmitted and the CAN node *can_0*. The *mask_param* and *code_param* registers are initially loaded into the corresponding registers. These registers are not updated every time a message is transmitted. These registers are modified only when there is a need to modify the filter.

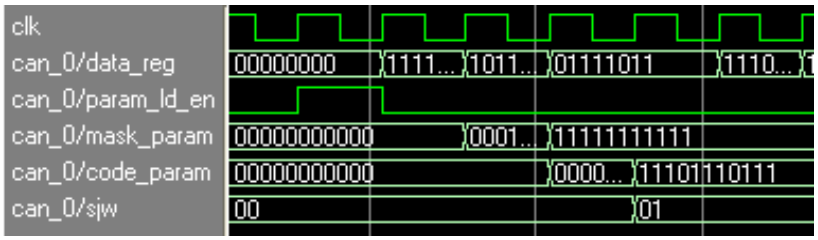

Fig. 5.2 Host controller - CAN controller interface to load Parameters

The loading of the parameter registers is initiated by asserting the *param_ld* signal high by the host controller. The host controller proceeds to send the data in bytes. The first byte transmitted corresponds to the 8 least significant bits of the *mask_param*. The next bytes 3 least significant bits corresponds to the 3 most significant bits of the *mask_param* and the 5 most significant bits to the 5 least significant bits of the *code_param*. The next bytes 6 least significant bits correspond to the 6 most significant bits of the code_*param* and the 2 most significant bits to the *SJW*. The CAN controller node *can_0's* parameter registers are loaded with the following values

mask_param = 11'b11111111111

code_param = 11'b11101110111

SJW = 2'b01

While the parameter registers are fixed for a node, the transmission buffers are updated every time a new message is to be transmitted. The host controller signals its need for data transmission by asserting the *tx_buff_ld* signal high. Following the high assertion of the *tx_buff_ld* signal, the host controller sends out the message on the 8 bit data bus in bytes. The signal *tx_buffer_busy* goes high when the controller completes loading the transmit buffer and stays high till the message has been transmitted successfully. The RTR bit and the Data Length Code are extracted from the message. The *frame_gen_intl* signal is asserted high on loading the last buffer. This signal initializes the frame generation process.

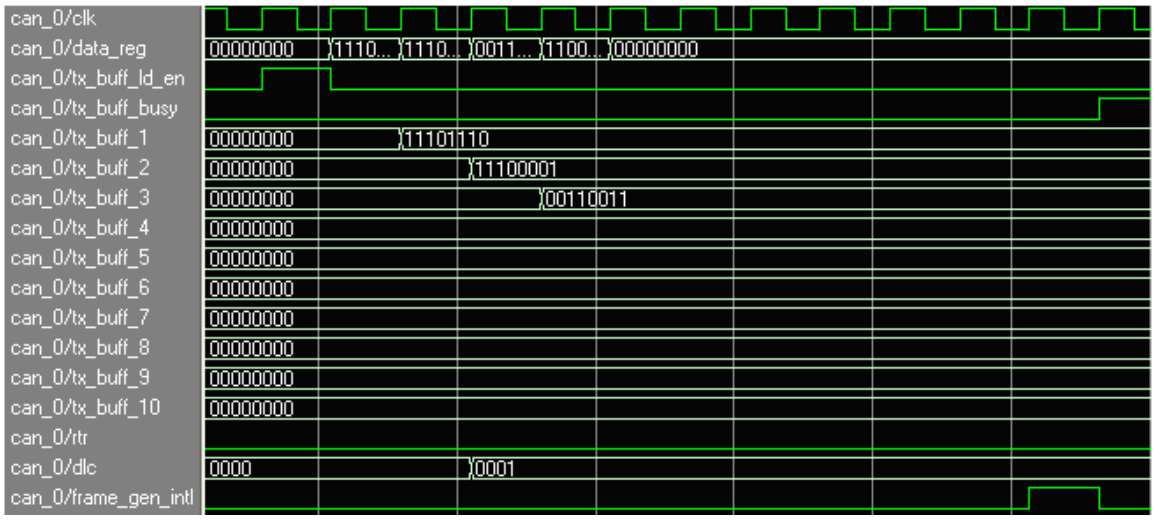

Fig. 5.3 Host Controller - CAN controller interface to load a Message

The CAN controller node *can_0's* transmission buffer registers are loaded with the following values.

tx_buff_1 = 8'b11101110

tx_buff_2 = 8'b11100001

tx_buff_3 = 8'b00110011

tx_buff_4 = 8'b11001100

tx_buff_5 = 8'b00000000

tx_buff_6 = 8'b00000000

tx_buff_7 = 8'b00000000

tx_buff_8 = 8'b00000000

tx_buff_9 = 8'b00000000

tx_buff_10 = 8'b00000000

The DLC is given by the least four bits of tx*_buff_2* and the RTR bit is given by the 4th bit of *tx_buff_2*.

DLC = 4'b0001

RTR = 1'b0

5.1.2.3 Data and Remote Frame Framing

The Frame generation is initialized by the high assertion of the *frame_gen_intl* signal. Based on the Data Length Code (DLC) and the Remote Transfer Request (RTR) bit the *par_ser_data* frame is generated. The *par_ser_data* frame is formed by the concatenation of the dominant Start Bit, the Message Identifier, the Control Field and the Data Field if it exists. Dominant bits are appended to the frame to counter for the trailing bits which are not defined by the message. The CRC generation initialization is demonstrated in Fig. 5.4.

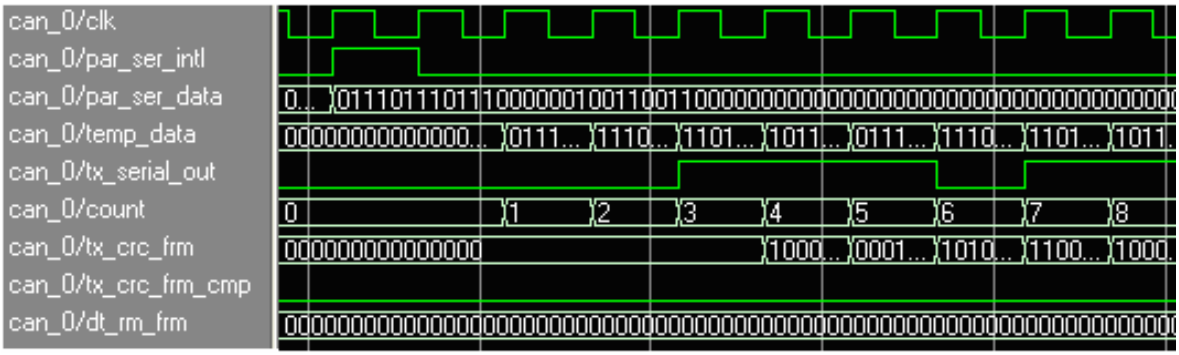

Fig. 5.4 CRC Generation Initialization

The *par_ser_data* frame is serialized by asserting *par_ser_intl* signal high. The message is loaded into another register *temp_data* and the CRC calculation is enabled by asserting the *tx_crc_enable* signal high. The content of *temp_data* is left shifted out to the serial input of the CRC generation module through the *tx_serial_out* signal. A count is incremented for every bit shifted out. As soon as the count reaches the frame length of the

CRC input frame the *tx_crc_frm_cmp* signal is asserted high and the parallel to serial conversion is stopped. The high assertion of the *tx_crc_frm_cmp* signal indicates the completion of the CRC calculation. The message frame generated after appending the CRC frame is the *dt_rm_frm*. The *dt_rm_frm* is then bit stuffed before transmission. The bit stuffing is initialized by asserting the *bit_stf_intl* signal high.

The Data / Remote Frame generation is demonstrated in Fig. 5.5. The generated CRC sequence is 15'b100101111010111. The generated Data Frame is 98'b01110111011100000010011001110010111101011111111111111111111111111111111111111111111111111111111111.

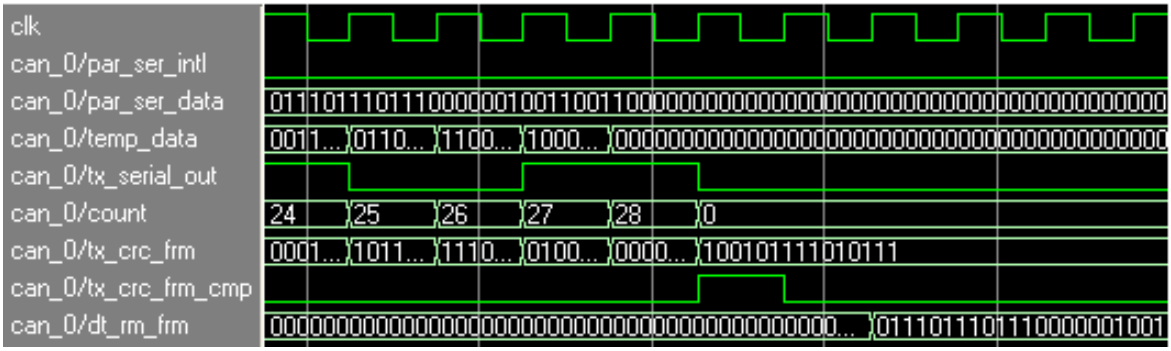

Fig. 5.5 Data Remote Frame generation

5.1.2.4 Bit Stuffing

The bit stuffing mechanism is demonstrated in Fig. With the initialization of bit stuffing the contents of the 98 bit register *dt_rm_frm* are transferred to a temporary register *msg*. The contents of the *msg* register are serialized and then checked for the bit stuffing condition. At every positive edge of the clock the contents of the *msg* register are shifted left by 1. The left shifted bit is placed on the *dt_rm_out* bus which is connected to the CAN bus. A count of the bits in the original un-stuffed message stream is kept in the register *bit_count*. The stuffed bit stream being put out on the CAN bus has the Start Bit being transmitted first, followed by the message identifier's most significant bit and so on.

Two counters *one_count* and *zero_count* keep tab of the number of consecutive ones and zeros encountered respectively in the message stream. If one of the counters

value equals five, the contents of the *msg* register are not shifted. This is to accommodate the stuffed bit being put out on the bus. During this bit time, the value of *bit_count* also remains the same. If the *one_count* equals five, a dominant bit is stuffed into the bit stream and if the *zero_count* equals five, a recessive bit is stuffed into the bit stream.

This is evident when the *bit_count* equals 17 in the figure. Here the message to be transmitted has six consecutive zeros starting from the dominant *RTR* bit till the end of the *DLC*. On detecting five zeros at *bit_count* 17 a recessive bit is stuffed into the bit stream. The *bit_count* is not incremented and the *msg* register is not shifted left as explained above. The bit stuffing mechanism is demonstrated in Fig. 5.6.

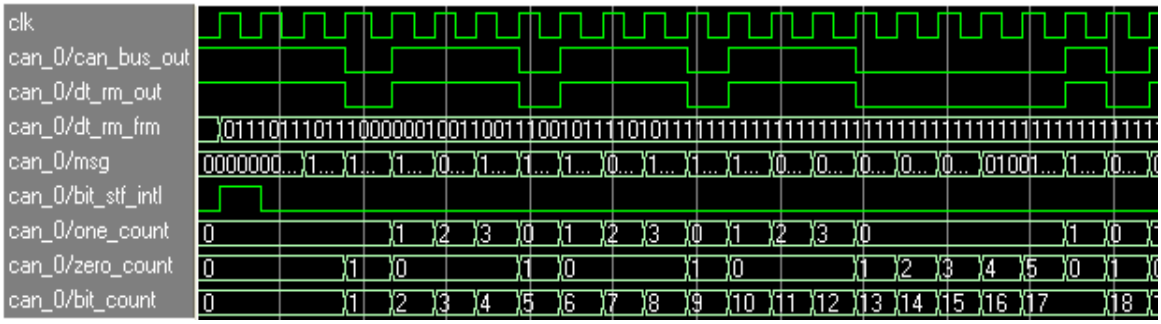

Fig. 5.6 The bit stuffing Mechanism

5.1.2.5 Bus Arbitration

The arbitration process for CAN is also dependent on the Wire-ANDing operation performed on the outputs of the respective nodes. In the fig nodes *can_0* and *can_3* compete for bus access. The nodes wait till the bus is free. Both the nodes start transmission as soon as the bus is free. As long as the bit transmitted by the node and the bit received by the node on the CAN bus is the same, the node will continue transmitting. When a node detects a dominant bit on the CAN bus after transmitting a recessive bit during the arbitration period the node loses arbitration. The bus arbitration mechanism is demonstrated in Fig. 5.7.

The message identifier of the two nodes competing for Bus access in the Fig. 5.7 are *can_0* = 111011101110 and *can_3* = 111011101111. The nodes transmit their bits simultaneously. In the above case both the messages have the same identifier but *can_0* is trying to transmit a Data Frame and *can_3* is trying to transmit a Remote Frame.

Since the *RTR* bit is also taken into account when determining the priority, a Data Frame always has a higher priority when compared to a remote frame. When the nodes transmit their *RTR* bit *can_0* transmits a dominant bit and *can_3* transmits a recessive bit. The result of the Wire-And causes the Bus to go to the dominant state. Thus *can_0* wins the arbitration and continues to transmit the message, with its arbitration status *arbtr_sts*, continuing to remain high. *can_3* loses arbitration and its *arbtr_status* signal is asserted low. The high assertion of the *msg_due_tx* signal indicates that the node has a message due for transmission.

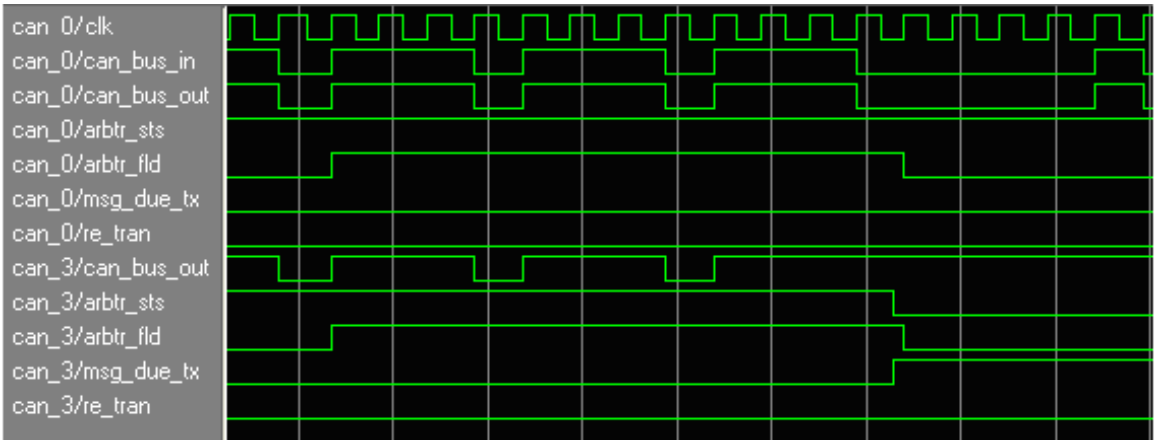

Fig 5.7 Bus Arbitration

Once the node, that lost arbitration receives last bit of Inter Frame Space the node can compete for arbitration. The Fig. 5.8 shows that *can_1* and *can_2* lost arbitration earlier and the *arbtr_sts* signal is asserted low and the *msg_due_tx* signal is asserted high for these nodes. On receiving the last bit of IFS the *arbtr_sts* signal is asserted high. The *re_tran* signal is asserted high for one clock cycle, if there is a message due for transfer.

The *re_tran* signal initiates the re-transmission of the message which lost arbitration earlier. On detecting a high on the *re_tran* signal the message stored in the *dt_rm_frm* register is bit stuffed and transmitted on the CAN Bus.

Another round of arbitration is performed and the message with the highest priority gets through again leaving the messages with lower priority to contend for the bus in the subsequent cycles. The retransmission mechanism is demonstrated in Fig. 5.8. In the Fig. 5.8 case the message identifiers of the two nodes competing for Bus access are

can_1 = 111101111110 and *can_2* = 111101101111. The node *can_2* wins the arbitration by transmitting a dominant bit at the 4th least significant bit of the identifier. The node *can_1* contends for access in the subsequent cycles.

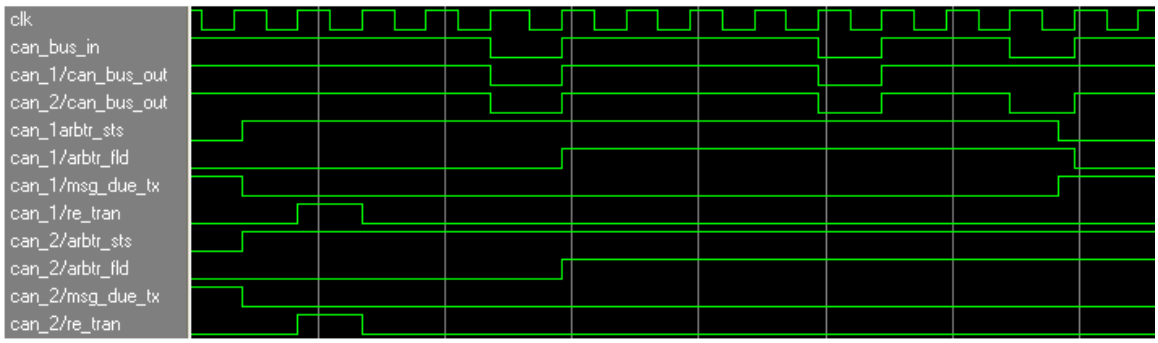

Fig 5.8 Re-transmission of Message

5.1.2.6 Acknowledgement

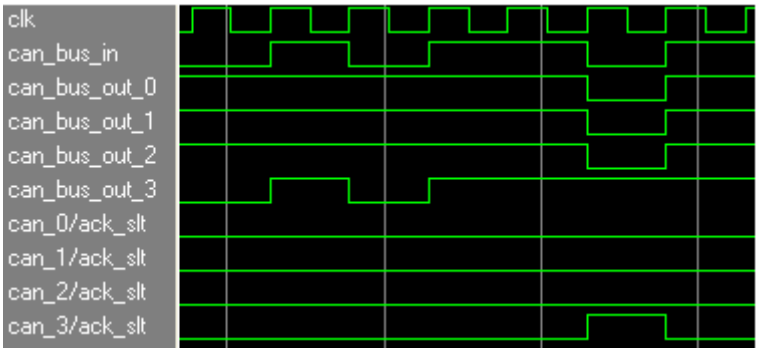

Fig 5.9 Acknowledgement

During the transmission of the acknowledgement slot a transmitter transmits a recessive bit and expects to receive a dominant bit. The acknowledgement slot follows the *crc_dlm* bit. Referring to Fig. 5.9 the bit time for ACK Slot is indicated by the high assertion of the *ack_slt* signal. This signal remains high throughout the bit time of the ACK Slot. During the ACK slot the transmitting node *can_3* in this case, transmits a recessive bit, the other nodes in the network namely *can_0, can_1 and can_2*, transmit a dominant bit if it has been an error-free reception till that bit. With the bit on the CAN bus being the Wire-AND of the bits transmitted by all the nodes on the network, the bus bit sampled back by the transmitting station Node *can_3* is logic low.

5.1.2.7 Signaling Transmission Success

The successful transmission of the message is indicated by the *tx_success* signal going high for one clock cycle. On successful transmission of the message the buffers are reset, the *tx_buffer_busy* signal goes low and the controller is ready to receive the next message for transmission. In case of an error during transmission the buffers are not reset and will continue to hold on to the current value until the message is transmitted successfully. The signaling of successful transmission is demonstrated in Fig. 5.10.

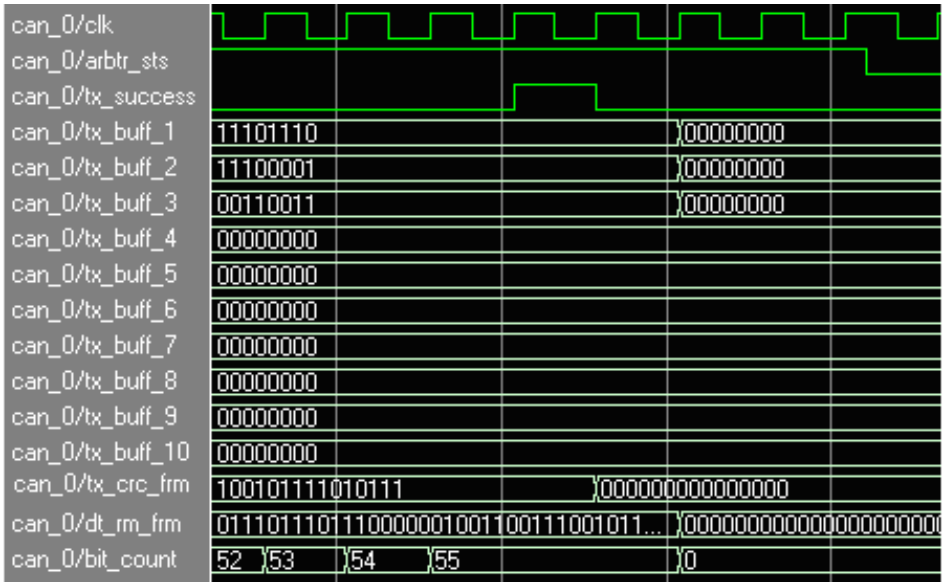

Fig 5.10 Signaling Successful Transmission of Message

5.1.2.8 Bit synchronization

Initially the CAN Bus is in idle state this is indicated by the variable *bus_idle* being logic high. The *arbtr_sts* of the node is high during the bus idle state. The bit synchronization commences when a dominant bit is received during the bus idle state. This indicates that a new message is being transmitted by a node on the CAN bus. This dominant bit is the Start of Frame (SOF) bit of a message. The bit synchronization mechanism is demonstrated in Fig. 5.11.

If the node is not the transmitter of the message the *arbtr_sts* signal of the node is asserted low. The *bus_idle* signal is asserted low to indicate that the bus is no longer idle. The *bit_destf_intl* signal is asserted high to initialize the de-stuffing process. The system

clock is synchronized with the incoming bit stream on detecting the SOF bit. The initial synchronization with the SOF is the hard synchronization discussed in Chapter II. After the hard synchronization the system clock is synchronized with the bit stream at every recessive to dominant transition in the bit stream. The transitions are monitored by the *edge_detected* signal which is asserted high when the *sync_bit* and the *can_bus_in* are of opposite polarity. The sampling point is adjusted to synchronize the system clock to the bit stream. The *can_bus_i*n is sampled at the negative edge of the *sampling_pt.* At every positive edge of the system clock the sampled bit is loaded onto the *serial_in* register. The *serial_in* register is then de-stuffed to extract the message from the bit stream.

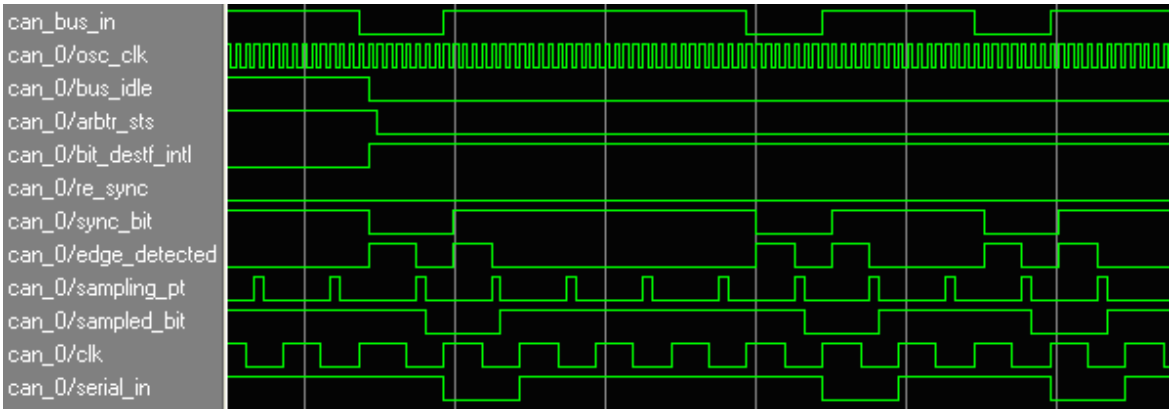

Fig. 5.11 CAN Bit Synchronization

5.1.2.9 Bit De-stuffing

The *bit_destf_intl* signal is asserted high to initialize the de-stuffing process. The message stream received is then de-stuffed to get the respective fields of the transmitted message. The *serial_in* register holds the synchronized stuffed serial bit stream received by the node.

A stuff bit of opposite polarity follows a sequence of 5 consecutive bits of the same polarity. Two counters *one_count* and *zero_count* keep tab of the number of consecutive ones and zeros encountered respectively in the message stream. If the counters value is less than five, the serial input is shifted into the least significant bit position of the *destf_out* register. If one of the counters value equals five then the next bit in the serial bit stream is the stuff bit and is ignored. A count of the number of received

de-stuffed bits is maintained in *rcvd_bit_coun*t this count is not incremented when a stuff bit is encountered. The bit de-stuffing mechanism is demonstrated in Fig. 5.12.

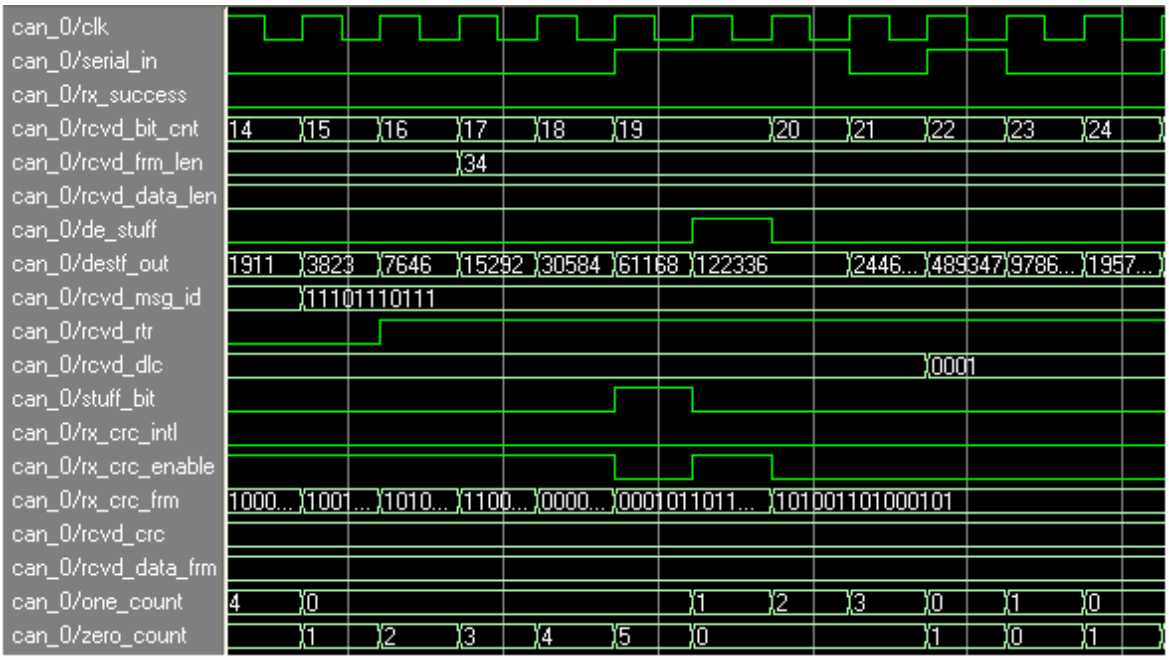

Fig. 5.12 Bit de-stuffing and CRC calculation

A sequence of five consecutive zeros is detected. The next bit in the bit stream is the stuff bit which is recessive in this case. This bit is not shifted into the least significant bit position of the *destf_out* register. The signals *de_stuff* and *stuff_bit* are asserted logic high on detecting a stuff bit in the bit stream. The *rcvd_bit_coun*t is also not incremented.

The information from the de-stuffed bit stream is extracted from the *destf_out* register. When the *rcvd_bit_coun*t equals a particular value a certain number of least significant bits of the *destf_out* register are loaded into the corresponding receive registers. For example when the receive bit count equals 14 the eleven least significant bits of the *destf_out* register are loaded onto the *rcvd_msg_id* register. These bits represent the message identifier of the received message. Similarly when the *rcvd_bit_coun*t equals 15 the least significant bits of the *destf_out* register gives the RTR of the received message. This is stored in the *rcvd_rtr* register. Similarly the *rcvd_dlc, rcvd_frame_len, rcvd_data_len, rcvd_data_frm, rcvd_crc_frm* can all be extracted from the bit stream.

The signals *de_stuff* is asserted high on detecting a stuff bit in the bit stream. The *de_stuff* signal is used to cover corner cases where a stuff bit occurs at a value of *rcvd_bit_coun*t where a register is to be loaded. The *rcvd_bit_coun*t does not increment on detecting a stuff bit. Under this circumstance it is possible for an unintended value to be loaded into one of the registers. To ensure this does not happen the registers are loaded only if the corresponding *rcvd_bit_coun*t value is reached and the *de_stuff* signal is asserted low.

The generation of the CRC sequence for the received message is initialized on detecting the SOF bit. The *rx_crc_enable* signal is asserted high on detecting the SOF bit. The *stuff_bit* signal indicates the presence of a stuff bit by going high. On detecting logic high on the *stuff_bit* signal the *rx_crc_enable* signal is asserted low. The *rx_crc_enable* signal is asserted low on detecting a stuff bit in the incoming bit stream as the CRC sequence is calculated on the un-stuffed bit stream on the transmitter side. The CRC sequence is calculated till last bit of Data Field in case of a Data Frame or till the last bit of Control Field in case of a Remote frame.

5.1.2.10 Signaling Reception Success

The successful reception of the message is indicated by the *rx_success* signal going high for one clock cycle, as seen in Fig. 5.13. On successful transmission of the message all the registers are reset and the controller is ready to receive the next message. In case on an error during reception the message is ignored and the registers are reset.

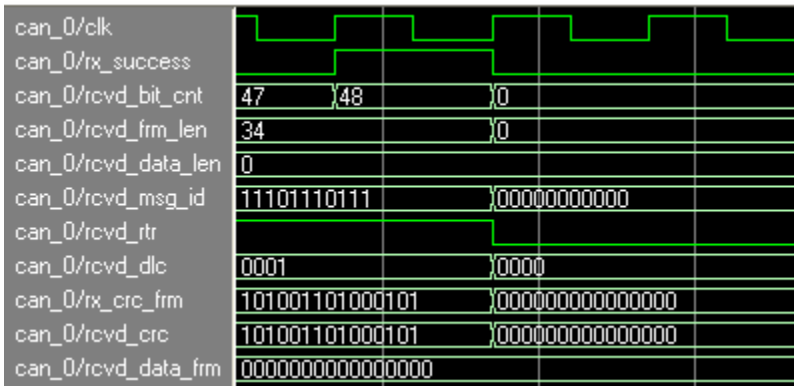

Fig. 5.13 Signaling Successful Reception of a Message

5.1.2.11 Acceptance Filtering

The acceptance filter for node *can_0* is designed to accept only messages with the message identifier value equal to 11'b11101110111. On receiving the last bit of the EOF field the *rcvd_lst_bit_eof* signal is asserted high. The signals *bt_ack_err_pre* and *stf_frm_crc_err_pre* indicate if there has been an error during the reception of the message. If there was an error during the reception of the message the corresponding signal is asserted high. If the received message identifier passes the filter, and if there have been no errors during the reception of the message, the message is accepted. The acceptance is indicated by asserting the *acpt_sts* signal high. The acceptance filtering mechanism is demonstrated in Fig.5.14.

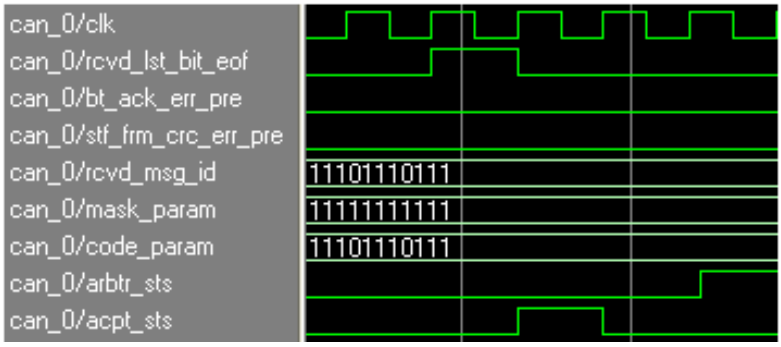

Fig. 5.14 Acceptance Filtering

5.1.2.12 Receive Buffer Storage

There are two receiving buffers *rx_buff0* and *rx_buff1* each ten bytes deep. The *rx_buff_0_wr_stat* and *rx_buff_1_wr_stat* signals act as status signals. These signals are asserted high if the buffers are free and are ready to accept a new message. The *rx_buff_0* buffer is always checked first, only if it is written into does the node check the *rx_buff_1*. The signals *rx_buff_0_wrtn* and *rx_buff_1_wrtn* are asserted high if the corresponding buffer is written.

On detecting an active high signal on the *acpt_sts* signal the node checks the buffers and writes into the buffer that is free at the next clock cycle. This way even if a new message is received the previous message is not overwritten. The *rx_buff_0_wrtn* and *rx_buff_1_wrtn* act as status signals. The corresponding signal is asserted high the buffer

is written into. The data is written into the buffer in the following order the received message Id given by *rcvd_msg_id*, *rcvd_rtr*, the *rcvd_dlc* and the *rcvd_data_frm* with the MSB first. Once the data is loaded on to the buffer the *buff_rdy* signal is asserted high to indicate to the host controller that there is a message to be read from the receive buffers. Fig 5.15 demonstrates the loading of the buffers.

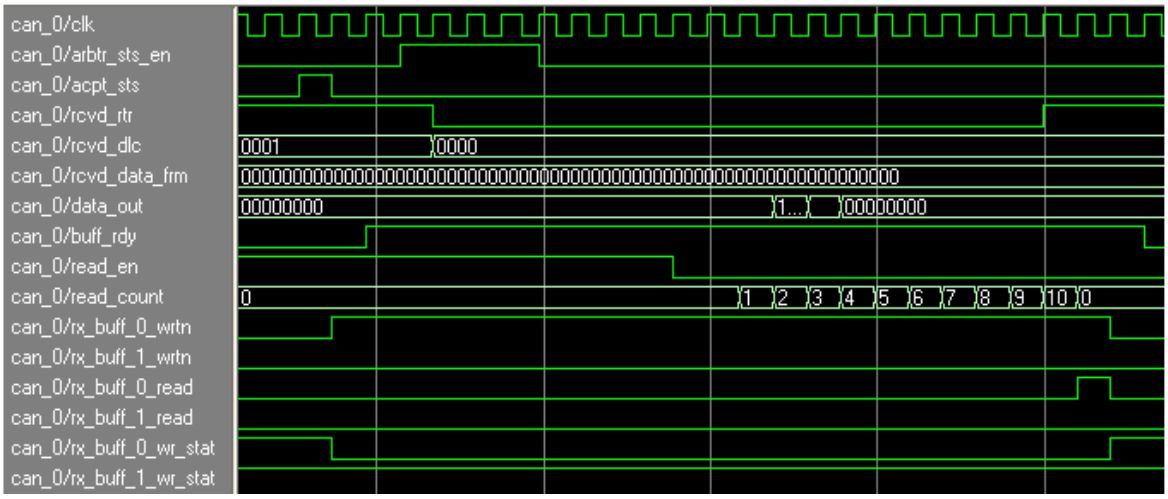

Fig. 5.15 Receive Buffer Storage

5.1.2.13 Host Interface for Data Reception

On detecting a high on the *buff_rdy* signal the host controller initiates the read operation by asserting the *rd_en* signal low. Node *can_0*, on receiving a logic low *rd_en* puts out the contents of the buffers starting form *rx_buff0 [0]* to *rx_buff0 [9]* if it is *rx_buff0* that is written into or *rx_buff1 [0]* to *rx_buf10 [9]* if it is *rx_buff1* that is written into. The data is sent out on the *data_out* bus one byte at a time. A counter *read_count* is used to keep count of the number of bytes read from the buffer.

Once a buffer is read from the *rx_buff_0_read* or the *rx_buff_1_read* signal is asserted high for the corresponding buffer. With the high assertion of the buffer read signal the write status of the buffer is asserted high and the buffer written status signal is asserted low. The contents of the corresponding buffer are also reset and the buffer is ready to take in a new message. The *rd_en* signal is also asserted high once the message has been read.

Fig. 5.16 demonstrates the interface between the host controller, and the CAN controller for data reception.

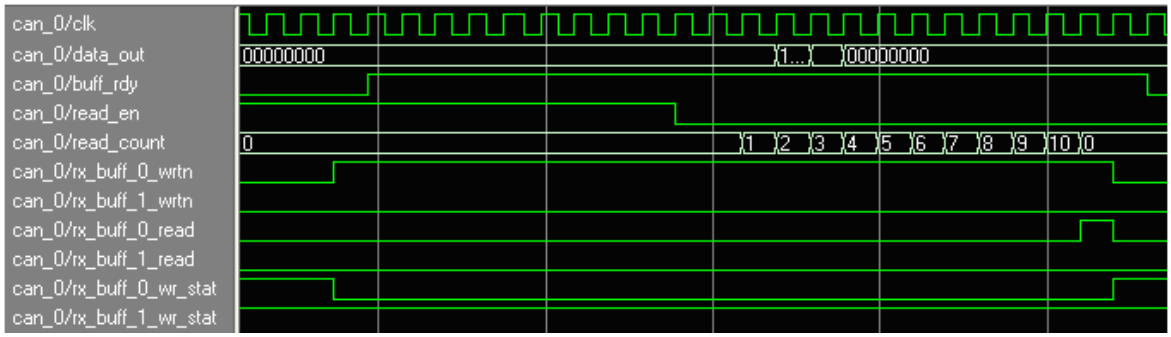

Fig. 5.16 Host Interface for Data Reception

5.1.2.14 Error Signaling and Error Frame Generation

The transmission and reception of an error frame is verified by inducing errors in the data frame being transmitted. Each of the five different types of error are dealt with separately and demonstrated in the following discussion.

5.1.2.14.1 Acknowledge and Form Error

If the transmitting node receives a recessive bit in the acknowledgement slot it is understood that none of the nodes in the network received the message correctly and an ACK error is signaled.

An acknowledgement error is induced by forcing the *can_bus_out* of the three receiving nodes in the system to remain in their recessive states during the acknowledgement slot. This simulates a condition where in the receiving nodes have received an erroneous message frame. This causes the transmitting node to detect an acknowledge error. The presence of the acknowledge error is indicated by asserting the *ack_err* signal high. The transmitting node *can_0* proceeds to transmit an active error frame, indicated on Fig. 5.17 by the signal *act_err_flg_tx*.

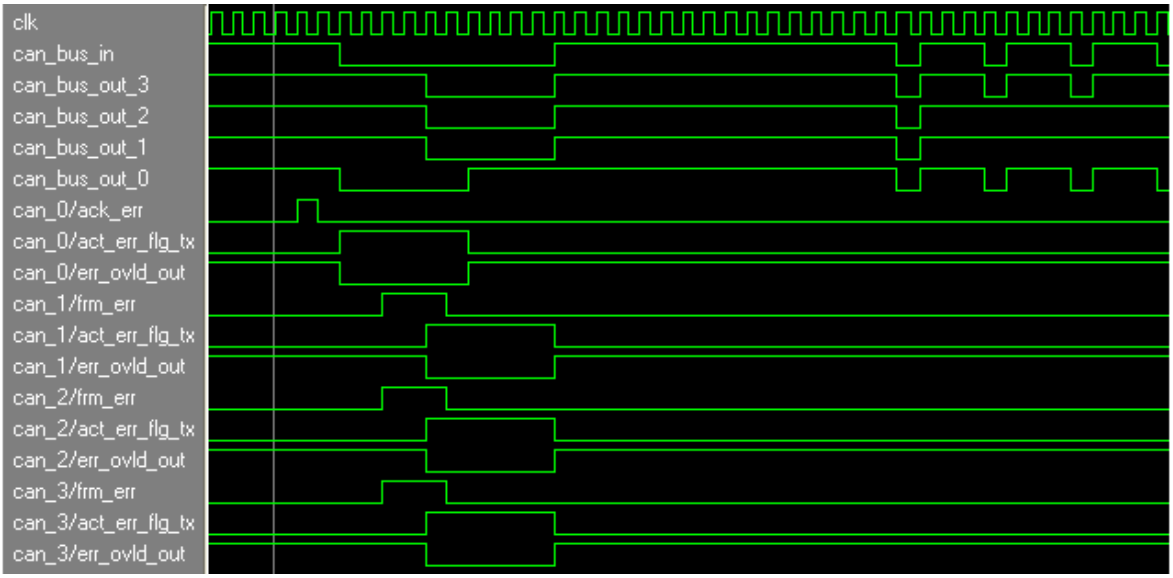

Fig. 5.17 Error Frame Due to Acknowledgement and Stuff Error

Due to the transmission of the Error Frame by the transmitting node the other nodes in the network detect a form error and transmit their own Error Frame. The presence of the form error is indicated by asserting the *frm_err* signal high. On completion of the Error Frame Inter Frame Space, the Nodes compete for the transmission of the next message on the CAN bus.

5.1.2.14.2 CRC and Form Error

A CRC error is caused if a bit being transmitted on the bus changes from the value which was initially used by the transmitting node to calculate the CRC frame. The CAN controller on receiving the CRC frame of the message performs the CRC sequence comparison. If the received CRC frame and the generated CRC do not match, a CRC error is flagged by asserting the *crc_err* signal high.

Considering the transmission of the same data frame as in the earlier case, a CRC error is induced by complementing one of the bits of the received CRC frame. This will ensure that the received CRC and the generated CRC sequence do not match. Due to this, the receiver node can*_1* detects a CRC error due to the erroneous sequence. The node *can_1* proceeds to transmit an active Error Flag. As in the previous cases the other nodes

in the network receive the Error Frame and start generating their own Error Frame due to the violation of one of the rules. In the above Fig. 5.18 the violation in the other nodes is due to the violation of the form fields, indicated by the high assertion of the *frm_err* signal.

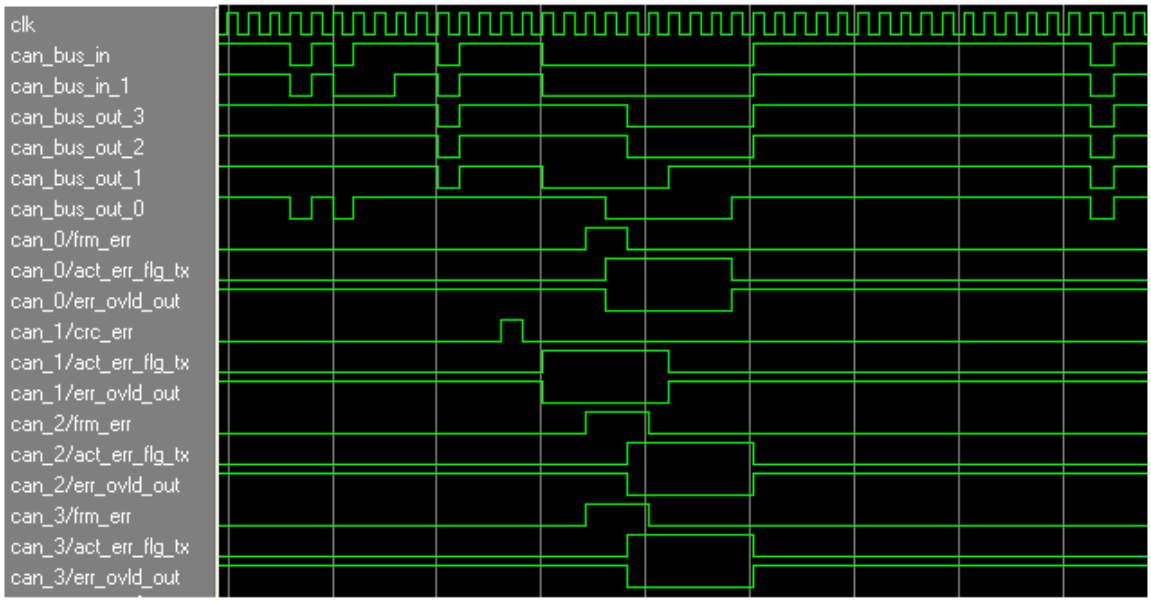

Fig. 5.18 Error Frame Due to CRC and Form Error

5.1.2.14.3 Bit and Stuff Error

A bit error is induced if a transmitter sends a dominant bit but detects a recessive bit on the bus line, or sends a recessive bit and detects a dominant bit on the bus line. There a few exceptions to the bit error rule which were discussed in Chapter III.

To demonstrate this, a bit error is induced in the transmitting node by complementing the *can_bus_in* value. The transmitting node is *can_0.* The node detects a bit error at the complemented bit position and since it's not the Arbitration or Acknowledgement field that is being transmitted sends out an Active Error Frame. The presence of the bit error is indicated by asserting the *bt_err* signal high. The other nodes in the network detect a stuff error due to a sequence of six consecutive dominant bits and transmit their own Error Frame. The presence of the stuff error is indicated by asserting the *stf_err* signal high. On completion of the Error Frame IFS, the Nodes compete for the transmission of the next message on the CAN bus as seen in Fig 5.19.

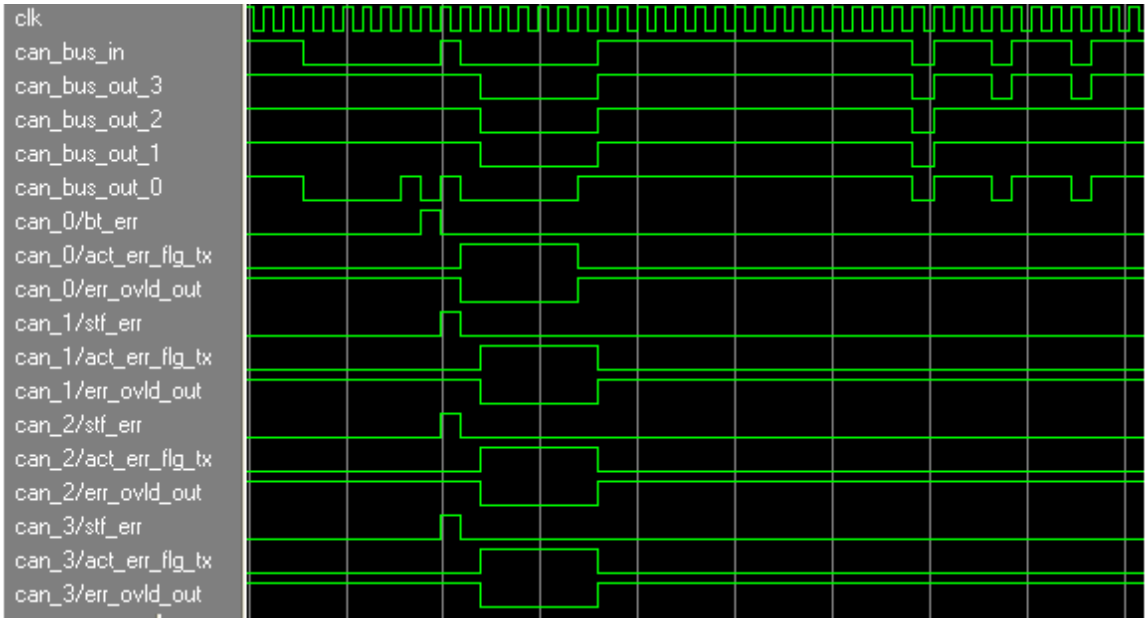

Fig. 5.19 Error Frame Due to Bit and Stuff Error

5.1.2.15 Error Confinement

To implement the Error Confinement mechanism, CAN makes use of two error counters, one to keep track of transmit errors (Transmit Error Counter) and the other to keep track of receive errors (Receive Error Counter). In the Fig. 5.20 an acknowledge error is detected and the Transmit Error Counter of the transmitting station Node *can_0* tx*_err_cntr* is incremented by 8. The Acknowledge error detected by Node can is shown by *ack_err*.

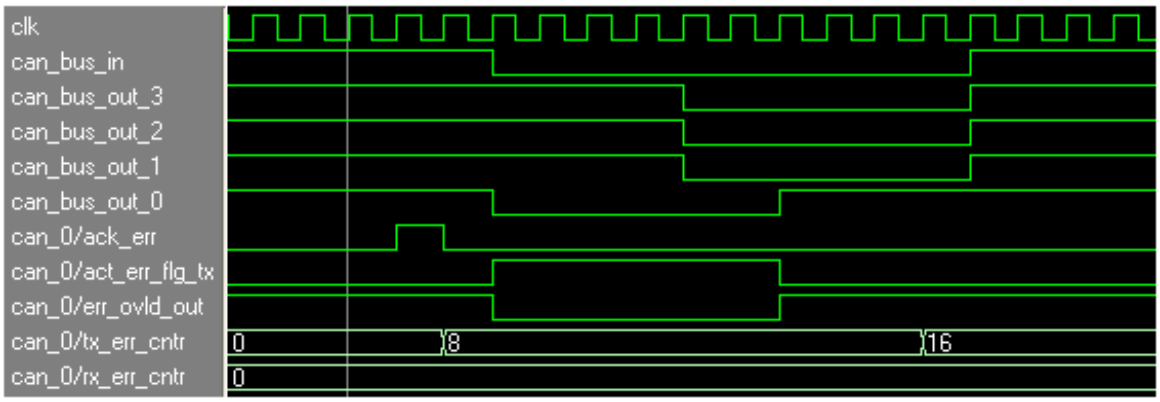

Fig. 5.20 Error Limitation Transmitter

Since the first bit on the *can_bus_in,* after the transmission of the error flag is a zero the node understands that the error is local, hence the *tx_err_cntr* is further augmented by 8. When a node successfully transmits a message and receives a high *tx_success* signal the *tx_err_cntr* is decremented by 1.

Similarly a receiver's error counter the *rx_err_cntr* is incremented when the receiver detects an error. The incremental value for the receiver is comparatively low. In the Fig. 5.21 the receiver detects a stuff error in the bit stream and the Receive Error Counter of the receiving node the *rx_err_cntr* is incremented by 1. Since the first bit after the transmission of the Error Flag is a recessive bit the node understands that the error was does not increment its error counter. This is due to the fact that another node had already initiated its Error Flag by the time the node *can_1* initiates its error flag. The *rx_err_cntr* is decremented by 1 on successful reception of a message indicated by the high assertion of the *rx_success* flag.

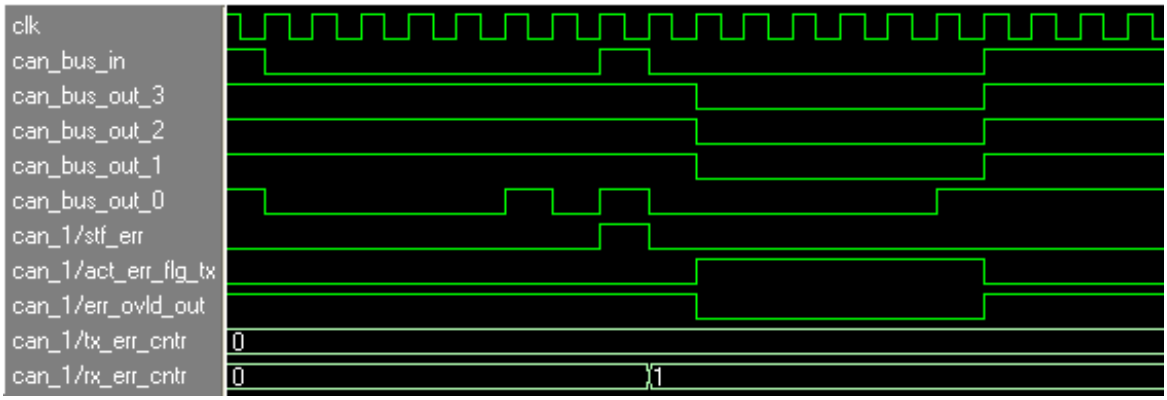

Fig. 5.21 Error Limitation Receiver

5.1.2.16 Overload Signaling and Overload Frame Generation

A message received when both the buffers are full cannot be stored in the receive buffers and will be lost. To avoid this situation an Overload Frame is generated by the receiver to indicate an overload condition to the other participating nodes. The transmission of the next message on the Bus is delayed by the transmission of the Overload Frame. The overload signaling and overload frame generation is demonstrated in Fig. 5.22.

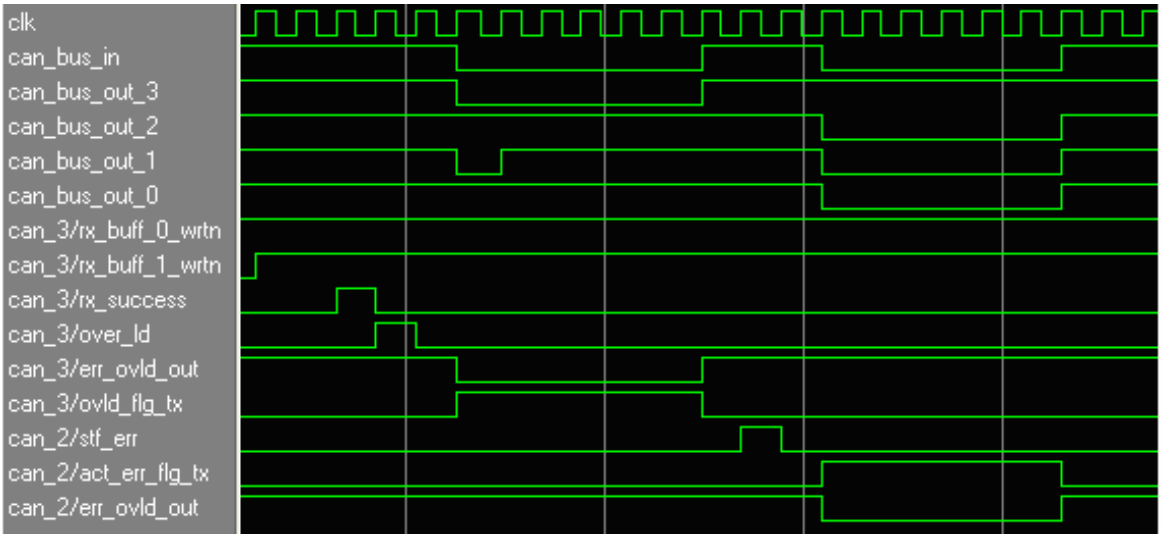

Fig. 5.22 Overload Signaling and Overload Frame Generation

As discussed earlier in the Receive Buffer Storage section the *rx_buff_0_wrtn* and *rx_buff_1_wrtn* signals indicate that the buffers have been written into when they are asserted high. When both the buffers are loaded the node asserts the *over_ld* flag high to indicate an overload condition. The high assertion of the *over_ld* signal initiates the transmission of an Overload Frame. This is indicated by the *ovld_flg_tx* flag going high. The other nodes in the network detect a stuff error in the arbitration field and send out their respective Error Flag. When a stuff error is detected in the arbitration field the error counters are not incremented. Thus the overload flag delays the transmission of the next frame giving it sufficient time for the host controller to read the data from the receive buffers.

5.2 Formal Verification Results

The Logic Equivalence Check (LEC) was performed using Verplex. The results of the LEC on the RTL code and final post layout netlist match and show proof that they are logically equivalent. The design is a flip flop based design. There are no latches in the design. There are 1113 DFF's in the design. The input and outputs of these flip flops act as the check points. There are no unmapped points, no black boxes and no floating pins in the design. Table 5.1 shows the mapping and the compare statistics of the design.

Table 5.1 Mapping and Compare Statistics

	Compare Result	Golden	Revised
Root module name		CAN	CAN
Primary inputs		15	15
Mapped		15	15
Primary outputs		11	11
Mapped		11	11
Equivalent	11		
State key points		1113	1113
Mapped		1113	1113
Equivalent	1113		

There is Potential loss of RHS MSB or carry-out bit warning for the design. This warning can be safely ignored as this refers to the counters that have been implemented in the design. The counters have been designed such that there is no necessity for a carry-out. Two examples of the warnings are shown below.

RTL1.5b: Potential loss of RHS MSB or carry-out bit

Type: Golden Severity: Warning Occurrence: 47

1: par_ser_conv: count [6:0] LHS: 7 RHS: 8

on line 1079 in file '/proj0/Sreeram/CAN/modelsim/rtl_code/CAN_total.v'

7: bit_stuff: zero_count [2:0] LHS: 3 RHS: 4

on line 1467 in file '/proj0/Sreeram/CAN/modelsim/rtl_code/CAN_total.v'

5.3 Timing Results

The Physical Design of the CAN controller was performed using Magma's Blast Fusion. Magma's Blast Fusion has an inbuilt incremental timing analyzer. The final timing report for the CAN controller is given below. The timing report shows that the post layout timing for the CAN controller satisfies the timing requirements specified in the constraint file.

The Best case timing analysis on the design shows a positive slack of 157ps. This

indicates that the design exceeds the timing requirements for hold. Table 5.2 shows the best case timing report for the design.

Table 5.2 Best Case Timing Report

```
################################################################################
# Mantle analysis report
# Command:
#         report timing summary \
#              /work/CAN/CAN \
#              -append CAN_timing_rpt.txt
# Date:    Tue May 30 13:51:20 2006
# Version: mantle version 2005.03.81-linux24_x86_64
################################################################################
Cell count                    7546
Node count                   21354
Event count                  41330
Endpoint count                1123

Worst early slack              157
Total Negative early slack       0
Failing endpoints                0

Path Summary
        start point                  start edge        end point                end edge  slack  delay
------------------------------------ ----------  ---------------------------  --------  -----  -----
tx_buffer.data_reg_in_reg[0]/Q   FALL  tx_buffer.data_reg_reg[0]/D  FALL    157    141
```

The worst case timing analysis on the design shows a positive slack of 157ps. This indicates the design exceeds the timing requirements for setup. Table 5.3 shows the worst

case timing report for the design.

Table 5.3 Worst Case Timing Report

```
##########################################################################
# Mantle analysis report
# Command:
#         report timing summary \
#              /work/CAN/CAN \
#              -append CAN_timing_rpt.txt
# Date:    Tue May 30 13:51:23 2006
# Version: mantle version 2005.03.81-linux24_x86_64
##########################################################################

Cell count                 7546
Node count                21354
Event count               41330
Endpoint count             1123

Worst late slack          21644
Total Negative late slack     0
Failing endpoints             0

Path Summary
          start point          start edge       end point       end edge    slack    delay
-----------------------------------  ----------  ----------------  ----------  ---------  ------
arbtr_sts_ctrl.arbtr_sts_reg/Q       FALL        can_bus_out       FALL        21644      2793
```

5.4 DRC Report

The DRC report generated by Magma Blast Fusion is given below. Magma Blast Fusion checks for the design rules specified in the technology library. The tool on

detecting Design Rule violations clears these violations incrementally. The incremental method to clear DRC violations ensures that new violations do not crop up because of the reroute done to clear the current violations. Table 5.4 shows the DRC report for the design.

Table 5.4 Design Rule Check

POST-121 ----------

POST-121 Summary of short and spacing violations:

POST-121 statistics on shorts and spacing violations:

POST-121

POST-121 * different-net

POST-121 * short and spacing violation

POST-121 * involving at least one regular wire or via

POST-121 ----------

POST-126 no violation is reported

POST-975 ----------

POST-975 Summary of violations (DRC/attention rectangles)

POST-975

POST-975 SPCE: preroute spacing SHRT: preroute short ISPC: preroute interlayer spacing

POST-975 offg: off grid ispc: interlayer spacing spce: regular spacing

POST-975 shrt: regular short wdth: width ntch: notch

POST-975 dgbo: dogbone open: open ilnd: island

POST-975 dgnl: diagonal width hole: hole mprt: multiport

POST-975 pwro: power open viar: via reliability viav: via-to-via

POST-975 dupl: duplicate shed: short-edge prot: protrusion

POST-975 viso: insufficient via overhang

POST-975

POST-975 * the following categories are not shown (no violations):

POST-975 {SPCE SHRT ISPC offg ispc spce shrt wdth ntch dgbo open ilnd dgnl hole mprt pwro viar viav dupl shed prot viso}

POST-975 * the following categories are shown (with violations):
POST-975 {}
POST-975 ----------

5.5 Model Report

The Magma Blast Fusion generates a model report with information on the type and number of cells used. The cell statistics report shows that there are 1113 Flip flops, 3694 Boolean cells, 236 buffer cells and 449 inverter cells for a total of 5492 Standard cells occupying a total area of 0.069 mm^2. No macros have been used in the design. A total of 2054 filler cells occupying an area of 0.008 mm^2, have been used in the design. The total area occupied by the cells is 0.076 mm^2. The total cell row utilization is 84.7% and the total utilization is 64.1%. The model report also provides model information, Floorplan information, net statistics, pin statistics and wire statistics. Table 5.5 shows the Model report for the design.

Table 5.5 Model Report

CK-7 Report for model /work/CAN/CAN
CK-5 Collecting data on model CAN
---------- M O D E L S T A T I S T I C S ----------
Generated for user guest on host qtlxblr2 on Tue May 30 13:51:18 2006
Model: /work/CAN/CAN

Cell Statistics	- count -	- area -
Inverter cells:	449	
Buffer cells:	236	
Boolean cells:	3694	
Flip-Flops cells:	1113	
Standard cell total:	5492	0.069mm2
Hard Macros:	0	0.000mm2
Filler cells:	2054	0.008mm2

Total cells: 7546 0.076mm2

Model Information

Model width, height: 0.327 mm x 0.327 mm = 0.107 mm2

Aspect ratio (w/h): 1.00

Buckets : 100 x 113 = 11300

Total utilization: 64.1 %

Floorplans

Name: /work/CAN/CAN/floorplan: fplan (primary)

Outer shape area : 0.107 mm2 (A)

Inner shape area : 0.081 mm2

Cellrows	: 99	Available area:	0.081 mm2 (a)	
Soft, hard macros	: 0	Total area	:	0.000 mm2 (m)
Pads	: 0	Total area	:	0.000 mm2 (p)
Standard, super cells	: 5492	Total area	:	0.069 mm2 (s)
Fillers, endcaps	: 2054	Total area	:	0.008 mm2

Total utilization : 64.1 % ((s + m + p) / A)

Cellrow utilization : 84.7 % (s / a)

Net Statistics

Number of signal nets: 5553

Number of power/clock nets: 2

Number of cell pins: 21326

Average nets per standard cell: 1.01

Average pins per standard cell: 3.89

Average pins per signal net: 3.84 (maximum = 81)

Pin Statistics

Total	26
Input	11
Output	15

Wire statistics

Layer	-- Segment Statistics --	-- Wire Statistics --
METAL1:	0.000 (0.0%) in 2	0.002 (0.9%) in 4260
METAL2:	0.039 (23.6%) in 23211	0.044 (25.7%) in 32947
METAL3:	0.051 (30.7%) in 12056	0.052 (29.9%) in 12581
METAL4:	0.036 (21.8%) in 4174	0.036 (20.9%) in 4415
METAL5:	0.027 (16.0%) in 1817	0.026 (15.0%) in 1664
METAL6:	0.013 (7.9%) in 502	0.013 (7.6%) in 357
Total:	0.16681 meter in 41762	0.17316 meter in 56224 wires

Signal wire length: 0.173 meter in 56224 wires
Prerouted wires (pwr/clk): 0.034meter in 118 wires (19.6%)
Prerouted segments (pwr/clk): 0.000meter in 0 segs (0.0%)
Average net wire length: 31um in 10.1 wires
Average net segment length: 30um in 7.5 legs
Last track routing overflows: (6.29%)

CHAPTER 6
CONCLUSION AND FUTURE WORK

6.1 Conclusion

The design of the CAN controller for this project has been described in detail in Chapter III and the results obtained from the simulation of the post layout model, the Logic Equivalence Check and the Timing Analysis have been elucidated in Chapter V.

The CAN controller has been implemented into TSMC 0.13μm CMOS technology library. This TSMC technology is a single poly, six metal layers process. Error-free transmission and reception of message frames has been demonstrated with four nodes instantiated on a network each taking turns to transmit messages. The detection and transmission of error frames in the event of an error has been demonstrated by inducing ACK, Stuff, CRC, Bit and Form errors in the messages. All the basic functions involved in the transmission and reception of the message as per CAN 2.0A protocol have been demonstrated and found to be meeting the requirements.

The Logic Equivalence has been checked and the final design is proven to be logically equivalent to the RTL code. The design meets timing requirements specified. The timing sign off has been performed using the Magma design tools. The Graphic Data System (GDS) II format of the design has been exported and the design can go in for fabrication.

Thus the ASIC implementation of a basic protocol controller for CAN 2.0A protocol has been accomplished.

6.2 Future Work

The design of an ASIC for a CAN 2.0A controller has been accomplished. This design could be taken one step further by implementing different operating modes such as the Listen Only mode where the node has no influence on the bus, no active error flags, no overload flags, and no acknowledgement can be sent, but it can receive messages, a Self Test mode and Self Reception Request to enable full node testability without any other active node, and a Sleep mode with reduced power consumption.

The CAN controller is designed to operate at a fixed bit rate of 1M Bits/sec. By providing a Programmable Bit Time Logic with programmable bit rate, the controller can be extended to operate at multiple bit rates. The controller can be further extended by making the propagation segment and phase segment lengths programmable. More functionality can be incorporated into the controller by enabling single or triple sampling of the bit.

The controller has been implemented with two receive buffers. By providing a receive FIFO the load on the host controller can be considerably reduced. This would ensure that the CPU has more time to take care of other system related functions.

The timing of the design has been verified using Magma's tools which are not the industry sign off tool. The timing sign off can be performed using the industry sign off tool, Synopsys PrimeTime. The DRC and LVS checks have been performed using Magma's tools as well; these could be cross checked using the LVS and DRC industry sign off tool, Mentor Graphics' Calibre.

These features will add to the final quality of the ASIC. The optimized hardware for the CAN controller can be fabricated, making the hardware available for further use.

LIST OF REFERENCES

1 *CAN Specification Version 2.0*, Robert Bosch GmbH, Stuttgart, Germany, 1991.

2 Wolfhard Lawrenz, *CAN System Engineering: From Theory to Practical Applications*, Springer-Verlag, 1997, ISBN 0–387–94939-9.

3 Daniel Mannisto, Mark Dawson, *An Overview of Controller Area Network (CAN) Technology,* mBus, 2003.

4 Steve Corrigan, *Introduction to the Controller Area Network (CAN) - Texas Instruments Application Report, SLOA101,* 2002.

5 Karthik Ranganathan, *RTL System Design of CAN 2.0A Controller,* Master's Thesis, Texas Tech University, Lubbock, TX, 2005.

6 ISO. International Standard *ISO 11898: Road Vehicles – Interchange of digital information – Controller Area Network (CAN) for high speed communication,* ISO 11898:1993[E], 1993.

7 Karl Henrik Johansson, Martin Torngren, Lars Nielsen, *Vehicle Applications of Controller Area Network, 2005.*

8 Stuart Robb, *CAN Bit Timing Requirements – Motorola Semiconductor Application Note, AN1798,* 1999.

9 Lars-Berno Fredriksson, *Controller Area Networks and the protocol CAN for machine control systems,* Kvaser AB, P.O. Box 4076, S-511 04 Kinnahult, Sweden.

10 Peter Bagschik, *An Introduction to CAN, I+ME ACTIA GmbH, ISO* 7498, 1998.

11 Blagomir Donchev, Marin Hristov, *Implementation of CAN Controller With FPGA Structures, 7th International Conference, CADSM, 2003.*

12 Florian Hartwich, Armin Bassemir, *The Configuration of the CAN Bit Timing, 6th International CAN Conference,* Robert Bosch GmbH, Abt. K8/EIS Tübinger Straße 123, 72762 Reutlingen.

13 *http://www.can-cia.org/can/protocol, 2004, CAN Protocol,* Home page of the organization CAN In Automation (CiA).

14 Jose Rufino*, An Overview of Controller Area Network,* Instituto Superior Técnico

- Universidade Técnica de Lisboa, Avenida Rovisco Pais – 1096 Lisboa Codex – Portugal.

15 Namsub Kim, Dawi Kim, Kyuhyung Cho, Jinsang Kim, Wonkyung Cho, *Design and Verification of a CAN Controller for Custom ASIC,* CAN in Automation, iCC, 2005.

16 Michael John Sebastian Smith, *Application-Specific Integrated Circuits,* Addison-Wesley Longman Publishing Co., Inc., 1997, ISBN 0-201-50022-1.

APPENDIX A

CONTROLLER AREA NETWORK

```
// Controller Area Network top module
module CAN (osc_clk, g_rst, init_err_st, can_bus_in, param_ld, data_in,
tx_buff_ld, rd_en, tx_buff_busy, can_bus_out, buff_rdy, data_out);

input osc_clk;              // oscillator clock

input g_rst;                // global reset

input init_err_st;          // signal to reset error counters

input can_bus_in;           // input form CAN bus

input param_ld;             // Signal to load register files

input [7:0] data_in;        // data input bus

input tx_buff_ld;           // load tx_buff

input rd_en;                // signal to read the RX buffers

output tx_buff_busy;        // status signal to indicate that buff is
                               loaded

output can_bus_out;         // output from CAN controller

output buff_rdy;            // signal to indicate that the receive
                            // buffer is loaded

output [7:0] data_out;      // data output bus

wire frame_gen_intl;        // buffer status full = 1 empty =0

wire [10:0] mask_param;     // mask parameter for id acceptance
                               checking

wire [10:0] code_param;     // code parameter for id acceptance
                            // checking

wire [1:0] sjw;             // re-synchronizing jump width

wire [7:0] tx_buff_1;       // tranmit buffers

wire [7:0] tx_buff_2;       // tranmit buffers

wire [7:0] tx_buff_3;       // tranmit buffers

wire [7:0] tx_buff_4;       // tranmit buffers

wire [7:0] tx_buff_5;       // tranmit buffers
```

```
wire [7:0] tx_buff_6;        // tranmit buffers
wire [7:0] tx_buff_7;        // tranmit buffers

wire [7:0] tx_buff_8;        // tranmit buffers

wire [7:0] tx_buff_9;        // tranmit buffers

wire [7:0] tx_buff_10;       // tranmit buffers

wire rtr;                    // remote transfer request bit

wire [3:0] dlc;              // data length code register

wire tx_crc_frm_cmp;         // indicates the completion of crc frame

wire [14:0] tx_crc_frm;      // transmitter data crc_frame

wire [82:0] par_ser_data;    // par data input to par to series
                             // converter module

wire par_ser_intl;           // initialize par to series conversion

wire [97:0] dt_rm_frm;       // output frame to bit stuffing module

wire bit_stf_intl;           // output to initialise bit stuffing
                             // module

wire [7:0] dt_rm_frm_len;    // data frame length

wire tx_serial_out;          // serial data output to crc module

wire tx_crc_enable;          // enable signal to tx crc module

wire tx_crc_intl;            // initialization signal to tx crc module

wire [1:0] err_state;        // indicates error state of controller
                             // (Active / Passive /Bus off)

wire arbtr_sts;              // signal to indicate if the node is a
                             // transmitter or receiver

wire abort_dt_rm_tx;         // Signal to abort data transmission

wire re_tran;                // signal to resume data transmission

wire dt_rm_out;              // serial output from the data/remote
                             // frame generator module

wire dt_rm_frm_tx;           // indicates the transmission of
                             // data/remote frame

wire arbtr_fld;              // indicates the transmission of arbtr fld
```

```
wire dt_rm_eof_tx_cmp;      // indicates the end of transmission of
                            // data/remote frame

wire ack_slt;               // indicates the transmission of
                            // acknowledgement slot

wire txed_lst_bit_ifs;      // indicates the transmission of last bit
                            // of the inter frame space

wire ifs_flg_tx;            // indicates transmission of ifs_flg

wire [1:0] tx_bit;          // register to hold previously transmitted
                            // bits

wire bt_err;                // indicates the occurrence of a bit error

wire ack_err;               // indicates the occurrence of an
                            // acknowledgement error

wire stf_err;               // indicates the occurrence of a stuff err

wire frm_err;               // indicates the occurrence of a frame err

wire crc_err;               // indicates the occurrence of a crc err

wire over_ld;               // indicates the occurrence of an over ld
                            // condition

wire sampling_pt;           // the sampling point of the can bus data

wire sampled_bit;           // register to store the sampled bit

wire tx_success;            // indicates the successful transmission
                            // of message

wire rx_success;            // indicates the successful reception of
                            // message

wire tx_eof_success;        // indicates the successful transmission
                            // of msg till the end of frame

wire rx_eof_success;        // indicates the successful reception of
                            // msg till the end of frame

wire err_ovld_out;          // serial output from the error/overload
                            // frame generator module

wire act_err_frm_tx;        // indicates active error frm transmission
                            // stays high till end of frm

wire psv_err_frm_tx;        // indicates passive err frm transmission

wire ovld_frm_tx;           // indicates overload frm transmission
```

```
wire act_err_flg_tx;          // indicates active err flag transmission

wire psv_err_flg_tx;          // indicates passive err flag transmission

wire ovld_flg_tx;             // indicates overload flag transmission

wire cons_zero_flg;           // indicates the reception of 0's after
                              // the transmission of an error flag

wire ovld_err_ifs_tx;         // indicates error/overload inter frame
                              // space transmission

wire ovld_err_tx_cmp;         // indicates successful transmission of an
                              // error frame

wire bus_off_sts;             // indicates that the controller is in
                              // bus off state

wire send_ack;                // indicates the need to send out an
                              //acknowledgement

wire rcvd_lst_bit_eof;        // indicates the reception of the last bit
                              // of the end of frame flag

wire rcvd_eof_flg;            // indicates the reception of end of frame

wire [6:0] rcvd_data_len;     // register to store received data length

wire [6:0] rcvd_bt_cnt;       // counter to count the total number of
                              // bits received

wire bt_ack_err_pre;          // indicates the presence of a bit or
                              // acknowledgement error

wire stf_frm_crc_err_pre;     // indicates the presence of a stuff, CRC
                              // or form error

wire txmtr;                   // indicates if the controller is/was a
                              // transmitter/receiver 1 -indicates a tx
                              // and 0 a receiver

wire msg_due_tx;              // indicates that a message is due for
                              // transmission

wire serial_in;               // register to hold the synchronized
                              // sampled bit

wire rx_buff_0_wrtn;          // indicates buffer 0 has been written

wire rx_buff_1_wrtn;          // indicates buffer 1 has been written

wire rcvd_lst_bit_ifs;        // indicates the reception of the last bit
                              // of the inter frame space
```

```
wire clk;                    // generated system clock

wire bit_destf_intl;         // indicates the initialization of the bit
                             // stuff module

wire [14:0] rx_crc_frm;      // register to hold the generated crc
                             // frame for the received message

wire [10:0] rcvd_msg_id;     // register to hold the received message
                             // identity

wire rcvd_rtr;               // register to hold the received remote
                             // transfer bit

wire [3:0] rcvd_dlc;         // register to hold the received data
                             // length code

wire [14:0] rcvd_crc;        // register to hold the received crc frame

wire [63:0] rcvd_data_frm;   // register to hold the received data

wire rx_crc_intl;            // indicates the initialization of the
                             // receiver crc generator

wire rx_crc_enable;          // enable signal for the receiver crc
                             // generator

wire de_stuff;               // indicates that de-stuff is taking place

wire rcvd_crc_flg;           // indicates the reception of the crc
                             // frame

wire [2:0] one_count;        // counter to count the consecutive one's
                             // in the received message

wire [2:0] zero_count;       // counter to count the consecutive zero's
                             // in the received message

wire acpt_sts;               // indicates the received message is
                             // accepted

registry reg_file (clk, g_rst, data_in, param_ld, mask_param,
code_param, sjw);

tx_buff tx_buffer (clk, g_rst, data_in, tx_buff_ld, tx_success,
frame_gen_intl, tx_buff_busy, tx_buff_1, tx_buff_2, tx_buff_3,
tx_buff_4, tx_buff_5, tx_buff_6, tx_buff_7, tx_buff_8, tx_buff_9,
tx_buff_10, rtr, dlc);

dt_rm_frame_gen data_remote_frm (clk, g_rst, frame_gen_intl,
tx_success, tx_buff_1, tx_buff_2, tx_buff_3, tx_buff_4, tx_buff_5,
tx_buff_6, tx_buff_7, tx_buff_8, tx_buff_9, tx_buff_10, rtr, dlc,
tx_crc_frm_cmp, tx_crc_frm, par_ser_data, par_ser_intl, dt_rm_frm,
bit_stf_intl, dt_rm_frm_len);
```

```
par_ser_conv parser (clk, g_rst, par_ser_intl, tx_success,
par_ser_data, dlc, rtr, tx_serial_out, tx_crc_intl, tx_crc_enable,
tx_crc_frm_cmp);

can_crc tx_crc (clk, g_rst, tx_serial_out, tx_crc_enable, tx_crc_intl,
tx_success, rx_success, tx_crc_frm);

bit_stuff bt_stf (clk, g_rst, dt_rm_frm, bit_stf_intl, dt_rm_frm_len,
tx_success, err_state, arbtr_sts, abort_dt_rm_tx, re_tran, dt_rm_out,
dt_rm_frm_tx, arbtr_fld, dt_rm_eof_tx_cmp, txed_lst_bit_ifs, ack_slt,
ifs_flg_tx);

ovld_err_frm_gen ovld_err (clk, g_rst, init_err_st, tx_bit, serial_in,
arbtr_sts, arbtr_fld, bt_err, ack_err, stf_err, frm_err, crc_err,
over_ld, sampled_bit, rx_eof_success, tx_eof_success, rx_success,
tx_success, err_ovld_out, err_state, ovld_frm_tx, ovld_flg_tx, txmtr,
act_err_frm_tx, psv_err_frm_tx, act_err_flg_tx, psv_err_flg_tx,
cons_zero_flg, ovld_err_ifs_tx, ovld_err_tx_cmp, bus_off_sts);

slzd_frm_tx slzr (clk, g_rst, dt_rm_frm_tx, act_err_frm_tx,
psv_err_frm_tx, ovld_frm_tx, dt_rm_out, err_ovld_out, send_ack,
bus_off_sts, arbtr_sts, can_bus_out, abort_dt_rm_tx, tx_bit);

msg_processor msg_prsr (clk, g_rst, stf_err, bt_err, crc_err, ack_err,
frm_err, rcvd_eof_flg, rcvd_lst_bit_ifs, dt_rm_eof_tx_cmp,
txed_lst_bit_ifs, ovld_err_tx_cmp, rcvd_data_len, rcvd_bt_cnt,
de_stuff, act_err_frm_tx, psv_err_frm_tx, tx_buff_busy, arbtr_sts,
msg_due_tx, serial_in, rx_buff_0_wrtn, rx_buff_1_wrtn, bt_ack_err_pre,
stf_frm_crc_err_pre, rx_eof_success, rx_success, tx_eof_success,
tx_success,re_tran, send_ack, txmtr, over_ld);

arbtr_ctrl arbtr_sts_ctrl (osc_clk, g_rst, arbtr_fld, rcvd_lst_bit_ifs,
txed_lst_bit_ifs, ovld_err_tx_cmp, tx_buff_busy, bt_ack_err_pre,
bit_destf_intl, dt_rm_frm_tx, sampling_pt, can_bus_out, can_bus_in,
act_err_frm_tx, psv_err_frm_tx, arbtr_sts, msg_due_tx);

synchronizer synchro (osc_clk, g_rst, can_bus_in, rcvd_lst_bit_ifs,
sjw, ovld_err_tx_cmp, txed_lst_bit_ifs, arbtr_sts, clk, bit_destf_intl,
sampling_pt, sampled_bit);

bit_destuff destuff (clk, g_rst, arbtr_sts, bit_destf_intl,
sampled_bit, tx_success, rx_success, act_err_frm_tx, psv_err_frm_tx,
ovld_frm_tx, serial_in, rx_crc_frm, rcvd_bt_cnt, de_stuff, one_count,
zero_count, rcvd_eof_flg, rcvd_msg_id, rcvd_rtr, rcvd_dlc, rcvd_crc,
rx_crc_intl, rx_crc_enable, rcvd_data_len, rcvd_data_frm,
rcvd_lst_bit_ifs, rcvd_crc_flg, rcvd_lst_bit_eof);

can_crc rx_crc (clk, g_rst, serial_in, rx_crc_enable, rx_crc_intl,
tx_success, rx_success, rx_crc_frm);

bit_stuff_monitor stf_error (clk, g_rst, serial_in, arbtr_fld,
one_count, zero_count, stf_err);
```

```
crc_checker crc_err_chk (clk, g_rst, rx_success, act_err_frm_tx,
psv_err_frm_tx, rx_crc_frm, rcvd_crc_flg, rcvd_crc, crc_err);

form_checker frm_error (clk, g_rst, rx_success, act_err_frm_tx,
psv_err_frm_tx, rcvd_bt_cnt, rcvd_data_len, serial_in, frm_err);

bit_monitor bit_err_chk (clk, g_rst, can_bus_out, sampled_bit,
dt_rm_frm_tx, act_err_flg_tx, psv_err_flg_tx, ovld_flg_tx,
cons_zero_flg, ovld_err_ifs_tx, tx_success, arbtr_fld, ack_slt,
ifs_flg_tx, arbtr_sts, bt_err);

ack_checker ack_err_chk (clk, g_rst, ack_slt, sampled_bit, arbtr_sts,
act_err_frm_tx, psv_err_frm_tx, tx_success, ack_err);

accp_checker id_check (clk, g_rst, arbtr_sts, mask_param, code_param,
rcvd_lst_bit_eof, stf_frm_crc_err_pre, bt_ack_err_pre, rcvd_msg_id,
acpt_sts);

rx_buff rx_buffer (clk, g_rst, acpt_sts, rd_en, rcvd_msg_id, rcvd_rtr,
rcvd_dlc, rcvd_data_frm, buff_rdy, rx_buff_0_wrtn, rx_buff_1_wrtn,
data_out);

endmodule
```

APPENDIX B

REGISTRY

```
// module to store the register data
module registry (clk, g_rst, data_in, param_ld, mask_param, code_param,
sjw);

input clk;                    // sys_clk
input g_rst;                  // global rst active low
input [7:0] data_in;
input param_ld;

output [10:0] mask_param;
output [10:0] code_param;
output [1:0] sjw;

reg [7:0] data_reg;
reg [7:0] data_reg_in;
reg [10:0] mask_param;
reg [10:0] code_param;
reg [1:0] sjw;
reg [2:0] state;
reg param_ld_en;
reg param_ld_in;

parameter [2:0] idle = 3'd0,
                               prmtr_0 = 3'd1,
                               prmtr_1 = 3'd2,
                               prmtr_2 = 3'd3,
                               prmtr_ld_comp = 3'd4;


// Block to enable parameter load
always @ (posedge clk or posedge g_rst)
   begin
       if (g_rst)
          param_ld_in <= 1'b0;
       else if (param_ld)
          param_ld_in <= 1'b1;
       else param_ld_in <= 1'b0;
    end

// Block to enable parameter load
always @ (posedge clk or posedge g_rst)
   begin
       if (g_rst)
          param_ld_en <= 1'b0;
       else if (param_ld_in)
          param_ld_en <= 1'b1;
       else param_ld_en <= 1'b0;
    end
```

```
//Blocks to synchronize data_in
always @ (posedge clk or posedge g_rst)
   begin
      if(g_rst)
         data_reg_in <= 8'd0;
      else data_reg_in <= data_in;
    end

always @ (posedge clk or posedge g_rst)
   begin
      if(g_rst)
         data_reg <= 8'd0;
      else data_reg <= data_reg_in;
    end

//Block to determine output
always @ (posedge clk or posedge g_rst)
      begin
       if(g_rst)
        begin
           mask_param <= 11'd0;
           code_param <= 11'd0;
           sjw <= 2'd0;
         end
        else begin
            case(state)
            idle:begin
               mask_param <= 11'd0;
               code_param <= 11'd0;
               sjw <= 2'd0;
            end
            prmtr_0:begin
              mask_param <= {3'd0, data_reg[7:0]};
             end
            prmtr_1:begin
              mask_param <= {data_reg[2:0], mask_param[7:0]};
              code_param <= {6'd0, data_reg[7:3]};
             end
            prmtr_2:begin
              code_param <= {data_reg[5:0], code_param[4:0]};
              sjw <= data_reg[7:6];
             end
            prmtr_ld_comp:begin
              mask_param <= mask_param;
              code_param <= code_param;
              sjw <= sjw;
             end
            default: begin
               mask_param <= 11'd0;
               code_param <= 11'd0;
               sjw <= 2'd0;
        end
     endcase
      end
```

```
end

//Block to determine nxt_state
always @ (posedge clk or posedge g_rst)
   begin
     if (g_rst)
          state <= idle;
     else begin
     case (state)
         idle:begin
            if (param_ld_en)
               begin
                  state <= prmtr_0;
               end
            else begin
                  state <= idle;
               end
           end
         prmtr_0:begin
              state <= prmtr_1;
           end
         prmtr_1:begin
              state <= prmtr_2;
           end
         prmtr_2:begin
              state <= prmtr_ld_comp;
              end
                  prmtr_ld_comp:begin
                                if (param_ld_en)
                  state <= prmtr_0;
                else begin
                  state <= prmtr_ld_comp;
                 end
            end
         default: begin
                state <= idle;
            end
         endcase
         end
  end // end of always block

endmodule
```

APPENDIX C

TRANSMITTER BUFFER

```
// module to load transmitter buffers
module tx_buff (clk, g_rst, data_in, tx_buff_ld, tx_success,
frame_gen_intl, tx_buff_busy, tx_buff_1, tx_buff_2, tx_buff_3,
tx_buff_4, tx_buff_5, tx_buff_6, tx_buff_7, tx_buff_8,  tx_buff_9,
tx_buff_10, rtr, dlc);

input clk;                    // sys_clk
input g_rst;                  // global rst active low
input [7:0] data_in;          // data_in from host ctrl
input tx_buff_ld;             // load tx_buff
input tx_success;             // transmission succesfully completed

output frame_gen_intl;         // buffer status full =1 empty =0
output tx_buff_busy;           // status signal to indicate that buff
                               // is loaded
output [7:0] tx_buff_1;        // tranmit buffers
output [7:0] tx_buff_2;        // tranmit buffers
output [7:0] tx_buff_3;        // tranmit buffers
output [7:0] tx_buff_4;        // tranmit buffers
output [7:0] tx_buff_5;        // tranmit buffers
output [7:0] tx_buff_6;        // tranmit buffers
output [7:0] tx_buff_7;        // tranmit buffers
output [7:0] tx_buff_8;        // tranmit buffers
output [7:0] tx_buff_9;        // tranmit buffers
output [7:0] tx_buff_10;       // tranmit buffers
output rtr;                    // remote transfer req bit
output [3:0] dlc;              // data_in length code

reg frame_gen_intl;
reg tx_buff_busy;           // status signal to indiacte buffer is full
reg [7:0] tx_buff_1;        // tranmit buffers
reg [7:0] tx_buff_2;        // tranmit buffers
reg [7:0] tx_buff_3;        // tranmit buffers
reg [7:0] tx_buff_4;        // tranmit buffers
reg [7:0] tx_buff_5;        // tranmit buffers
reg [7:0] tx_buff_6;        // tranmit buffers
reg [7:0] tx_buff_7;        // tranmit buffers
reg [7:0] tx_buff_8;        // tranmit buffers
reg [7:0] tx_buff_9;        // tranmit buffers
reg [7:0] tx_buff_10;       // tranmit buffers
reg rtr;                    // remote transfer req bit
reg [3:0] dlc;              // data_in length code

parameter [3:0] idle = 4'd0,
                buf0 = 4'd1,
                buf1 = 4'd2,
                buf2 = 4'd3,
                buf3 = 4'd4,
                buf4 = 4'd5,
```

```
                buf5 = 4'd6,
                buf6 = 4'd7,
                buf7 = 4'd8,
                buf8 = 4'd9,
                buf9 = 4'd10,
                buf_ld_comp = 4'd12;

reg [3:0] state;
reg tx_buff_ld_en;
reg [7:0] data_reg;
reg [7:0] data_reg_in;
reg tx_buff_ld_en_in;

always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         tx_buff_ld_en_in <= 1'b0;
      else if (tx_buff_ld)
         tx_buff_ld_en_in <= 1'b1;
      else tx_buff_ld_en_in <= 1'b0;
    end
//block to enable transmitter buffer loading
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         tx_buff_ld_en <= 1'b0;
      else if (tx_buff_ld_en_in)
         tx_buff_ld_en <= 1'b1;
      else tx_buff_ld_en <= 1'b0;
    end
//blocks to synchronize data_in
always @ (posedge clk or posedge g_rst)
   begin
      if(g_rst)
         data_reg_in <= 8'd0;
      else data_reg_in <= data_in;
    end

always @ (posedge clk or posedge g_rst)
   begin
      if(g_rst)
         data_reg <= 8'd0;
      else data_reg <= data_reg_in;
    end

// block to determine output
always @ (posedge clk or posedge g_rst)
   begin
     if (g_rst)
        begin
          frame_gen_intl <= 1'b0;
          tx_buff_busy <= 1'b0;
          tx_buff_1 <= 8'd0;
          tx_buff_2 <= 8'd0;
```

```
      tx_buff_3 <= 8'd0;
      tx_buff_4 <= 8'd0;
      tx_buff_5 <= 8'd0;
      tx_buff_6 <= 8'd0;
      tx_buff_7 <= 8'd0;
      tx_buff_8 <= 8'd0;
      tx_buff_9 <= 8'd0;
      tx_buff_10 <= 8'd0;
      rtr <= 1'b0;
      dlc <= 4'd0;
    end
else begin
  case (state)
    idle:begin
             frame_gen_intl <= 1'b0;
             tx_buff_busy <= 1'b0;
             tx_buff_1 <= 8'd0;
             tx_buff_2 <= 8'd0;
             tx_buff_3 <= 8'd0;
             tx_buff_4 <= 8'd0;
             tx_buff_5 <= 8'd0;
             tx_buff_6 <= 8'd0;
             tx_buff_7 <= 8'd0;
             tx_buff_8 <= 8'd0;
             tx_buff_9 <= 8'd0;
             tx_buff_10 <= 8'd0;
             rtr <= 1'b0;
             dlc <= 4'd0;
      end
     buf0: begin
             tx_buff_1 <= data_reg;
           end
     buf1: begin
              tx_buff_1 <= tx_buff_1;
              tx_buff_2 <= data_reg;
              rtr <= data_reg[4];
              dlc <= data_reg[3:0];
       end
     buf2: begin
             tx_buff_2 <= tx_buff_2;
             tx_buff_3 <= data_reg;
      end
     buf3: begin
             tx_buff_3 <= tx_buff_3;
             tx_buff_4 <= data_reg;
      end
     buf4: begin
             tx_buff_4 <= tx_buff_4;
             tx_buff_5 <= data_reg;
      end
     buf5: begin
             tx_buff_5 <= tx_buff_5;
             tx_buff_6 <= data_reg;
      end
```

```
          buf6: begin
                  tx_buff_6 <= tx_buff_6;
                  tx_buff_7 <= data_reg;
           end
          buf7: begin
                  tx_buff_7 <= tx_buff_7;
                  tx_buff_8 <= data_reg;
           end
          buf8: begin
                  tx_buff_8 <= tx_buff_8;
                  tx_buff_9 <= data_reg;
           end
          buf9: begin
                  tx_buff_9 <= tx_buff_9;
                  tx_buff_10 <= data_reg;
                  frame_gen_intl <= 1'b1;
           end
          buf_ld_comp: begin
                  tx_buff_10 <= tx_buff_10;
                  tx_buff_busy <= 1'b1;
                  frame_gen_intl <= 1'b0;
           end
          default: begin
               tx_buff_busy <= 1'b0;
               frame_gen_intl <= 1'b0;
               tx_buff_1 <= 8'd0;
               tx_buff_2 <= 8'd0;
               tx_buff_3 <= 8'd0;
               tx_buff_4 <= 8'd0;
               tx_buff_5 <= 8'd0;
               tx_buff_6 <= 8'd0;
               tx_buff_7 <= 8'd0;
               tx_buff_8 <= 8'd0;
               tx_buff_9 <= 8'd0;
               tx_buff_10 <= 8'd0;
               rtr <= 1'b0;
               dlc <= 4'd0;
           end
      endcase
   end
end                     // end of always block

//block to determine next state
always @ (posedge clk or posedge g_rst)
   begin
     if (g_rst)
          state <= idle;
     else begin
     case (state)
         idle:begin
            if (tx_buff_ld_en)
               begin
                  state <= buf0;
               end
```

```
            else begin
                  state <= idle;
                end
             end
            buf0: begin
                   state <= buf1;
                end
            buf1: begin
                     state <= buf2;
              end
            buf2: begin
                     state <= buf3;
              end
            buf3: begin
                    state <= buf4;
             end
            buf4: begin
                    state <= buf5;
             end
            buf5: begin
                    state <= buf6;
             end
            buf6: begin
                    state <= buf7;
             end
            buf7: begin
                    state <= buf8;
             end
            buf8: begin
                    state <= buf9;
             end
            buf9: begin
                    state <= buf_ld_comp;
             end
            buf_ld_comp: begin
                if (tx_success)
                   state <= idle;
                else begin
                   state <= buf_ld_comp;
                 end
             end
            default: begin
                 state <= idle;
             end
     endcase
   end
 end
endmodule
```

APPENDIX D

DATA REMOTE FRAME GENERATION

```
// module to generate data or remote frame
module dt_rm_frame_gen (clk, g_rst, frame_gen_intl, tx_success,
tx_buff_1, tx_buff_2, tx_buff_3, tx_buff_4, tx_buff_5, tx_buff_6,
tx_buff_7, tx_buff_8, tx_buff_9, tx_buff_10, rtr, dlc, tx_crc_frm_cmp,
tx_crc_frm, par_ser_data, par_ser_intl, dt_rm_frm, bit_stf_intl,
dt_rm_frm_len);

//inputs
input clk;                    // sys_clk
input g_rst;                  // global rst active low
input tx_success;             // transmission succesfully completed
input frame_gen_intl;         // buffer status full =1 empty =0
input rtr;                    // remote transfer req bit = 1 if remote
                              // transfer req
input [3:0] dlc;              // data length code
input [7:0] tx_buff_1;        // tranmit buffers
input [7:0] tx_buff_2;        // tranmit buffers
input [7:0] tx_buff_3;        // tranmit buffers
input [7:0] tx_buff_4;        // tranmit buffers
input [7:0] tx_buff_5;        // tranmit buffers
input [7:0] tx_buff_6;        // tranmit buffers
input [7:0] tx_buff_7;        // tranmit buffers
input [7:0] tx_buff_8;        // tranmit buffers
input [7:0] tx_buff_9;        // tranmit buffers
input [7:0] tx_buff_10;       // tranmit buffers
input tx_crc_frm_cmp;         // flag to indicate completion of crc
                              // frame
input [14:0] tx_crc_frm;      // crc_frame

output [82:0] par_ser_data;   // par_ser_data input to par to series
converter module
output par_ser_intl;          // initialize par to series conv
output [97:0] dt_rm_frm;      // output to bit stuffing module
output bit_stf_intl;          // output to initialise bit stuffing
output [7:0] dt_rm_frm_len;   // frame length output to bit stuffing
                              // module

reg [82:0] par_ser_data;      // par_ser_data input to par to series
converter module
reg par_ser_intl;             // initialize par to series conv
reg [97:0] dt_rm_frm;         // to bit stuffing module
reg bit_stf_intl;             // to initialise bit stuffing module
reg [7:0] dt_rm_frm_len;

parameter [2:0] idle = 3'd0,
                dt_frm = 3'd1,
                rmt_frm = 3'd2,
                par_ser = 3'd3,
                app_msg = 3'd4,
```

```
               frm_cmp = 3'd5;

reg [2:0] state;

// block to determne next state
always @ (posedge clk or posedge g_rst)
   begin
     if (g_rst)
          state <= idle;
     else begin
        case (state)
         idle:begin
            if (frame_gen_intl)
               begin
                  if (~rtr)
                     state <= dt_frm;
                  else
                     state <= rmt_frm;
               end
            else begin
               state <= idle;
             end
           end
         dt_frm:begin
             state <= par_ser;
           end
         rmt_frm:begin
                state <= par_ser;
           end
         par_ser:begin
                if (tx_crc_frm_cmp)
                   begin
                      state <= app_msg;
                   end
                else begin
                   state <= par_ser;
                 end
            end
         app_msg:begin
            state <= frm_cmp;
           end
         frm_cmp:begin
          if (tx_success)
             begin
               state <= idle;
             end
          else begin
              state <= frm_cmp;
             end
           end
     default: begin
         state <= idle;
       end
    endcase
```

```
  end
end

// block to determne output
always @ (posedge clk or posedge g_rst)
   begin
     if (g_rst)
       begin
          par_ser_intl  <= 1'b0;
          par_ser_data  <= 83'd0;
          dt_rm_frm  <= 98'd0;
          dt_rm_frm_len  <= 8'd0;
          bit_stf_intl  <= 1'b0;
        end
     else begin
       case (state)
         idle:begin
               par_ser_intl  <= 1'b0;
               par_ser_data  <= 83'd0;
               dt_rm_frm  <= 98'd0;
               dt_rm_frm_len  <= 8'd0;
               bit_stf_intl  <= 1'b0;
           end
         dt_frm:begin
             case(dlc)
                4'b1000: begin
                       par_ser_data  <= {1'b0, tx_buff_1,
                       tx_buff_2[7:4], 2'd0, tx_buff_2[3:0], tx_buff_3,
                       tx_buff_4, tx_buff_5, tx_buff_6, tx_buff_7,
                       tx_buff_8, tx_buff_9, tx_buff_10};
                       par_ser_intl  <= 1'b1;
                 end
                4'b0111: begin
                       par_ser_data  <= {1'b0, tx_buff_1,
                       tx_buff_2[7:4], 2'd0, tx_buff_2[3:0],
                       tx_buff_3, tx_buff_4, tx_buff_5, tx_buff_6,
                       tx_buff_7, tx_buff_8, tx_buff_9, 8'd0 };
                       par_ser_intl  <= 1'b1;
                 end
                4'b0110: begin
                       par_ser_data  <= {1'b0, tx_buff_1,
                       tx_buff_2[7:4], 2'd0, tx_buff_2[3:0],
                       tx_buff_3, tx_buff_4, tx_buff_5, tx_buff_6,
                       tx_buff_7, tx_buff_8, 16'd0};
                       par_ser_intl  <= 1'b1;
                 end
                4'b0101: begin
                       par_ser_data  <= {1'b0, tx_buff_1,
                       tx_buff_2[7:4], 2'd0, tx_buff_2[3:0],
                       tx_buff_3, tx_buff_4, tx_buff_5, tx_buff_6,
                       tx_buff_7, 24'd0};
                       par_ser_intl  <= 1'b1;
                 end
                4'b0100: begin
```

```
                par_ser_data  <= {1'b0, tx_buff_1,
                tx_buff_2[7:4], 2'd0, tx_buff_2[3:0],
                tx_buff_3, tx_buff_4, tx_buff_5, tx_buff_6,
                32'd0};
                par_ser_intl  <= 1'b1;
             end
            4'b0011: begin
                par_ser_data  <= {1'b0, tx_buff_1,
                tx_buff_2[7:4], 2'd0, tx_buff_2[3:0],
                tx_buff_3, tx_buff_4, tx_buff_5, 40'd0};
                par_ser_intl  <= 1'b1;
             end
            4'b0010: begin
                par_ser_data  <= {1'b0, tx_buff_1,
                tx_buff_2[7:4], 2'd0, tx_buff_2[3:0],
                tx_buff_3, tx_buff_4, 48'd0};
                par_ser_intl  <= 1'b1;
             end
            4'b0001: begin
                par_ser_data  <= {1'b0, tx_buff_1,
                tx_buff_2[7:4], 2'd0, tx_buff_2[3:0],
                tx_buff_3, 56'd0};
                par_ser_intl  <= 1'b1;
             end
            4'b0000: begin
                par_ser_data  <= {1'b0, tx_buff_1,
                tx_buff_2[7:4], 2'd0, tx_buff_2[3:0], 64'd0};
                par_ser_intl  <= 1'b1;
             end
            default: begin
                par_ser_data  <= 83'd0;
                par_ser_intl  <= 1'b0;
             end
        endcase
      end
    rmt_frm:begin
          par_ser_data  <= {1'b0, tx_buff_1, tx_buff_2[7:4], 2'd0,
          tx_buff_2[3:0], 64'd0};
          par_ser_intl  <= 1'b1;
      end
    par_ser:begin
             par_ser_data  <= par_ser_data;
             par_ser_intl  <= 1'b0;
           end
    app_msg:begin
       bit_stf_intl  <= 1'b1;
      case(rtr)
       1'b0: begin
         case(dlc)
           4'b1000: begin
               dt_rm_frm  <= {1'b0, tx_buff_1, tx_buff_2[7:4],
               2'd0, tx_buff_2[3:0], tx_buff_3, tx_buff_4,
               tx_buff_5, tx_buff_6, tx_buff_7, tx_buff_8,
               tx_buff_9, tx_buff_10, tx_crc_frm};
```

```
    dt_rm_frm_len  <= 8'd98;
 end
4'b0111: begin
    dt_rm_frm  <= {1'b0, tx_buff_1, tx_buff_2[7:4],
    2'd0, tx_buff_2[3:0],tx_buff_3, tx_buff_4,
    tx_buff_5, tx_buff_6, tx_buff_7, tx_buff_8,
    tx_buff_9, tx_crc_frm, 8'hff};
    dt_rm_frm_len  <= 8'd90;
  end
4'b0110: begin
   dt_rm_frm  <= {1'b0, tx_buff_1, tx_buff_2[7:4],
   2'd0, tx_buff_2[3:0], tx_buff_3, tx_buff_4,
   tx_buff_5, tx_buff_6, tx_buff_7, tx_buff_8,
   tx_crc_frm, 16'hffff};
   dt_rm_frm_len  <= 8'd82;
  end
4'b0101: begin
    dt_rm_frm  <= {1'b0, tx_buff_1, tx_buff_2[7:4],
    2'd0, tx_buff_2[3:0], tx_buff_3, tx_buff_4,
    tx_buff_5, tx_buff_6, tx_buff_7, tx_crc_frm,
    24'hffffff};
    dt_rm_frm_len  <= 8'd74;
  end
4'b0100: begin
    dt_rm_frm  <= {1'b0, tx_buff_1, tx_buff_2[7:4],
    2'd0, tx_buff_2[3:0], tx_buff_3, tx_buff_4,
    tx_buff_5, tx_buff_6, tx_crc_frm, 32'hffffffff};
    dt_rm_frm_len  <= 8'd66;
  end
4'b0011: begin
    dt_rm_frm  <= {1'b0, tx_buff_1, tx_buff_2[7:4],
    2'd0, tx_buff_2[3:0], tx_buff_3, tx_buff_4,
    tx_buff_5, tx_crc_frm, 40'hfffffffff};
    dt_rm_frm_len  <= 8'd58;
 end
4'b0010: begin
    dt_rm_frm  <= {1'b0, tx_buff_1, tx_buff_2[7:4],
    2'd0, tx_buff_2[3:0], tx_buff_3, tx_buff_4,
    tx_crc_frm, 48'hffffffffffff};
    dt_rm_frm_len  <= 8'd50;
 end
4'b0001: begin
    dt_rm_frm  <= {1'b0, tx_buff_1, tx_buff_2[7:4],
    2'd0, tx_buff_2[3:0], tx_buff_3, tx_crc_frm,
    56'hffffffffffffff};
    dt_rm_frm_len  <= 8'd42;
 end
4'b0000: begin
    dt_rm_frm  <= {1'b0, tx_buff_1, tx_buff_2[7:4],
    2'd0, tx_buff_2[3:0], tx_crc_frm,
    64'hffffffffffffffff};
    dt_rm_frm_len  <= 8'd34;
 end
default: begin
```

```
                    dt_rm_frm  <= 98'd0;
                    dt_rm_frm_len  <= 8'd0;
                  end
                endcase
              end
             1'b1: begin
                    dt_rm_frm  <= {1'b0, tx_buff_1, tx_buff_2[7:4],
                    2'd0, tx_buff_2[3:0], tx_crc_frm,
                    64'hffffffffffffffff};
                    dt_rm_frm_len  <= 8'd34;
              end
            default:begin
               dt_rm_frm  <= 98'd0;
               dt_rm_frm_len  <= 8'd0;
              end
            endcase
           end
      frm_cmp:begin
              bit_stf_intl  <= 1'b0;
             end
      default: begin
          par_ser_data  <= 83'd0;
          par_ser_intl  <= 1'b0;
          dt_rm_frm  <= 98'd0;
          dt_rm_frm_len  <= 8'd0;
        end
     endcase
    end
   end
endmodule
```

APPENDIX E

PARALLEL TO SERIES CONVERTER

```
// Module to generate serial output from the data/remote frame
module par_ser_conv (clk, g_rst, par_ser_intl, tx_success,
par_ser_data, dlc, rtr, tx_serial_out, tx_crc_intl, tx_crc_enable,
tx_crc_frm_cmp);
 input clk;
 input g_rst;
 input par_ser_intl;
 input tx_success;
 input [82:0] par_ser_data;
 input [3:0] dlc;
 input rtr;

 output tx_serial_out;
 output tx_crc_intl;
 output tx_crc_enable;
 output tx_crc_frm_cmp;

 reg tx_serial_out;
 reg tx_crc_intl;
 reg tx_crc_enable;
 reg tx_crc_frm_cmp;
  reg [1:0] state;
 reg [82:0] temp_data;
 reg [6:0] count;

 parameter [1:0] idle = 2'd0,
                 load = 2'd1,
                 slz = 2'd2,
                 slz_comp = 2'd3;


// Always block to determine nxt state
always @ (posedge clk or posedge g_rst)
   begin
     if (g_rst)
          state <= idle;
     else begin
       case (state)
           idle:begin
              if (par_ser_intl)
                 begin
                    state <= load;
                end
              else begin
                    state <= idle;
                end
            end
            load:begin
                state <= slz;
```

```
            end
            slz:begin
                if (~rtr)
                begin
                   if (count < (7'd20 + (7'd8 * (7'd8 * dlc[3] + 7'd4 *
                   dlc[2] + 7'd2 * dlc[1] + 7'd1 * dlc[0]))))
                      begin
                          state <= slz;
                      end
                   else begin
                         state <= slz_comp;
                      end
                  end
                else begin
                    if (count < 7'd20)
                       begin
                           state <= slz;
                       end
                    else begin
                          state <= slz_comp;
                       end
                   end
            end
            slz_comp:begin
                if (tx_success)
                   begin
                      state <= idle;
                   end
                else begin
                   state <= slz_comp;
                end
           end
           default: begin
               state <= idle;
           end
         endcase
      end
    end

// block to determine output
always @ (posedge clk or posedge g_rst)
  begin
     if (g_rst)
       begin
         tx_crc_intl <= 1'b1;
         temp_data  <=  83'd0;
         tx_serial_out <= 1'b0;
         tx_crc_enable <= 1'b0;
         tx_crc_frm_cmp <= 1'b0;
         count <= 7'd0;
       end
     else begin
       case (state)
           idle:begin
```

```
  if (par_ser_intl)
     begin
        tx_crc_intl  <= 1'b0;
        temp_data  <=  83'd0;
        tx_serial_out <= 1'b0;
        tx_crc_enable <= 1'b0;
        tx_crc_frm_cmp <= 1'b0;
        count <= 7'd0;
     end
  else begin
        tx_crc_intl <= 1'b1;
        temp_data  <=  83'd0;
        tx_serial_out <= 1'b0;
        tx_crc_enable <= 1'b0;
        tx_crc_frm_cmp <= 1'b0;
        count <= 7'd0;
     end
end
load:begin
    tx_crc_intl  <= 1'b0;
    temp_data <= par_ser_data;
    tx_serial_out <= 1'b0;
    tx_crc_enable <= 1'b1;
    tx_crc_frm_cmp <= 1'b0;
    count <= count + 7'd1;
end
slz:begin
    if (~rtr)
    begin
       if (count < (7'd20 + (7'd8*( 7'd8 * dlc[3] + 7'd4 *
       dlc[2] + 7'd2 * dlc[1] + 7'd1 * dlc[0]))))
          begin
              tx_crc_intl  <= 1'b0;
              tx_serial_out <= temp_data[82];
              temp_data <= temp_data << 1;
              tx_crc_enable <= 1'b1;
              count <= count + 7'd1;
          end
       else begin
             tx_crc_intl  <= 1'b0;
             temp_data <= 83'd0;
             tx_serial_out <= 1'b0;
             tx_crc_enable <= 1'b0;
             tx_crc_frm_cmp <= 1'b1;
             count <= 7'd0;
          end
      end
    else begin
        if (count < 7'd20)
           begin
              tx_crc_intl  <= 1'b0;
              tx_serial_out <= temp_data[82];
              temp_data <= temp_data << 1;
              tx_crc_enable <= 1'b1;
```

```
                        count <= count + 7'd1;
                    end
                 else begin
                        tx_crc_intl  <= 1'b0;
                        temp_data <= 83'd0;
                        tx_serial_out <= 1'b0;
                        tx_crc_enable <= 1'b0;
                        tx_crc_frm_cmp <= 1'b1;
                        count <= 7'd0;
                    end
                end
          end
          slz_comp:begin
                tx_crc_frm_cmp <= 1'b0;
                tx_crc_intl  <= 1'b0;
                temp_data <= 83'd0;
                tx_serial_out <= 1'b0;
                tx_crc_enable <= 1'b0;
             end
          default: begin
              tx_crc_intl <= 1'b1;
              temp_data  <=  83'd0;
              tx_serial_out <= 1'b0;
              tx_crc_enable <= 1'b0;
              tx_crc_frm_cmp <= 1'b0;
          end
        endcase
    end
  end
endmodule
```

APPENDIX F

CRC GENERATOR

```
// module to generate crc farme
module can_crc (clk, g_rst, data, enable, initialize, tx_success,
rx_success, crc_frm);

input clk;
input g_rst;
input data;
input enable;
input initialize;
input tx_success;
input rx_success;

output [14:0] crc_frm;

reg [14:0] crc_frm;

wire          crc_next;
wire   [14:0] crc_tmp;


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


always @ (posedge clk or posedge g_rst)
begin
  if(g_rst)
    crc_frm <= 15'h0;
  else if (tx_success || rx_success)
     crc_frm <= 15'h0;
  else if (initialize)
    crc_frm <= 15'h0;
  else if (enable)
    begin
      if (crc_next)
        crc_frm <= crc_tmp ^ 15'h4599;
      else
     begin
        crc_frm <= crc_tmp;
     end
    end
end

endmodule
```

APPENDIX G

BIT STUFFING

```
// Module to produce bit stuffed serial output
module bit_stuff (clk, g_rst, dt_rm_frm, bit_stf_intl, dt_rm_frm_len,
tx_success, err_state, arbtr_sts, abort_dt_rm_tx, re_tran, dt_rm_out,
dt_rm_frm_tx, arbtr_fld, dt_rm_eof_tx_cmp , txed_lst_bit_ifs, ack_slt,
ifs_flg_tx);

input clk;
input g_rst;
input arbtr_sts;                // signal to indicate if the node is a
                                // transmitter or receiver
input abort_dt_rm_tx;
input re_tran;                  // request for re transmission if node
                                // loses arbitration
input [1:0] err_state;
input tx_success;
input [97:0] dt_rm_frm;         // input frame to bit stuffing module
input bit_stf_intl;             // initialize signal to start bit
                                // stuffing
input [7:0] dt_rm_frm_len;      // gives the length of the data frame

output dt_rm_out;               // serial output from the data/remote
                                // frame generator module
output dt_rm_frm_tx;            // indicates the transmission of
                                // data/remote frame
output arbtr_fld;               // indicates the transmission of the
                                // arbtr fld
output ack_slt;                 // indicates the transmission of
                                //  acknowledgement bit
output dt_rm_eof_tx_cmp ;       // indicates the end of transmission
                                // of the data/remote frame
output txed_lst_bit_ifs;        // indicates the transmission of the
                                // last bit of ifs
output ifs_flg_tx;

reg dt_rm_out;
reg dt_rm_frm_tx;
reg arbtr_fld;
reg ack_slt;
reg dt_rm_eof_tx_cmp ;
reg txed_lst_bit_ifs;
reg ifs_flg_tx;
reg [2:0] eof_bit_cnt;
reg [3:0] ifs_bit_cnt;
reg [97:0] msg;
reg [2:0] one_count;
reg [2:0] zero_count;
reg [6:0] bit_count;
reg [3:0] state;
```

```
parameter [3:0] idle = 4'd0,
                load = 4'd1,
                zero_4 = 4'd2,
                zero_5 = 4'd3,
                one_stf = 4'd4,
                ones_4 = 4'd5,
                ones_5 = 4'd6,
                zero_stf = 4'd7,
                crc_delim = 4'd8,
                ack_slot = 4'd9,
                ack_delim = 4'd10,
                eof = 4'd11,
                ifs = 4'd12,
                dt_rm_cmp = 4'd13;

// block to determine the next state
always @ (posedge clk or posedge g_rst)
  begin
     if (g_rst)
     begin
         state <= idle;
      end
    else if (abort_dt_rm_tx)
      begin
         state <= idle;
       end
    else if (arbtr_sts)
      begin
        case (state)
         idle:begin
            if (bit_stf_intl || re_tran)
               begin
                 state <= load;
                end
            else begin
                 state <= idle;
             end
           end
         load:begin
                state <= zero_4;
               end
         zero_4:begin
                   if (bit_count == (dt_rm_frm_len - 7'd1))
                      begin
                         state <= crc_delim;
                      end
                   else if (msg == dt_rm_frm)
                      begin
                          if (!msg[96])
                            state <= zero_4;
                          else state <= ones_4;
                      end
                   else if (!msg[97])
                      begin
```

```
                if (zero_count < 3'd3)
                  begin
                     state <= zero_4;
                  end
                else
                  begin
                     state <= zero_5;
                   end
              end
            else begin
                state <= ones_4;
            end
   end
zero_5:begin
          if (bit_count == (dt_rm_frm_len - 7'd1))
            begin
              state <= crc_delim;
            end
          else begin
            state <= one_stf;
           end
    end
one_stf:begin
          if (bit_count == (dt_rm_frm_len - 7'd1))
             begin
                state <= crc_delim;
             end
          else if (!msg[97])
               begin
                  state <= zero_4;
               end
          else begin
                  state <= ones_4;
             end
     end
ones_4:begin
          if (bit_count == (dt_rm_frm_len - 7'd1))
              begin
                 state <= crc_delim;
              end
          else if (msg[97])
               begin
                 if (one_count < 3'd3)
                    begin
                       state <= ones_4;
                     end
                 else begin
                    state <= ones_5;
                   end
               end
            else begin
               state <= zero_4;
             end
        end
```

```
ones_5:begin
           if (bit_count == (dt_rm_frm_len - 7'd1))
              begin
                 state <= crc_delim;
              end
           else begin
              state <= zero_stf;
           end
    end
zero_stf:begin
         if (bit_count == (dt_rm_frm_len - 7'd1))
             begin
                state <= crc_delim;
             end
          else if (~msg[97])
              begin
                 state <= zero_4;
             end
          else begin
             state <= ones_4;
         end
      end
 crc_delim:begin
         state <= ack_slot;
      end
 ack_slot:begin
         state <= ack_delim;
        end
 ack_delim:begin
         state <= eof;
        end
 eof:begin
        if(eof_bit_cnt <= 3'd5)
           begin
               state <= eof;
           end
         else begin
               state <= ifs;
           end
       end
 ifs:begin
            if(err_state == 2'b0)
                begin
                   if (ifs_bit_cnt < 4'd2)
                      begin
                        state <= ifs;
                      end
                  else begin
                        state <= dt_rm_cmp;
                      end
                end
             else if(err_state == 2'b01)
                 begin
                    if (ifs_bit_cnt < 4'd10)
```

```
                                begin
                                   state <= ifs;
                                 end
                              else begin
                                   state <= dt_rm_cmp;
                                   end
                           end
                    end
               dt_rm_cmp:begin
                        if(tx_success)
                           begin
                             state <= idle;
                            end
                        else begin
                           state <= dt_rm_cmp;
                         end
                      end
                default:begin
                   state <= idle;
                 end
            endcase
          end
     else state <= idle;
  end

// block to determine the output
always @ (posedge clk or posedge g_rst)
  begin
     if (g_rst)
     begin
         dt_rm_out <= 1'b1;
         ack_slt <= 1'b0;
         msg <= 98'd0;
         bit_count <= 7'd0;
         dt_rm_frm_tx <= 1'b0;
         dt_rm_eof_tx_cmp <= 1'b0;
         txed_lst_bit_ifs <= 1'b0;
         one_count <= 3'd0;
         zero_count <= 3'd0;
         eof_bit_cnt <= 3'd0;
         ifs_bit_cnt <= 4'd0;
         ifs_flg_tx <= 1'b0;
       end
     else begin
      case (state)
           idle:begin
                   dt_rm_out <= 1'b1;
                   ack_slt <= 1'b0;
                   msg <= 98'd0;
                   bit_count <= 7'd0;
                   dt_rm_frm_tx <= 1'b0;
                   dt_rm_eof_tx_cmp <= 1'b0;
                   txed_lst_bit_ifs <= 1'b0;
                   one_count <= 3'd0;
```

```
        zero_count <= 3'd0;
        eof_bit_cnt <= 3'd0;
        ifs_bit_cnt <= 4'd0;
        ifs_flg_tx <= 1'b0;
    end
load:begin
       dt_rm_frm_tx <= 1'b1;
       msg <= {dt_rm_frm[96:0], 1'b1};
      end
zero_4:begin
          dt_rm_out <= 1'b0;
          msg <= msg << 1;
          dt_rm_frm_tx <= 1'b1;
          bit_count <= bit_count + 7'd1;
          zero_count <= zero_count + 3'd1;
          one_count <= 3'd0;
   end
zero_5:begin
          dt_rm_out <= 1'b0;
          msg  <= msg;
          bit_count  <= bit_count + 7'd1;
          zero_count <= zero_count + 3'd1;
          one_count <= 3'd0;
    end
one_stf:begin
          dt_rm_out <= 1'b1;
          msg <= msg << 1;
          one_count <= one_count + 3'd1;
          zero_count <= 3'd0;
     end
ones_4:begin
          dt_rm_out <= 1'b1;
          msg <= msg << 1;
          bit_count  <= bit_count + 7'd1;
          one_count <= one_count + 3'd1;
          zero_count <= 3'd0;
        end
ones_5:begin
           dt_rm_out <= 1'b1;
           msg  <= msg;
           bit_count  <= bit_count + 7'd1;
           one_count <= one_count + 3'd1;
           zero_count <= 3'd0;
    end
zero_stf:begin
          dt_rm_out <= 1'b0;
          msg <=  msg << 1;
          zero_count <= zero_count + 3'd1;
          one_count <= 3'd0;
      end
 crc_delim:begin
          dt_rm_out <= 1'b1;
          bit_count  <= bit_count + 7'd1;
          one_count <= 3'd0;
```

```
          zero_count <= 3'd0;
      end
ack_slot:begin
          ack_slt  <= 1'b1;
          dt_rm_out <= 1'b1;
          bit_count  <= bit_count + 7'd1;
        end
ack_delim:begin
          ack_slt  <= 1'b0;
          dt_rm_out <= 1'b1;
          bit_count  <= bit_count + 7'd1;
        end
eof:begin
         if(eof_bit_cnt <= 3'd5)
           begin
               dt_rm_out <= 1'b1;
               bit_count  <= bit_count + 7'd1;
               eof_bit_cnt <= eof_bit_cnt + 3'd1;
           end
         else begin
               ack_slt  <= 1'b0;
               dt_rm_out <= 1'b1;
               bit_count  <= bit_count + 7'd1;
               eof_bit_cnt <= 3'd0;
               ifs_flg_tx <= 1'b1;
           end
       end
ifs:begin
            if(err_state == 2'b0)
                begin
                   if (ifs_bit_cnt < 4'd2)
                      begin
                        dt_rm_out <= 1'b1;
                        bit_count  <= bit_count + 7'd1;
                        ifs_bit_cnt <= ifs_bit_cnt + 4'd1;
                        ifs_flg_tx <= 1'b1;
                        if (ifs_bit_cnt == 4'd0)
                           dt_rm_eof_tx_cmp  <= 1'b1;
                        else dt_rm_eof_tx_cmp <= 1'b0;
                      end
                  else begin
                        dt_rm_out <= 1'b1;
                        txed_lst_bit_ifs <= 1'b1;
                        bit_count  <= bit_count + 7'd1;
                        ifs_bit_cnt <= 4'd0;
                        ifs_flg_tx <= 1'b0;
                      end
                end
             else if(err_state == 2'b01)
                   begin
                        if (ifs_bit_cnt < 4'd10)
                          begin
                            dt_rm_out <= 1'b1;
                            bit_count  <= bit_count + 7'd1;
```

```
                                        ifs_bit_cnt <=ifs_bit_cnt +4'd1;
                                        ifs_flg_tx <= 1'b1;
                                        if (ifs_bit_cnt == 4'd0)
                                           dt_rm_eof_tx_cmp  <= 1'b1;
                                        else dt_rm_eof_tx_cmp <= 1'b0;
                                       end
                                    else begin
                                        dt_rm_out <= 1'b1;
                                        txed_lst_bit_ifs  <= 1'b1;
                                        bit_count  <= bit_count + 7'd1;
                                        ifs_bit_cnt <= 4'd0;
                                        ifs_flg_tx <= 1'b0;
                                    end
                                end
                            end
                        dt_rm_cmp:begin
                                dt_rm_frm_tx <= 1'b0;
                                txed_lst_bit_ifs <= 1'b0;
                                dt_rm_out <= 1'b1;
                             end
                         default:begin
                            dt_rm_out <= 1'b1;
                          end
                endcase
            end
          end

          // block to indicate transmission of arbtr field
          always @ (posedge clk or posedge g_rst)
             begin
                if (g_rst)
                   arbtr_fld <= 1'b0;
                else if (~arbtr_sts)
                   arbtr_fld <= 1'b0;
                else if ((bit_count > 7'd0) && (bit_count < 7'd13))
                   arbtr_fld <= 1'b1;
                else
                   arbtr_fld <= 1'b0;
              end

endmodule
```

APPENDIX H

OVERLOAD ERROR FRAME GENERATOR

```
// module to generate error or overload signal
module ovld_err_frm_gen (clk, g_rst, init_err_st, tx_bit, serial_in,
arbtr_sts, arbtr_fld, bt_err, ack_err, stf_err, frm_err, crc_err,
over_ld, sampled_bit, rx_eof_success, tx_eof_success, rx_success,
tx_success, err_ovld_out, err_state, ovld_frm_tx, ovld_flg_tx, txmtr,
act_err_frm_tx, psv_err_frm_tx, act_err_flg_tx, psv_err_flg_tx,
cons_zero_flg, ovld_err_ifs_tx, ovld_err_tx_cmp, bus_off_sts);

// inputs
input clk;
input g_rst;
input init_err_st;
input [1:0] tx_bit;
input arbtr_sts;
input arbtr_fld;
input serial_in;
input bt_err;
input ack_err;
input stf_err;
input frm_err;
input crc_err;
input over_ld;
input sampled_bit;
input tx_success;
input rx_success;
input tx_eof_success;
input rx_eof_success;
input txmtr;

//outputs
output act_err_frm_tx;       // indicates active error frm transmission
                             // stays high till end of frm  tr'mission
output psv_err_frm_tx;       // indicates passive error frm tr'mission
                             // stays high till end of frm tr'mission
output ovld_frm_tx;          // indicates overload frm transmission
                             // stays high till end of frm tr'mission
output act_err_flg_tx;       // indicates active error flag tr'mission
                             // stays high till end of flag tr'mission
output psv_err_flg_tx;       // indicates passive error flag tr'mission
                             // stays high till end of flag tr'mission
output ovld_flg_tx;          // indicates overload flag transmission
                             // stays high till end of flag tr'mission
output cons_zero_flg;
output ovld_err_ifs_tx;
output ovld_err_tx_cmp;
output err_ovld_out;
output [1:0] err_state;
output bus_off_sts;
```

```
reg act_err_frm_tx;
reg psv_err_frm_tx;
reg ovld_frm_tx;
reg act_err_flg_tx;
reg psv_err_flg_tx;
reg ovld_flg_tx;
reg cons_zero_flg;
reg ovld_err_ifs_tx;
reg ovld_err_tx_cmp;
reg err_ovld_out;
reg [1:0] err_state;
reg bus_off_sts;
reg init_bit;
reg fst_bit_zero_err;
reg cons_domn_err;
reg tx_excp_1;
reg tx_excp_1_trig;
reg tx_excp_2;
reg [2:0] ovld_err_flg_cnt;
reg [2:0] dlm_cnt;
reg [7:0] cons_zero_cnt;
reg [3:0] ovld_err_ifs_cnt;
reg [8:0] tx_err_cntr;
reg [8:0] rx_err_cntr;
reg [3:0] cons_domn_err_cnt;
reg [2:0] state, nxt_state;
reg init_err_st_in;
reg init_err_st_en;

// state machine to generate error frm and overload frm
parameter [2:0] idle = 3'd0,
         err_flg = 3'd1,
         ovld_flg = 3'd2,
         delm_init = 3'd3,
         ovld_err_dlm = 3'd4,
         ovld_err_ifs = 3'd5,
         ovld_err_comp = 3'd6,
         bus_off = 3'd7;

// Block to synchronise init_err_st
always @ (posedge clk or posedge g_rst)
   begin
       if (g_rst)
          init_err_st_in <= 1'b0;
       else if (init_err_st)
          init_err_st_in <= 1'b1;
       else init_err_st_in <= 1'b0;
      end

always @ (posedge clk or posedge g_rst)
   begin
       if (g_rst)
          init_err_st_en <= 1'b0;
       else if (init_err_st_in)
```

```
          init_err_st_en <= 1'b1;
       else init_err_st_en <= 1'b0;
      end

// block to increment ovld_err_flg_cnt
always @ (posedge clk or posedge g_rst)
begin
if (g_rst)
  ovld_err_flg_cnt <= 3'd0;
else if ((nxt_state == err_flg) || (nxt_state == ovld_flg))
  ovld_err_flg_cnt <= ovld_err_flg_cnt + 3'd1;
else
  ovld_err_flg_cnt <= 3'd0;
end

// block to increment cons_zero_cnt
always @ (posedge clk or posedge g_rst)
begin
if (g_rst)
  cons_zero_cnt <= 8'd0;
else if (nxt_state == delm_init)
  cons_zero_cnt <= cons_zero_cnt + 8'd1;
else
  cons_zero_cnt <= 8'd0;
end

// block to increment dlm_cnt
always @ (posedge clk or posedge g_rst)
begin
if (g_rst)
  dlm_cnt <= 3'd0;
else if (nxt_state == ovld_err_dlm)
  dlm_cnt <= dlm_cnt + 3'd1;
else
  dlm_cnt <= 3'd0;
end

// block to increment ovld_err_ifs_cnt
always @ (posedge clk or posedge g_rst)
begin
if (g_rst)
  ovld_err_ifs_cnt <= 4'd0;
else if (nxt_state ==ovld_err_ifs)
  ovld_err_ifs_cnt <= ovld_err_ifs_cnt + 4'd1;
else
  ovld_err_ifs_cnt <= 4'd0;
end

// block to check for the 1st '1' bit after txn. of  error flag
always @ (posedge clk or posedge g_rst)
   begin
      if(g_rst)
         init_bit <= 1'b0;
      else if (nxt_state == delm_init)
```

```
         begin
           if(sampled_bit)
               init_bit <= 1'b1;
           else init_bit <= 1'b0;
         end
      else init_bit <= 1'b0;
   end

// sequential always block
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         state <= idle;
      else state <= nxt_state;
   end

// block to determine nxt state
always @ (state or init_err_st_en or err_state or bt_err or ack_err or
stf_err or frm_err or crc_err or init_bit or ovld_err_flg_cnt or
cons_zero_cnt or dlm_cnt or ovld_err_ifs_cnt or tx_success or over_ld)
begin
case (state)
   idle:begin
      if (bt_err || ack_err || stf_err || frm_err || crc_err)
         begin
            nxt_state = err_flg;
          end
      else if (over_ld)
         begin
            nxt_state = ovld_flg;
          end
      else begin
             nxt_state = idle;
          end
    end
   err_flg:begin
       if ((err_state == 2'b00) || (err_state == 2'b01))
          begin
             if (ovld_err_flg_cnt < 3'b110)
               begin
                  nxt_state = err_flg;
                end
             else begin
                    nxt_state = delm_init;
                  end
           end
       else if (err_state == 2'b10)
            begin
                nxt_state = bus_off;
             end
       else nxt_state = idle;
    end
   ovld_flg:begin
          if (ovld_err_flg_cnt < 3'd6)
```

```
          begin
            nxt_state = ovld_flg;
           end
       else begin
             nxt_state = delm_init;
          end
 end
delm_init:begin
        if ((cons_zero_cnt <= 8'd1))
           begin
             nxt_state = delm_init;
           end
        else begin
           if (init_bit)
              begin
                 nxt_state = ovld_err_dlm;
               end
           else begin
                 nxt_state = delm_init;
               end
         end
   end
ovld_err_dlm:begin
        if (dlm_cnt < 3'd6)
           begin
                nxt_state = ovld_err_dlm;
            end
        else begin
                nxt_state = ovld_err_ifs;
            end
 end
ovld_err_ifs:begin
        if(err_state == 2'b00)
           begin
             if(ovld_err_ifs_cnt < 4'd3)
               begin
                    nxt_state = ovld_err_ifs;
               end
             else begin
                 nxt_state = ovld_err_comp;
             end
          end
        else if (err_state == 2'b01)
          begin
            if(ovld_err_ifs_cnt < 4'd11)
              begin
                  nxt_state = ovld_err_ifs;
              end
            else begin
                nxt_state = ovld_err_comp;
             end
          end
        else if (err_state == 2'b10)
           begin
```

```
                nxt_state = bus_off;
              end
           else begin
              nxt_state = idle;
             end
        end
     ovld_err_comp:begin
           nxt_state = idle;
        end
     bus_off:begin
         if(init_err_st_en)
            begin
              nxt_state = idle;
             end
          else begin
               nxt_state = bus_off;
              end
        end
     default: begin
                nxt_state = idle;
              end
   endcase
  end

// always block to determine output
always @ (posedge clk or posedge g_rst)
 begin
  if (g_rst)
     begin
        ovld_frm_tx <= 1'b0;
        act_err_frm_tx <= 1'b0;
        psv_err_frm_tx <= 1'b0;
        act_err_flg_tx <= 1'b0;
        psv_err_flg_tx <= 1'b0;
        ovld_flg_tx <= 1'b0;
        err_ovld_out <= 1'b1;
        cons_zero_flg <= 1'b0;
        ovld_err_ifs_tx <= 1'b0;
        ovld_err_tx_cmp <= 1'b0;
        fst_bit_zero_err  <= 1'b0;
      end
  else begin
   case (state)
     idle:begin
        if (bt_err || ack_err || stf_err || frm_err || crc_err ||
            over_ld)
           begin
            err_ovld_out <= 1'b1;
           end
        else
           begin
             ovld_frm_tx <= 1'b0;
             act_err_frm_tx <= 1'b0;
             psv_err_frm_tx <=1'b0;
```

```
        act_err_flg_tx <=1'b0;
        psv_err_flg_tx <=1'b0;
        ovld_flg_tx <= 1'b0;
        err_ovld_out <= 1'b1;
        cons_zero_flg <= 1'b0;
        ovld_err_ifs_tx <= 1'b0;
        ovld_err_tx_cmp <=1'b0;
        fst_bit_zero_err  <=1'b0;
      end
    end
err_flg:begin
     if (err_state == 2'b00)
          begin
              err_ovld_out  <=1'b0;
              act_err_frm_tx <= 1'b1;
              act_err_flg_tx <=1'b1;
           end
    else if (err_state == 2'b01)
        begin
              err_ovld_out  <=1'b0;
              psv_err_frm_tx <=1'b1;
              psv_err_flg_tx <=1'b1;
       end
    else if (err_state == 2'b10)
       begin
           err_ovld_out  <=1'b1;
           act_err_frm_tx <= 1'b0;
           psv_err_frm_tx <=1'b0;
           act_err_flg_tx <=1'b0;
           psv_err_flg_tx <=1'b0;
        end
    end
 ovld_flg:begin
          err_ovld_out <= 1'b0;
          ovld_frm_tx <= 1'b1;
          ovld_flg_tx <= 1'b1;
    end
 delm_init:begin
        if ((cons_zero_cnt <= 8'd2))
          begin
            fst_bit_zero_err  <=1'b0;
            err_ovld_out  <=1'b1;
            cons_zero_flg  <=1'b0;
            act_err_flg_tx <=1'b0;
            psv_err_flg_tx <=1'b0;
            ovld_flg_tx <=1'b0;
          end
        else if (cons_zero_cnt == 8'd3)
          begin
            if (init_bit)
              begin
                 fst_bit_zero_err  <=1'b0;
                 err_ovld_out  <=1'b1;
                 cons_zero_flg  <=1'b0;
```

```
                    act_err_flg_tx <=1'b0;
                    psv_err_flg_tx <=1'b0;
                    ovld_flg_tx <=1'b0;
                  end
                else begin
                    fst_bit_zero_err <= 1'b1;
                    err_ovld_out  <= 1'b1;
                    cons_zero_flg  <= 1'b1;
                    act_err_flg_tx <= 1'b0;
                    psv_err_flg_tx <= 1'b0;
                    ovld_flg_tx <= 1'b0;
                  end
              end
            else begin
              if (init_bit)
                begin
                 fst_bit_zero_err  <=1'b0;
                 cons_zero_flg  <=1'b0;
                 err_ovld_out  <=1'b1;
                 act_err_flg_tx <=1'b0;
                 psv_err_flg_tx <=1'b0;
                 ovld_flg_tx <=1'b0;
                end
              else begin
                 fst_bit_zero_err  <=1'b0;
                 cons_zero_flg  <=1'b1;
                 err_ovld_out  <=1'b1;
                 act_err_flg_tx <=1'b0;
                 psv_err_flg_tx <=1'b0;
                 ovld_flg_tx <=1'b0;
               end
          end
    end
ovld_err_dlm:begin
        err_ovld_out <= 1'b1;
        fst_bit_zero_err  <=1'b0;
        cons_zero_flg <=1'b0;
    end
ovld_err_ifs:begin
        if(err_state == 2'b01)
           begin
             if(ovld_err_ifs_cnt < 4'd11)
               begin
                    err_ovld_out <= 1'b1;
                    ovld_err_ifs_tx <=1'b1;
               end
             else begin
                 err_ovld_out <= 1'b1;
             end
          end
        else
          begin
            if(ovld_err_ifs_cnt < 4'd3)
              begin
```

```
                    err_ovld_out <= 1'b1;
                    ovld_err_ifs_tx <=1'b1;
                end
               else begin
                   err_ovld_out <= 1'b1;
               end
             end
          end
       ovld_err_comp:begin
             err_ovld_out <= 1'b1;
             psv_err_frm_tx <= 1'b0;
             act_err_frm_tx <= 1'b0;
             ovld_frm_tx <= 1'b0;
             ovld_err_ifs_tx <= 1'b0;
             ovld_err_tx_cmp <=1'b1;
          end
       bus_off:begin
             ovld_frm_tx <= 1'b0;
             act_err_frm_tx <= 1'b0;
             psv_err_frm_tx <= 1'b0;
             act_err_flg_tx <= 1'b0;
             psv_err_flg_tx <= 1'b0;
             ovld_flg_tx <= 1'b0;
             err_ovld_out <= 1'b1;
             cons_zero_flg <= 1'b0;
             ovld_err_ifs_tx <= 1'b0;
             ovld_err_tx_cmp <= 1'b0;
             fst_bit_zero_err  <= 1'b0;
          end
       default: begin
             ovld_frm_tx <= 1'b0;
             act_err_frm_tx <= 1'b0;
             psv_err_frm_tx <= 1'b0;
             act_err_flg_tx <= 1'b0;
             psv_err_flg_tx <= 1'b0;
             ovld_flg_tx <= 1'b0;
             err_ovld_out <= 1'b1;
             cons_zero_flg <= 1'b0;
             ovld_err_ifs_tx <= 1'b0;
             ovld_err_tx_cmp <= 1'b0;
             fst_bit_zero_err  <= 1'b0;
          end
     endcase
  end
end


// block to count the occurance of consecutive zero's after
error/overload flag
always @ (posedge clk or posedge g_rst)
   begin
     if (g_rst)
       begin
         cons_domn_err  <= 1'b0;
```

```
        cons_domn_err_cnt  <= 4'd0;
      end
    else if (ovld_err_tx_cmp || rx_success)
      begin
        cons_domn_err  <= 1'b0;
        cons_domn_err_cnt  <= 4'd0;
       end
    else if (cons_zero_flg)
      begin
        if (cons_zero_cnt == 8'd4)
           begin
              cons_domn_err  <= 1'b0;
              cons_domn_err_cnt  <= 4'd2;
            end
        else if (cons_domn_err_cnt == 4'd7)
           begin
              cons_domn_err <= 1'b1;
              cons_domn_err_cnt  <= 4'd0;
            end
        else begin
             cons_domn_err  <= 1'b0;
             cons_domn_err_cnt <= cons_domn_err_cnt + 4'd1;
            end
      end
  end


//tx. error limitation exception 1
always @ (posedge clk or posedge g_rst)
  begin
   if (g_rst)
      begin
         tx_excp_1 <= 1'b0;
         tx_excp_1_trig <= 1'b0;
      end
   else if (ovld_err_tx_cmp || rx_success)
      begin
         tx_excp_1 <= 1'b0;
         tx_excp_1_trig <= 1'b0;
      end
   else if ((err_state == 2'b01) && ack_err)
         tx_excp_1_trig <= 1'b1;
   else if(tx_excp_1_trig && psv_err_flg_tx)
      begin
        if(~sampled_bit)
             tx_excp_1 <= tx_excp_1;
        else tx_excp_1 <= 1'b1;
      end
   else tx_excp_1 <= 1'b0;
 end


//tx. error limitation exception 2
always @ (posedge clk or posedge g_rst)
```

```
   begin
      if (g_rst)
         begin
            tx_excp_2 <= 1'b0;
         end
      else if (ovld_err_tx_cmp || rx_success)
         begin
            tx_excp_2 <= 1'b0;
         end
      else if (arbtr_fld  & stf_err & tx_bit[1] & serial_in)
            tx_excp_2 <= 1'b1;
    end


//Block to increment the tx & rx error counters
always @ (posedge clk or posedge g_rst)
   begin
       if (g_rst)
         begin
           tx_err_cntr <= 9'd0;
           rx_err_cntr <= 9'd0;
         end
       else if (init_err_st_en)
         begin
           tx_err_cntr <= 9'd0;
           rx_err_cntr <= 9'd0;
         end
       else if (~txmtr)
          begin
            if (( stf_err || frm_err || crc_err ) && (~((act_err_flg_tx
               || ovld_flg_tx) && bt_err))) rx_err_cntr <= rx_err_cntr
               + 9'd1;
            else if ((fst_bit_zero_err && (act_err_frm_tx ||
                     psv_err_frm_tx )) || ((act_err_flg_tx ||
                     ovld_flg_tx) && bt_err) || cons_domn_err)
                rx_err_cntr <= rx_err_cntr + 9'd8;
            else if (rx_eof_success)
               begin
                  if (rx_err_cntr == 9'd0)
                    rx_err_cntr <= 9'd0;
                  else if ((rx_err_cntr >= 9'd1) || (rx_err_cntr <
                           9'd128))
                    rx_err_cntr <= rx_err_cntr - 9'd1;
                  else
                    rx_err_cntr <= 9'd120;
               end
            else rx_err_cntr <= rx_err_cntr;
          end
      else begin
             if ((((~tx_excp_1) && (~tx_excp_2)) && ( stf_err ||
                  frm_err || crc_err || bt_err || ack_err))||
                  (fst_bit_zero_err && (act_err_frm_tx ||
                  psv_err_frm_tx)) || (cons_domn_err)||
                  ((act_err_flg_tx || ovld_flg_tx) && bt_err))
```

```
                tx_err_cntr <= tx_err_cntr + 9'd8;
            else if (tx_eof_success)
                begin
                   if (tx_err_cntr == 9'd0)
                     tx_err_cntr <= 9'd0;
                   else tx_err_cntr <= tx_err_cntr - 9'd1;
                end
            else tx_err_cntr <= tx_err_cntr;
       end
  end

// Block to assign error state of node based on the counter values
always @ (posedge clk or posedge g_rst)
    begin
     if (g_rst)
       begin
          err_state <= 2'b00;
          bus_off_sts <= 1'b0;
        end
     else if (init_err_st_en)
        begin
           err_state <= 2'b00;
           bus_off_sts <= 1'b0;
         end
     else if((rx_err_cntr <= 9'd127 ) && ( tx_err_cntr <= 9'd127))
       begin
          err_state <= 2'b00;
          bus_off_sts <= 1'b0;
        end
     else if(((rx_err_cntr > 127) || (tx_err_cntr > 9'd127)) &&
            (tx_err_cntr <= 9'd255))
       begin
          err_state <= 2'b01;
          bus_off_sts <= 1'b0;
        end
     else if (tx_err_cntr > 9'd255)
       begin
          err_state <= 2'b10;
          bus_off_sts <= 1'b1;
        end
   end

endmodule
```

APPENDIX I

SERIALIZED FRAME TRANSMITTER

```
// module to serialize the output
module slzd_frm_tx (clk, g_rst, dt_rm_frm_tx, act_err_frm_tx,
psv_err_frm_tx, ovld_frm_tx, dt_rm_out, err_ovld_out, send_ack,
bus_off_sts, arbtr_sts, can_bus_out, abort_dt_rm_tx, tx_bit);

input clk;
input g_rst;
input dt_rm_frm_tx;
input act_err_frm_tx;
input psv_err_frm_tx;
input ovld_frm_tx;
input dt_rm_out;
input err_ovld_out;
input send_ack;
input bus_off_sts;
input arbtr_sts;

//outputs
output can_bus_out;
output [1:0] tx_bit;
output abort_dt_rm_tx;

reg can_bus_out;
reg [1:0] tx_bit;
reg abort_dt_rm_tx;
reg [1:0] state, nxt_state;

parameter [1:0] idle = 2'd0,
                dt_rm_tx = 2'd1,
                err_ovld_tx = 2'd2;

// sequential always block
always @ (posedge clk or posedge g_rst)
begin
if (g_rst)
   state <= idle;
else state <= nxt_state;
end

// combinational block to determine nxt state and output logic
always @ (state or act_err_frm_tx or psv_err_frm_tx or ovld_frm_tx or
dt_rm_frm_tx or bus_off_sts or send_ack or arbtr_sts or err_ovld_out or
dt_rm_out)
   begin
       case (state)
           idle: begin
               if (((act_err_frm_tx || psv_err_frm_tx) &&
                    (~bus_off_sts)) || ovld_frm_tx)
                  begin
```

```
            can_bus_out = err_ovld_out;
            nxt_state = err_ovld_tx;
            abort_dt_rm_tx = 1'b0;
         end
      else if (dt_rm_frm_tx && arbtr_sts && (~bus_off_sts))
         begin
            can_bus_out = dt_rm_out;
            nxt_state = dt_rm_tx;
            abort_dt_rm_tx = 1'b0;
         end
      else if ((~arbtr_sts) && send_ack && (~bus_off_sts))
         begin
            can_bus_out = 1'b0;
            nxt_state = idle;
            abort_dt_rm_tx = 1'b0;
          end
      else begin
          can_bus_out = 1'b1;
          nxt_state = idle;
          abort_dt_rm_tx = 1'b0;
         end
  end
 dt_rm_tx: begin
    if (((act_err_frm_tx || psv_err_frm_tx) &&
        (~bus_off_sts)) || ovld_frm_tx)
       begin
          can_bus_out = err_ovld_out;
          nxt_state = err_ovld_tx;
          abort_dt_rm_tx = 1'b1;
       end
    else if (dt_rm_frm_tx && arbtr_sts && (~bus_off_sts))
       begin
          can_bus_out = dt_rm_out;
          nxt_state = dt_rm_tx;
          abort_dt_rm_tx = 1'b0;
        end
    else begin
        can_bus_out = 1'b1;
        nxt_state = idle;
        abort_dt_rm_tx = 1'b0;
       end
  end
 err_ovld_tx: begin
     if (((act_err_frm_tx || psv_err_frm_tx) && (~bus_off_sts
         )) || ovld_frm_tx)
        begin
           can_bus_out = err_ovld_out;
           nxt_state = err_ovld_tx;
           abort_dt_rm_tx = 1'b0;
        end
    else begin
         can_bus_out = 1'b1;
         nxt_state = idle;
         abort_dt_rm_tx = 1'b0;
```

```
                end
            end
          default: begin
                can_bus_out = 1'b1;
                nxt_state = idle;
                abort_dt_rm_tx = 1'b0;
           end
      endcase
   end


// Reg to store the output of can bus for one clock cycle after
transmission of bit.
always @ (posedge clk or posedge g_rst)
   begin
       if (g_rst)
          tx_bit  <= 2'b00;
       else
          tx_bit  <=  {tx_bit[0], can_bus_out};
    end

endmodule
```

APPENDIX J

MESSAGE PROCESSOR

```
//Module to generate control signals and flags
module msg_processor (clk, g_rst, stf_err, bt_err, crc_err, ack_err,
frm_err, rcvd_eof_flg, rcvd_lst_bit_ifs, dt_rm_eof_tx_cmp,
txed_lst_bit_ifs, ovld_err_tx_cmp, rcvd_data_len, rcvd_bt_cnt,
de_stuff, act_err_frm_tx, psv_err_frm_tx, tx_buff_busy, arbtr_sts,
msg_due_tx, serial_in, rx_buff_0_wrtn, rx_buff_1_wrtn, bt_ack_err_pre,
stf_frm_crc_err_pre, rx_eof_success, rx_success, tx_eof_success,
tx_success, re_tran, send_ack, txmtr, over_ld);

// inputs
input clk;
input g_rst;
input stf_err;
input bt_err;
input crc_err;
input ack_err;
input frm_err;
input rcvd_eof_flg;
input rcvd_lst_bit_ifs;
input dt_rm_eof_tx_cmp;
input txed_lst_bit_ifs;
input ovld_err_tx_cmp;
input [6:0] rcvd_data_len;
input [6:0] rcvd_bt_cnt;
input de_stuff;
input act_err_frm_tx;
input psv_err_frm_tx;
input tx_buff_busy;
input arbtr_sts;
input msg_due_tx;
input serial_in;
input rx_buff_0_wrtn;
input rx_buff_1_wrtn;

//outputs
output bt_ack_err_pre;
output stf_frm_crc_err_pre;
output rx_eof_success;
output rx_success;
output tx_eof_success;
output tx_success;
output re_tran;
output send_ack;
output txmtr;
output over_ld;

//output registers
reg bt_ack_err_pre;
reg stf_frm_crc_err_pre;
```

```
reg rx_eof_success;
reg rx_success;
reg tx_eof_success;
reg tx_success;
reg re_tran;
reg send_ack;
reg txmtr;
reg over_ld;
reg msg_due_tx_reg;
reg tx_success_en;
reg rx_success_en;

// block to indicate if a stf crc or frm error occured
always @ (posedge clk or posedge g_rst)
    begin
       if (g_rst)
            stf_frm_crc_err_pre <= 1'b0;
       else if (ovld_err_tx_cmp)
            stf_frm_crc_err_pre <= 1'b0;
       else if ((rcvd_eof_flg || rcvd_lst_bit_ifs) && (~(act_err_frm_tx
             || psv_err_frm_tx)))
            stf_frm_crc_err_pre <= 1'b0;
       else if ((~arbtr_sts) && ((crc_err) || (stf_err) || (frm_err)))
            stf_frm_crc_err_pre <= 1'b1;
     end
// block to indicate the succesful reception of mesage till the eof
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         rx_eof_success <= 1'b0;
      else if (arbtr_sts || rx_success)
         rx_eof_success <= 1'b0;
      else if ((~arbtr_sts) && (rcvd_eof_flg) && (~stf_frm_crc_err_pre)
               && (~(act_err_frm_tx || psv_err_frm_tx)))
         rx_eof_success <= 1'b1;
      else rx_eof_success <= 1'b0;
  end

//block to indicate the succesful reception of message
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         rx_success_en <= 1'b0;
      else if ((~arbtr_sts) && rx_eof_success)
         rx_success_en <= 1'b1;
      else rx_success_en <= 1'b0;
  end

//block to indicate the succesful reception of message
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         rx_success <= 1'b0;
      else if (rx_success)
```

```
         rx_success <= 1'b0;
      else if (rx_success_en && rcvd_lst_bit_ifs && (~(act_err_frm_tx
               || psv_err_frm_tx)))
         rx_success <= 1'b1;
      else rx_success <= 1'b0;
  end
// block to indicate if a ack or bit error occured
always @ (posedge clk or posedge g_rst)
    begin
       if (g_rst)
            bt_ack_err_pre <= 1'b0;
       else if (ovld_err_tx_cmp)
            bt_ack_err_pre <= 1'b0;
       else if ((dt_rm_eof_tx_cmp || txed_lst_bit_ifs) &&
               (~(act_err_frm_tx || psv_err_frm_tx)))
            bt_ack_err_pre <= 1'b0;
       else if ((arbtr_sts) && ((bt_err) || (ack_err)))
            bt_ack_err_pre <= 1'b1;
     end

//block to indicate the succesful transmission of message  till eof
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         tx_eof_success <= 1'b0;
      else if ((~arbtr_sts) || tx_success)
         tx_eof_success <= 1'b0;
      else if (arbtr_sts && dt_rm_eof_tx_cmp && (~bt_ack_err_pre))
         tx_eof_success <= 1'b1;
      else tx_eof_success <= 1'b0;
     end
//block to indicate the succesful transmission of message
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         tx_success_en <= 1'b0;
      else if (arbtr_sts && tx_eof_success)
         tx_success_en <= 1'b1;
      else tx_success_en <= 1'b0;
  end

//block to indicate the succesful transmission of message
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         tx_success <= 1'b0;
      else if (tx_success)
         tx_success <= 1'b0;
      else if (tx_success_en && txed_lst_bit_ifs && (~(act_err_frm_tx
            || psv_err_frm_tx)))
         tx_success <= 1'b1;
      else tx_success <= 1'b0;
  end
```

```
// always block to generate send ack signal
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         send_ack <= 1'b0;
      else if ((~txmtr) && (rcvd_bt_cnt > 7'd20) && (rcvd_bt_cnt ==
              (7'd20 + rcvd_data_len + 7'd14))&& (~stf_frm_crc_err_pre)
               && (~de_stuff))
         send_ack <= 1'b1;
      else send_ack <= 1'b0;
    end

// block to indicate if a message is due for transmission
always @ (posedge clk or posedge g_rst)
   begin
       if (g_rst)
          msg_due_tx_reg <= 1'b0;
       else if (rcvd_lst_bit_ifs || ovld_err_tx_cmp || re_tran)
          msg_due_tx_reg <= 1'b0;
       else if (msg_due_tx)
          msg_due_tx_reg <= 1'b1;
   end

// block to initiate retransmission of message
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         re_tran <= 1'b0;
      else if (msg_due_tx_reg && tx_buff_busy && ((rcvd_lst_bit_ifs &&
              (~(act_err_frm_tx || psv_err_frm_tx)))  ||
               ovld_err_tx_cmp))
         re_tran <= 1'b1;
      else re_tran <= 1'b0;
    end
// block to indicate if the controller was/is a transmitter or receiver
always @ (posedge clk or posedge g_rst)
  begin
   if (g_rst)
      txmtr <= 1'b0;
   else if ((rcvd_bt_cnt == 7'd14) && arbtr_sts)
      txmtr <= 1'b1;
   else if (rcvd_lst_bit_ifs || txed_lst_bit_ifs)
      txmtr <= 1'b0;
   else  txmtr <= txmtr;
  end

// block to signal over load condition
always @ (posedge clk or posedge g_rst)
    begin
     if (g_rst)
        over_ld <= 1'b0;
     else if ((rx_success && rx_buff_0_wrtn && rx_buff_1_wrtn) ||
             ((~serial_in) && (rcvd_bt_cnt >= (7'd20 + rcvd_data_len +
              7'd14 + 7'd10)) && (rcvd_bt_cnt <= (7'd20 + rcvd_data_len
```

```
            + 7'd14 + 7'd12))))
        over_ld <= 1'b1;
     else over_ld <= 1'b0;
    end
endmodule
//module to produce arbtr_status
module arbtr_ctrl(osc_clk, g_rst, arbtr_fld, rcvd_lst_bit_ifs,
txed_lst_bit_ifs, ovld_err_tx_cmp, tx_buff_busy, bt_ack_err_pre,
bit_destf_intl, dt_rm_frm_tx, sampling_pt,  can_bus_out, can_bus_in,
act_err_frm_tx, psv_err_frm_tx, arbtr_sts, msg_due_tx);

input osc_clk;
input g_rst;
input arbtr_fld;
input rcvd_lst_bit_ifs;
input txed_lst_bit_ifs;
input ovld_err_tx_cmp;
input tx_buff_busy;
input bt_ack_err_pre;
input bit_destf_intl;
input dt_rm_frm_tx;
input sampling_pt;
input can_bus_out;
input can_bus_in;
input act_err_frm_tx;
input psv_err_frm_tx;

output arbtr_sts;
output msg_due_tx;

reg arbtr_sts;
reg msg_due_tx;

// block to send arbtr_sts signal
always @ (posedge osc_clk or posedge g_rst)
   begin
      if (g_rst)
         begin
            arbtr_sts <= 1'b1;
            msg_due_tx <= 1'b0;
          end
      else if (ovld_err_tx_cmp)
         begin
            arbtr_sts <= 1'b1;
            msg_due_tx <= 1'b0;
       end
      else if ((rcvd_lst_bit_ifs || txed_lst_bit_ifs) &&
               (~(act_err_frm_tx || psv_err_frm_tx)))
         begin
            arbtr_sts <= 1'b1;
            msg_due_tx <= 1'b0;
       end
      else if ((arbtr_sts && tx_buff_busy && bt_ack_err_pre) ||
            (arbtr_sts && tx_buff_busy && bit_destf_intl &&
```

```
            (~dt_rm_frm_tx)))
          begin
             arbtr_sts <= 1'b0;
             msg_due_tx <= 1'b1;
           end
      else if (arbtr_sts && (~tx_buff_busy) && bit_destf_intl &&
               (~dt_rm_frm_tx))
          begin
             arbtr_sts <= 1'b0;
             msg_due_tx <= 1'b0;
           end
      else if (sampling_pt && tx_buff_busy && arbtr_sts && arbtr_fld &&
               (can_bus_out != can_bus_in))
         begin
            arbtr_sts <= 1'b0;
            msg_due_tx <= 1'b1;
          end
      else begin
         arbtr_sts <= arbtr_sts;
         msg_due_tx <= msg_due_tx;
        end
    end
endmodule
```

APPENDIX K

SYNCHRONIZER

```
//Module to synchronize receiver to transmitter and send in serial bits
module synchronizer (osc_clk, g_rst, can_bus_in, rcvd_lst_bit_ifs, sjw,
ovld_err_tx_cmp, txed_lst_bit_ifs, arbtr_sts, clk, bit_destf_intl,
sampling_pt, sampled_bit);

// inputs
input osc_clk;
input g_rst;
input can_bus_in;
input [1:0] sjw;
input arbtr_sts;
input rcvd_lst_bit_ifs;
input ovld_err_tx_cmp;
input txed_lst_bit_ifs;

output clk;
output bit_destf_intl;
output sampling_pt;
output sampled_bit;

wire clk;

reg bit_destf_intl;
reg sampling_pt;
reg sampled_bit;
reg bus_idle;
reg re_sync;
reg sync_bit;
reg edge_detected;
reg [2:0] prop_seg_cnt;
reg [2:0] dly_cnt;
reg [2:0] ph_seg1_cnt;
reg [2:0] ph_seg2_cnt;
reg [2:0] clk_cnt;
reg [2:0] state, nxt_state;

parameter [2:0] sof = 3'd0,
                prop_seg = 3'd1,
                ph_seg1 = 3'd2,
                ph_seg2 = 3'd3,
                sync_seg = 3'd4;

assign clk = clk_cnt[2];

// Block to indicate bus_idle
always @ (posedge osc_clk or posedge g_rst)
  begin
    if (g_rst)
       begin
```

```
          bus_idle <= 1'b1;
          bit_destf_intl  <= 1'b0;
        end
    else if (((((~arbtr_sts) && (rcvd_lst_bit_ifs || ovld_err_tx_cmp))
             || (arbtr_sts && txed_lst_bit_ifs)) && can_bus_in))
       begin
          bus_idle <= 1'b1;
          bit_destf_intl  <= 1'b0;
        end
    else if ((~can_bus_in) && bus_idle)
       begin
          bus_idle <= 1'b0;
          bit_destf_intl <= 1'b1;
        end
   end

// block to sample can bus bit
always @ (posedge osc_clk or posedge g_rst)
   begin
       if (g_rst)
        begin
           sampled_bit <= 1'b1;
         end
       else if (sampling_pt)
          begin
             sampled_bit <= can_bus_in;
           end
     end

// block to increment prop_seg_cnt
always @ (posedge osc_clk or posedge g_rst)
  begin
    if (g_rst)
      prop_seg_cnt  <= 3'd1;// changed to 1 from 0
    else if ((~can_bus_in) && bus_idle)
      prop_seg_cnt <= 3'd1;
    else if ((nxt_state == prop_seg)|| (nxt_state == sof))
      prop_seg_cnt <= prop_seg_cnt + 3'd1;
    else
      prop_seg_cnt  <= 3'd0;
  end

// block to calculate dly_cnt to delay phase seg 1
always @ (posedge osc_clk or posedge g_rst)
  begin
    if (g_rst)
       dly_cnt  <= 3'd0;
    else if ((~can_bus_in) && bus_idle)
       dly_cnt  <= 3'd0;
    else if (re_sync)
       begin
           dly_cnt <= {1'b0, sjw};
         end
    else dly_cnt  <= 3'd0;
```

```
 end

// block to increment phase seg 1 cnt
always @ (posedge osc_clk or posedge g_rst)
  begin
    if (g_rst)
      ph_seg1_cnt <= 3'd0;
    else if ((~can_bus_in) && bus_idle)
      ph_seg1_cnt <= 3'd0;
    else if (nxt_state == ph_seg1)
      ph_seg1_cnt <= ph_seg1_cnt + 3'd1;
    else
      ph_seg1_cnt <= 3'd0;
  end

// block to increment phase seg 2 cnt
always @ (posedge osc_clk or posedge g_rst)
  begin
    if (g_rst)
      ph_seg2_cnt <= 3'd0;
    else if ((~can_bus_in) && bus_idle)
      ph_seg2_cnt <= 3'd0;
    else if (nxt_state == ph_seg2)
      ph_seg2_cnt <= ph_seg2_cnt + 3'd1;
    else
      ph_seg2_cnt <= 3'd0;
  end


// sequential always block
always @ (posedge osc_clk or posedge g_rst)
  begin
    if (g_rst)
       state <= prop_seg;
   else state <= nxt_state;
   end

// combinational always block to determine next state
always @ (state or bus_idle or edge_detected or re_sync or prop_seg_cnt
            or dly_cnt or ph_seg1_cnt or ph_seg2_cnt or can_bus_in or
             sync_bit)
  begin
     case (state)
         sof: begin
             if (prop_seg_cnt < 3'd4)
               begin
                 nxt_state = sof;
               end
             else begin
                 nxt_state = ph_seg1;
               end
           end
         prop_seg: begin
             if (prop_seg_cnt < 3'd4)
```

```
      begin
         if ((~can_bus_in) && bus_idle)
            begin
               nxt_state = sof;
            end
         else begin
               nxt_state = prop_seg;
            end
        end
    else begin
        if ((~can_bus_in) && bus_idle)
          begin
             nxt_state = sof;
           end
        else begin
             nxt_state = ph_seg1;
           end
      end
   end
ph_seg1: begin
    if (~re_sync)
       begin
          if ((~can_bus_in) && bus_idle)
             begin
               nxt_state = sof;
              end
          else begin
             nxt_state = ph_seg2;
             end
         end
    else begin
          if (ph_seg1_cnt < (dly_cnt) + 3'd1)
             begin
                if ((~can_bus_in) && bus_idle)
                   begin
                     nxt_state = sof;
                    end
                else begin
                   nxt_state = ph_seg1;
                   end
               end
          else begin
                if ((~can_bus_in) && bus_idle)
                   begin
                     nxt_state = sof;
                    end
                else begin
                   nxt_state = ph_seg2;
                   end
             end
      end
  end
ph_seg2: begin
     if ((sync_bit  == can_bus_in))
```

```
                begin
                  if (ph_seg2_cnt < 3'd2)
                     begin
                        if ((~can_bus_in) && bus_idle)
                           begin
                             nxt_state = sof;
                            end
                        else begin
                           nxt_state = ph_seg2;
                           end
                      end
                  else begin
                       if ((~can_bus_in) && bus_idle)
                          begin
                            nxt_state = sof;
                           end
                       else begin
                          nxt_state = sync_seg;
                          end
                     end
                end
             else begin
                  if ((~can_bus_in) && bus_idle)
                     begin
                       nxt_state = sof;
                      end
                  else begin
                       nxt_state = prop_seg;
                      end
                end
          end
        sync_seg: begin
            if ((~can_bus_in) && bus_idle)
               begin
                 nxt_state = sof;
                end
            else begin
                 nxt_state = prop_seg;
                end
          end
       default:begin
            nxt_state = prop_seg;
          end
      endcase
  end


//always block to determine next state
always @ (posedge osc_clk or posedge g_rst)
  begin
    if (g_rst)
     begin
      edge_detected  <=1'b1;
      sync_bit  <= 1'b1;
```

```
  re_sync <= 1'b0;
  sampling_pt  <= 1'b0;
 end
else begin
 case (state)
     sof: begin
         if (prop_seg_cnt < 3'd4)
           begin
             edge_detected  <=1'b1;
             sync_bit  <= can_bus_in;
             re_sync <= 1'b0;
             sampling_pt  <= 1'b0;
           end
         else begin
             edge_detected  <=1'b1;
             sync_bit  <= sync_bit;
             re_sync <= 1'b0;
             sampling_pt  <= 1'b0;
           end
       end
     prop_seg: begin
         if (prop_seg_cnt <= 3'd1)
           begin
             if (~edge_detected)
               begin
                 if (sync_bit && (~can_bus_in))
                    begin
                       sync_bit  <= can_bus_in;
                       edge_detected  <= 1'b1;
                       re_sync  <= 1'b0;
                       sampling_pt  <= 1'b0;
                     end
                 else if ((~sync_bit) && can_bus_in)
                    begin
                       sync_bit  <= can_bus_in;
                       edge_detected  <=1'b1;
                       re_sync <= 1'b0;
                       sampling_pt  <= 1'b0;
                     end
                 else begin
                       sync_bit  <= sync_bit;
                       edge_detected  <= 1'b0;
                       re_sync <= 1'b0;
                       sampling_pt  <= 1'b0;
                    end
               end
             else begin
                   sync_bit  <= sync_bit;
                   edge_detected  <=1'b1;
                   re_sync <= 1'b0;
                   sampling_pt  <= 1'b0;
                 end
           end
         else if (prop_seg_cnt > 3'd1 && prop_seg_cnt < 3'd4)
```

```
  begin
    if (~edge_detected)
      begin
        if (sync_bit && (~can_bus_in))
           begin
              sync_bit  <= can_bus_in;
              edge_detected  <=1'b1;
              re_sync  <= 1'b1;
              sampling_pt  <= 1'b0;
             end
        else if ((~sync_bit) && can_bus_in)
           begin
              sync_bit  <= can_bus_in;
              edge_detected  <=1'b1;
              re_sync <= 1'b0;
              sampling_pt  <= 1'b0;
            end
        else begin
              sync_bit  <= sync_bit;
              edge_detected  <= 1'b0;
              re_sync <= 1'b0;
              sampling_pt  <= 1'b0;
            end
       end
     else begin
         sync_bit  <= sync_bit;
         edge_detected  <=1'b1;
         re_sync <= re_sync;
         sampling_pt  <= 1'b0;
       end
   end
else begin
    if (~edge_detected)
       begin
         if (sync_bit && (~can_bus_in))
           begin
              sync_bit  <= can_bus_in;
              edge_detected  <=1'b1;
              re_sync <= 1'b1;
              sampling_pt  <= 1'b0;
             end
        else if ((~sync_bit) && can_bus_in)
           begin
              sync_bit  <= can_bus_in;
              edge_detected  <=1'b1;
              re_sync <= 1'b0;
              sampling_pt  <= 1'b0;
            end
        else begin
              sync_bit  <= sync_bit;
              edge_detected  <= 1'b0;
              re_sync <= 1'b0;
              sampling_pt  <= 1'b0;
            end
```

```
          end
          else begin
              sync_bit <= sync_bit;
              edge_detected <= 1'b1;
              re_sync <= re_sync;
              sampling_pt <= 1'b0;
           end
       end
    end
ph_seg1: begin
     if (~re_sync)
        begin
           edge_detected  <= 1'b0;
           sync_bit  <= sync_bit;
           sampling_pt <= 1'b1;
           re_sync <= 1'b0;
          end
     else begin
           if (ph_seg1_cnt < (dly_cnt + 3'd1))
              begin
                 edge_detected  <= 1'b0;
                 sync_bit  <= sync_bit;
                 sampling_pt <= 1'b0;
                 re_sync <= 1'b1;
               end
           else begin
                 edge_detected  <= 1'b0;
                 sync_bit  <= sync_bit;
                 sampling_pt <= 1'b1;
                 re_sync <= 1'b1;
              end
       end
  end
ph_seg2: begin
      if ((sync_bit  == can_bus_in))
        begin
          if (ph_seg2_cnt < 3'd2)
             begin
                sync_bit  <= sync_bit;
                edge_detected  <= 1'b0;
                sampling_pt  <= 1'b0;
                re_sync <= 1'b0;
              end
          else begin
               sync_bit  <= sync_bit;
               edge_detected  <= 1'b0;
               sampling_pt  <= 1'b0;
               re_sync <= 1'b0;
             end
         end
      else begin
           sync_bit  <= can_bus_in;
           edge_detected  <=1'b1;
           sampling_pt  <= 1'b0;
```

```
                re_sync <= 1'b0;
              end
          end
         sync_seg: begin
             if (sync_bit  == can_bus_in)
                begin
                  sync_bit  <= sync_bit;
                  sampling_pt  <= 1'b0;
                  edge_detected  <= 1'b0;
                  re_sync <= 1'b0;
               end
             else begin
                 sync_bit  <= can_bus_in;
                 sampling_pt  <= 1'b0;
                 edge_detected  <=1'b1;
                 re_sync <= 1'b0;
              end
          end
          default:begin
             sync_bit  <= can_bus_in;
             sampling_pt  <= 1'b0;
             edge_detected  <= 1'b0;
             re_sync <= 1'b0;
          end
       endcase
   end
end

// counter to generate a clock
always @ (posedge osc_clk or posedge g_rst)
   begin
      if (g_rst)
         clk_cnt <= 3'd7;
      else if ((~can_bus_in) && bus_idle)
         clk_cnt <= 3'd7;
      else clk_cnt <= clk_cnt - 3'd1;
  end

endmodule
```

APPENDIX L

BIT DE-STUFFING

```
// Module to destuff the received bits
module bit_destuff (clk, g_rst, arbtr_sts, bit_destf_intl, sampled_bit,
tx_success, rx_success, act_err_frm_tx, psv_err_frm_tx, ovld_frm_tx,
serial_in, rx_crc_frm, rcvd_bt_cnt, de_stuff, one_count, zero_count,
rcvd_eof_flg, rcvd_msg_id, rcvd_rtr, rcvd_dlc, rcvd_crc, rx_crc_intl,
rx_crc_enable, rcvd_data_len, rcvd_data_frm, rcvd_lst_bit_ifs,
rcvd_crc_flg, rcvd_lst_bit_eof);

input clk;
input g_rst;
input arbtr_sts;
input bit_destf_intl;
input sampled_bit;
input tx_success;
input rx_success;
input act_err_frm_tx;
input psv_err_frm_tx;
input ovld_frm_tx;
input [14:0] rx_crc_frm;

output serial_in;
output [6:0] rcvd_bt_cnt;
output [2:0] one_count;
output [2:0] zero_count;
output rx_crc_intl;
output rx_crc_enable;
output [6:0] rcvd_data_len;
output rcvd_eof_flg;
output [10:0] rcvd_msg_id;
output rcvd_rtr;
output [3:0] rcvd_dlc;
output [14:0] rcvd_crc;
output [63:0] rcvd_data_frm;
output rcvd_lst_bit_ifs;
output de_stuff;
output rcvd_crc_flg;
output rcvd_lst_bit_eof;

reg rcvd_lst_bit_eof;
reg serial_in;
reg [6:0] rcvd_bt_cnt;
reg rx_crc_intl;
reg rx_crc_enable;
reg [6:0] rcvd_data_len;
reg rcvd_crc_flg;
reg rcvd_eof_flg;
reg [10:0] rcvd_msg_id;
reg rcvd_rtr;
reg [3:0] rcvd_dlc;
```

```
reg [14:0] rcvd_crc;
reg [63:0] rcvd_data_frm;
reg rcvd_lst_bit_ifs;
reg de_stuff;
reg stuff_bit;
reg [63:0] destf_out;
reg [6:0] rcvd_data_frm_pad;
reg [2:0] eof_bit_cnt;
reg [1:0] ifs_bit_cnt;
reg [2:0] one_count;
reg [2:0] zero_count;
reg [3:0] state, nxt_state;
reg [6:0] rcvd_frm_len;

parameter [3:0] idle = 4'd0,
                zero_4 = 4'd1,
                zero_5 = 4'd2,
                one_destf = 4'd3,
                ones_4 = 4'd4,
                ones_5 = 4'd5,
                zero_destf = 4'd6,
                crc_delim = 4'd7,
                ack_slot = 4'd8,
                ack_delim = 4'd9,
                eof = 4'd10,
                ifs = 4'd11,
                frm_cmp = 4'd12;

always @ (posedge clk or posedge g_rst)
   begin
      if(g_rst)
         serial_in <= 1'b1;
      else serial_in <= sampled_bit;
    end

// block to increment one_count
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         one_count <= 3'd0;
      else if (state == idle)
         one_count <= 3'd0;
      else if ((nxt_state == ones_4) || (nxt_state == ones_5)
            ||(nxt_state == one_destf))
         one_count <= one_count + 3'd1;
      else
         one_count <=3'd0;
   end

// block to increment zero_count
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         zero_count <=3'd0;
```

```
      else if ((state == idle) && (nxt_state == zero_4))
         zero_count <= 3'd0;
      else if ((nxt_state == zero_4)||(nxt_state == zero_5)||(nxt_state
               == zero_destf))
         zero_count <= zero_count + 3'd1;
      else
         zero_count <=3'd0;
   end

// block to increment eof cnt
always @ (posedge clk or posedge g_rst)
  begin
     if (g_rst)
        eof_bit_cnt <=3'd0;
     else if (nxt_state == eof)
        eof_bit_cnt <= eof_bit_cnt + 3'd1;
     else
        eof_bit_cnt <=3'd0;
  end

// block to increment ifs cnt
always @ (posedge clk or posedge g_rst)
  begin
     if (g_rst)
        ifs_bit_cnt <=2'd0;
     else if (nxt_state == ifs)
        ifs_bit_cnt <= ifs_bit_cnt + 2'd1;
     else
        ifs_bit_cnt <=2'd0;
  end

 // Block to calculate the received frame length and received data
length
 always @ (posedge clk or posedge g_rst)
    begin
       if (g_rst)
          begin
             rcvd_frm_len <= 7'd0;
             rcvd_data_len <= 7'd0;
             rcvd_data_frm_pad <= 7'd64;
           end
       else if (rx_success || tx_success)
          begin
            rcvd_frm_len <= 7'd0;
            rcvd_data_len <= 7'd0;
            rcvd_data_frm_pad <= 7'd64;
          end
       else if (rcvd_rtr)
          begin
              rcvd_data_len <= 7'd0;
              rcvd_data_frm_pad <= 7'd64;
              rcvd_frm_len <= 7'd34;
            end
       else if ((rcvd_rtr == 1'b0) && (rcvd_bt_cnt == 7'd21))
```

```
          begin
             rcvd_data_len <= (7'd8 *(7'd8 *destf_out[3]+ 7'd4*
             destf_out[2] + 7'd2 * destf_out[1] + 7'd1* destf_out[0]));
             rcvd_data_frm_pad <= (7'd64 - (7'd8 *(7'd8 *destf_out[3]+
             7'd4* destf_out[2] + 7'd2 * destf_out[1] + 7'd1*
                  destf_out[0])));
             rcvd_frm_len <= (7'd19 + (7'd8 *(7'd8 *destf_out[3]+ 7'd4*
             destf_out[2] + 7'd2 * destf_out[1] + 7'd1* destf_out[0]))
             + 7'd15);
           end
       else rcvd_frm_len <= rcvd_frm_len;
    end

// block to increment rcvd_bt_cnt
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         rcvd_bt_cnt <= 7'd0;
      else if (rx_success)
         rcvd_bt_cnt <= 7'd0;
      else if ((nxt_state == zero_4)|| (nxt_state == zero_5)||
            nxt_state == ones_4)||  (nxt_state == ones_5)
           || (nxt_state == crc_delim) || (nxt_state == ack_slot) ||
            (nxt_state == ack_delim) || (nxt_state == eof) ||
            (nxt_state == ifs))
         rcvd_bt_cnt <= rcvd_bt_cnt + 7'd1;
      else if (nxt_state == idle)
         rcvd_bt_cnt <= 7'd0;
    end

// sequential always block
always @ (posedge clk or posedge g_rst)
  begin
     if (g_rst)
          state <= idle;
     else state <= nxt_state;
   end

//  block to determine nxt state
always @ (state  or bit_destf_intl or one_count or zero_count or
eof_bit_cnt or ifs_bit_cnt or rx_success or act_err_frm_tx or
psv_err_frm_tx or ovld_frm_tx or rcvd_bt_cnt or rcvd_frm_len or
serial_in)
   begin
       case(state)
           idle:begin
               if ((bit_destf_intl) && (~(act_err_frm_tx ||
                    psv_err_frm_tx || ovld_frm_tx)))
                  nxt_state = zero_4;
               else nxt_state = idle;
             end
            zero_4:begin
                if ((rcvd_bt_cnt > 7'd20) && (rcvd_bt_cnt ==
                  rcvd_frm_len + 7'd1))
```

```
       begin
          nxt_state = crc_delim;
        end
    else if  (~(act_err_frm_tx || psv_err_frm_tx ||
            ovld_frm_tx))
          begin
            if (~serial_in)
             begin
                if (zero_count < 3'd4)
                   nxt_state =  zero_4;
                else
                   nxt_state =  zero_5;
              end
            else begin
                nxt_state = ones_4;
             end
          end
    else begin
        nxt_state = idle;
      end
 end
zero_5:begin
    if ((rcvd_bt_cnt > 7'd20) && (rcvd_bt_cnt ==
        rcvd_frm_len + 7'd1))
       begin
          nxt_state = crc_delim;
        end
    else if (~(act_err_frm_tx || psv_err_frm_tx ||
             ovld_frm_tx))
       begin
         if (~serial_in)
           begin
             nxt_state =  idle;
           end
         else
            begin
              nxt_state = one_destf;
             end
        end
    else begin
          nxt_state =  idle;
       end
 end
one_destf:begin
    if ((rcvd_bt_cnt > 7'd20) && (rcvd_bt_cnt ==
        rcvd_frm_len + 7'd1))
       begin
          nxt_state = crc_delim;
        end
    else if (~(act_err_frm_tx || psv_err_frm_tx ||
             ovld_frm_tx))
       begin
         if (~serial_in)
             begin
```

```
                nxt_state =  zero_4;
              end
          else begin
              nxt_state = ones_4;
            end
        end
    else begin
          nxt_state =  idle;
       end
 end
ones_4:begin
    if ((rcvd_bt_cnt > 7'd20) && (rcvd_bt_cnt ==
        rcvd_frm_len + 7'd1))
       begin
          nxt_state = crc_delim;
        end
    else if  (~(act_err_frm_tx || psv_err_frm_tx ||
           ovld_frm_tx))
         begin
           if (serial_in)
            begin
               if (one_count < 3'd4)
                  nxt_state =  ones_4;
               else
                  nxt_state =  ones_5;
             end
           else nxt_state = zero_4;
          end
    else begin
        nxt_state = idle;
      end
 end
ones_5:begin
    if ((rcvd_bt_cnt > 7'd20) && (rcvd_bt_cnt ==
        rcvd_frm_len + 7'd1))
       begin
          nxt_state = crc_delim;
        end
    else if (~(act_err_frm_tx || psv_err_frm_tx ||
           ovld_frm_tx))
       begin
         if (serial_in)
           begin
             nxt_state =  idle;
           end
         else
             begin
                nxt_state = zero_destf;
              end
        end
    else begin
          nxt_state =  idle;
       end
 end
```

```
zero_destf:begin
    if ((rcvd_bt_cnt > 7'd20) && (rcvd_bt_cnt ==
      rcvd_frm_len + 7'd1))
       begin
          nxt_state = crc_delim;
        end
    else if (~(act_err_frm_tx || psv_err_frm_tx ||
            ovld_frm_tx))
       begin
         if (~serial_in)
            begin
              nxt_state =  zero_4;
            end
         else begin
             nxt_state = ones_4;
           end
        end
    else begin
          nxt_state =  idle;
       end
 end
 crc_delim: begin
     nxt_state = ack_slot;
   end
 ack_slot:begin
     nxt_state = ack_delim;
   end
 ack_delim: begin
     nxt_state = eof;
   end
 eof:begin
     if(eof_bit_cnt < 3'd7)
        begin
          nxt_state = eof;
         end
      else begin
          nxt_state = ifs;
         end
   end
 ifs: begin
     if (ifs_bit_cnt < 2'd3)
        begin
          nxt_state = ifs;
         end
     else begin
          nxt_state = frm_cmp;
      end
   end
 frm_cmp: begin
     nxt_state = idle;
   end
 default:begin
     nxt_state = idle;
   end
```

```
         endcase
      end

// block to determine output
always @ (posedge clk or posedge g_rst)
  begin
      if (g_rst)
         begin
             destf_out <= 64'd0;
         end
      else begin
        case (state)
            idle: begin
                destf_out <= 64'd0;
               end
            zero_4: begin
                destf_out <= {destf_out [62:0], 1'b0};
              end
            zero_5: begin
                destf_out <= {destf_out [62:0], 1'b0};
              end
            one_destf: begin
               destf_out <= destf_out;
              end
            ones_4: begin
                destf_out <= {destf_out [62:0], 1'b1};
              end
            ones_5: begin
                destf_out <= {destf_out [62:0], 1'b1};
              end
            zero_destf: begin
                destf_out <= destf_out;
              end
            crc_delim: begin
                destf_out <= destf_out;
              end
            ack_slot:begin
                destf_out <= destf_out;
              end
            ack_delim: begin
                destf_out <= destf_out;
              end
            eof:begin
                destf_out <= destf_out;
              end
            ifs: begin
                destf_out <= destf_out;
              end
            frm_cmp: begin
                destf_out <= 64'd0;
              end
            default:begin
              destf_out <= 64'd0;
              end
```

```
      endcase
  end
end

// always block to indicate bit destuff
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         stuff_bit <= 1'b0;
      else if ((nxt_state == zero_5) || (nxt_state == ones_5))
         stuff_bit <= 1'b1;
      else stuff_bit <= 1'b0;
    end

// always block to indicate bit destuff
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         de_stuff <= 1'b0;
      else if (stuff_bit)
         de_stuff <= 1'b1;
      else de_stuff <= 1'b0;
    end

// output generation block
always @ (posedge clk or posedge g_rst)
   begin
     if (g_rst)
       begin
          rcvd_lst_bit_eof <= 1'b0;
          rcvd_eof_flg <= 1'b0;
          rcvd_lst_bit_ifs <= 1'b0;
        end
     else if ( rx_success || (rcvd_bt_cnt == 7'd0))
       begin
          rcvd_lst_bit_eof <= 1'b0;
          rcvd_eof_flg <= 1'b0;
          rcvd_lst_bit_ifs <= 1'b0;
        end
     else if (~rcvd_rtr)
       begin
         if ((rcvd_bt_cnt == (7'd20 + rcvd_data_len + 7'd14 + 7'd9)) &&
             (~de_stuff))
              rcvd_lst_bit_eof <= 1'b1;
         else if ((rcvd_bt_cnt == (7'd20 + rcvd_data_len + 7'd14 +
                  7'd10)) && (~de_stuff))
             begin
                rcvd_lst_bit_eof <= 1'b0;
                rcvd_eof_flg <= 1'b1;
              end
         else if ((rcvd_bt_cnt == (7'd20 + rcvd_data_len + 7'd14 +
                  7'd11)) && (~de_stuff))
             rcvd_eof_flg <= 1'b0;
```

```
         else if ((rcvd_bt_cnt == (7'd20 + rcvd_data_len + 7'd14 +
                  7'd12)) && (~de_stuff))
            begin
              rcvd_lst_bit_ifs <= 1'b1;
             end
        else begin
             rcvd_lst_bit_ifs <= 1'b0;
         end
      end
    else begin
        if (rcvd_bt_cnt == 7'd43)
            rcvd_lst_bit_eof <= 1'b1;
        else if (rcvd_bt_cnt == 7'd44)
            begin
               rcvd_lst_bit_eof <= 1'b0;
               rcvd_eof_flg <= 1'b1;
             end
        else if (rcvd_bt_cnt == 7'd45)
            rcvd_eof_flg <= 1'b0;
        else if (rcvd_bt_cnt == 7'd46)
           begin
             rcvd_lst_bit_ifs <= 1'b1;
            end
        else begin
            rcvd_lst_bit_ifs <= 1'b0;
          end
     end
end

// block to store received mag identifier
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         begin
            rcvd_msg_id <= 11'd0;
             end
         else if (rx_success || tx_success)
            begin
            rcvd_msg_id <= 11'd0;
          end
      else if((rcvd_bt_cnt == 7'd14)&& (~de_stuff))
            begin
            rcvd_msg_id <=  destf_out[10:0];
             end
      else if (rcvd_bt_cnt > 7'd14)
            begin
            rcvd_msg_id <= rcvd_msg_id;
         end
      else rcvd_msg_id <=  11'd0 ;
    end

// block to store received rtr bit
always @ (posedge clk or posedge g_rst)
   begin
```

```
      if (g_rst)
         begin
            rcvd_rtr <= 1'b0;
          end
      else if (rx_success || tx_success)
         begin
            rcvd_rtr <= 1'b0;
          end
      else if (rcvd_bt_cnt == 7'd15 && (~de_stuff))
         begin
           rcvd_rtr <= destf_out[0];
         end
      else if (rcvd_bt_cnt > 7'd15)
       begin
         rcvd_rtr <= rcvd_rtr;
         end
      else rcvd_rtr <= 1'b0;
    end

// block to store received  dlc
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         begin
            rcvd_dlc <= 4'd0;
          end
      else if (rx_success || tx_success)
          begin
             rcvd_dlc <= 4'd0;
           end
      else if (rcvd_bt_cnt == 7'd21 && (~de_stuff))
         begin
            rcvd_dlc <= destf_out[3:0];
          end
      else if  (rcvd_bt_cnt > 7'd21)
         begin
                rcvd_dlc <= rcvd_dlc;
          end
      else rcvd_dlc <= 4'd0;
    end

//Block to store received data frame
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         begin
              rcvd_data_frm <= 64'd0;
         end
      else if (rx_success || tx_success)
         begin
           rcvd_data_frm <= 64'd0;
         end
      else if ((rcvd_bt_cnt > 7'd21) && (rcvd_bt_cnt == ( 7'd21 +
             rcvd_data_len)) && (~rcvd_rtr) && (~de_stuff) )
```

```
         rcvd_data_frm  <= destf_out[63:0];
      else if ((rcvd_bt_cnt == (7'd22 + rcvd_data_len))&& (~de_stuff))
         rcvd_data_frm  <= rcvd_data_frm << rcvd_data_frm_pad;
      else if ((rcvd_bt_cnt == (7'd23 + rcvd_data_len)) && (~de_stuff))
           begin
             rcvd_data_frm  <= rcvd_data_frm >> rcvd_data_frm_pad;
            end
      else if  ((rcvd_bt_cnt > (7'd23 + rcvd_data_len))&& (~de_stuff))
            begin
              rcvd_data_frm  <= rcvd_data_frm;
          end
      else rcvd_data_frm  <= rcvd_data_frm;
    end

// Block to store received CRC frame
always @ (posedge clk or posedge g_rst)
    begin
       if (g_rst)
          begin
             rcvd_crc <= 15'd0;
             rcvd_crc_flg <= 1'b0;
           end
       else if (rx_success || tx_success)
          begin
             rcvd_crc <= 15'd0;
             rcvd_crc_flg <= 1'b0;
           end
       else if ((rcvd_bt_cnt > 7'd21) &&  (rcvd_bt_cnt == (rcvd_frm_len
            + 7'd2)) && (~de_stuff))
          begin
             rcvd_crc  <=  destf_out[14:0];
             rcvd_crc_flg <= 1'b1;
              end
       else if ((rcvd_bt_cnt > 7'd21) && (rcvd_bt_cnt > (rcvd_frm_len +
                  7'd2)))
          begin
             rcvd_crc  <= rcvd_crc ;
             rcvd_crc_flg <= 1'b0;
           end
       else  rcvd_crc  <= 15'd0 ;
     end

// Block to enable crc module
always @ (posedge clk or posedge g_rst)
   begin
       if (g_rst)
          begin
               rx_crc_enable <= 1'b0;
            rx_crc_intl <= 1'b1;
              end
       else if (rcvd_bt_cnt == 7'd0)
          begin
            rx_crc_enable <= 1'b0;
            rx_crc_intl <= 1'b1;
```

```
            end
        else if (~rcvd_rtr)
               begin
               if ((nxt_state == ones_5) || (nxt_state == zero_5))
               begin
                rx_crc_enable <= 1'b0;
              end
              else if ((rcvd_bt_cnt > 7'd21) && (rcvd_bt_cnt == 7'd19 +
                     rcvd_data_len) && (~stuff_bit))
                  begin
                      rx_crc_enable <= 1'b0;
                   end
              else if ((rcvd_bt_cnt > 7'd21) && (rcvd_bt_cnt > 7'd19 +
                    rcvd_data_len))
                     begin
                         rx_crc_enable <= 1'b0;
                      end
              else begin
                    rx_crc_enable <= 1'b1;
                    rx_crc_intl <= 1'b0;
                   end
           end
        else begin
         if ((nxt_state == ones_5) || (nxt_state == zero_5))
             begin
                 rx_crc_enable <= 1'b0;
              end
         else if ((rcvd_bt_cnt == 7'd19) && (~stuff_bit))
             begin
                 rx_crc_enable <= 1'b0;
              end
         else if (rcvd_bt_cnt > 7'd19)
               begin
                 rx_crc_enable <= 1'b0;
              end
         else rx_crc_enable <= 1'b1;
            end
     end

endmodule
```

APPENDIX M

BIT STUFF MONITOR

```
// Module to indicate stuff errors
module bit_stuff_monitor (clk, g_rst, serial_in, arbtr_fld, one_count,
zero_count, stf_err);

input clk;
input g_rst;
input serial_in;
input arbtr_fld;

input [2:0] one_count;
input [2:0] zero_count;

output stf_err;

reg stf_err;

always @ (posedge clk or posedge g_rst)
  begin
   if (g_rst)
         begin
            stf_err <= 1'b0;
          end
   else if (arbtr_fld)
         begin
            stf_err <= 1'b0;
          end
   else if (one_count == 3'd5)
        if (serial_in)
           begin
             stf_err <= 1'b1;
           end
        else
          begin
            stf_err <= 1'b0;
           end
   else if (zero_count == 3'd5)
      if (~serial_in)
         begin
           stf_err <= 1'b1;
         end
      else
         begin
           stf_err <= 1'b0;
         end
   else begin
          stf_err <= 1'b0;
         end
  end
endmodule
```

APPENDIX N

CRC CHECKER

```
// Module to indicate crc error
module crc_checker (clk, g_rst, rx_success, act_err_frm_tx,
psv_err_frm_tx, rx_crc_frm,rcvd_crc_flg, rcvd_crc, crc_err);

input clk;
input g_rst;
input rx_success;
input act_err_frm_tx;
input psv_err_frm_tx;
input rcvd_crc_flg;
input [14:0] rcvd_crc;
input [14:0] rx_crc_frm;

output crc_err;

reg crc_err;

// Always block to generate crc error signal
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         crc_err <= 1'b0;
      else if (rx_success || act_err_frm_tx || psv_err_frm_tx)
         crc_err <= 1'b0;
      else if (rcvd_crc_flg)
         begin
            if (rcvd_crc != rx_crc_frm)
               crc_err <= 1'b1;
            else crc_err <= 1'b0;
          end
      else crc_err <= 1'b0;
    end

endmodule
```

APPENDIX O

FORM CHECKER

```
// module to indicate form error
module form_checker (clk, g_rst, rx_success, act_err_frm_tx,
psv_err_frm_tx, rcvd_bt_cnt, rcvd_data_len, serial_in, frm_err);

input clk;
input g_rst;
input rx_success;
input act_err_frm_tx;
input psv_err_frm_tx;
input serial_in;
input [6:0] rcvd_bt_cnt;
input [6:0] rcvd_data_len;

output frm_err;
reg frm_err;

// Always block to generate form error signal
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
          frm_err <= 1'b0;
      else if (rx_success || act_err_frm_tx || psv_err_frm_tx)
          frm_err <= 1'b0;
      else if ((rcvd_bt_cnt == (7'd20 + rcvd_data_len + 7'd14 + 7'd1))
             ||(rcvd_bt_cnt == (7'd20 + rcvd_data_len + 7'd14 + 7'd3))
             ||((rcvd_bt_cnt >= (7'd20 + rcvd_data_len + 7'd14 + 7'd4))
             &&(rcvd_bt_cnt <= (7'd20 + rcvd_data_len + 7'd14 +
             7'd10))))
           begin
              if (~serial_in)
                frm_err <= 1'b1;
              else
                frm_err <= 1'b0;
           end
      else frm_err <= 1'b0;
  end
endmodule
```

APPENDIX P

BIT MONITOR

```
// Module to indicate bit error
module bit_monitor (clk, g_rst, can_bus_out, sampled_bit, dt_rm_frm_tx,
act_err_flg_tx, psv_err_flg_tx, ovld_flg_tx, cons_zero_flg,
ovld_err_ifs_tx, tx_success, arbtr_fld, ack_slt, ifs_flg_tx, arbtr_sts,
bt_err);
input clk;
input g_rst;
input can_bus_out;
input sampled_bit;
input dt_rm_frm_tx;
input act_err_flg_tx;
input psv_err_flg_tx;
input ovld_flg_tx;
input cons_zero_flg;
input ovld_err_ifs_tx;
input tx_success;
input arbtr_fld;
input arbtr_sts;
input ack_slt;
input ifs_flg_tx;
output bt_err;
reg bt_err;
reg arbtr_sts_en;
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         arbtr_sts_en <= 1'b0;
      else if (arbtr_sts)
         arbtr_sts_en <= 1'b1;
      else arbtr_sts_en <= 1'b0;
    end
always @ (posedge clk or posedge g_rst)
   begin
   if (g_rst)
         bt_err <= 1'b0;
   else if((can_bus_out == sampled_bit) || (arbtr_sts_en && (arbtr_fld
            || ack_slt || ifs_flg_tx) && can_bus_out && (~sampled_bit))
            || (psv_err_flg_tx && can_bus_out && (~sampled_bit))
            ||(cons_zero_flg && can_bus_out && (~sampled_bit)) ||
            (ovld_err_ifs_tx && can_bus_out && (~sampled_bit)))
              bt_err <= 1'b0;
   else if ((dt_rm_frm_tx && arbtr_sts_en && (~(arbtr_fld || ack_slt ||
            ifs_flg_tx)) && (can_bus_out != sampled_bit)) ||
            ((~can_bus_out) && (sampled_bit) && (ovld_flg_tx
            ||act_err_flg_tx)))
             bt_err <= 1'b1;
    else bt_err <= 1'b0;
  end
endmodule
```

APPENDIX Q

ACKNOWLEDGEMENT CHECKER

```
//module to check for acknowledgement error
module ack_checker (clk, g_rst, ack_slt, sampled_bit, arbtr_sts,
                  act_err_frm_tx, psv_err_frm_tx, tx_success, ack_err);

// inputs
input clk;
input g_rst;
input ack_slt;
input act_err_frm_tx;
input psv_err_frm_tx;
input arbtr_sts;
input tx_success;
input sampled_bit;

output ack_err;
reg ack_err;

always @ (posedge clk or posedge g_rst)
    begin
       if (g_rst)
                    ack_err <= 1'b0;
       else if (tx_success || act_err_frm_tx || psv_err_frm_tx)
          ack_err <= 1'b0;
       else if (arbtr_sts && ack_slt && sampled_bit)
             ack_err <= 1'b1;
       else ack_err <= 1'b0;
     end

endmodule
```

APPENDIX R

ACCEPTANCE CHECKER

```
// module to check the acceptance status of the msg
module accp_checker (clk, g_rst, arbtr_sts, mask_param, code_param,
rcvd_lst_bit_eof, stf_frm_crc_err_pre,bt_ack_err_pre, rcvd_msg_id,
acpt_sts);

input clk;
input g_rst;
input arbtr_sts;
input rcvd_lst_bit_eof;
input [10:0] rcvd_msg_id;
input bt_ack_err_pre;
input stf_frm_crc_err_pre;
input [10:0] mask_param;      // input to the acceptance filter
input [10:0] code_param;      // input to the acceptance filter

output acpt_sts;

reg acpt_sts;
reg arbtr_sts_en;

always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         arbtr_sts_en <= 1'b0;
      else if (arbtr_sts)
         arbtr_sts_en <= 1'b1;
      else arbtr_sts_en <= 1'b0;
    end

always @ (posedge clk or posedge g_rst)
   begin
      if(g_rst)
         acpt_sts <= 1'b0;
      else if((~arbtr_sts_en) && rcvd_lst_bit_eof &&
              (~(stf_frm_crc_err_pre || bt_ack_err_pre)))
         begin
            if(((code_param ^ rcvd_msg_id) & mask_param) ==
               11'b00000000000)
               acpt_sts <= 1'b1;
            else
               acpt_sts <= 1'b0;
        end
      else acpt_sts <= 1'b0;
   end
endmodule
```

APPENDIX S

RECEIVE BUFFERS

```
// Module to load receive buffers with accepted message
module rx_buff (clk, g_rst, acpt_sts, rd_en, rcvd_msg_id, rcvd_rtr,
rcvd_dlc, rcvd_data_frm, buff_rdy, rx_buff_0_wrtn, rx_buff_1_wrtn,
data_out);

input clk;
input g_rst;
input acpt_sts;
input rd_en;
input [10:0] rcvd_msg_id;
input rcvd_rtr;
input [3:0] rcvd_dlc;
input [63:0] rcvd_data_frm;

output [7:0] data_out;
output buff_rdy;
output rx_buff_0_wrtn;
output rx_buff_1_wrtn;

reg [7:0] data_out;
reg rd_en_in;
reg read_en;
reg [7:0] rx_buff0 [0:9];
reg [7:0] rx_buff1 [0:9];
reg [3:0] read_count;
reg rx_buff_0_wrtn;
reg rx_buff_1_wrtn;
reg rx_buff_0_read;
reg rx_buff_1_read;
reg rx_buff_0_wr_stat;
reg rx_buff_1_wr_stat;
reg rx_buff_0_active;
reg rx_buff_1_active;
reg buff_rdy;

//block to synchronize rd_en signal
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         rd_en_in <= 1'b1;
      else if (~rd_en)
         rd_en_in <= 1'b0;
      else
         rd_en_in <= 1'b1;
     end

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

```
         read_en <= 1'b1;
      else if (~rd_en_in)
         read_en <= 1'b0;
      else
         read_en <= 1'b1;
     end

// block to indicate buffers are ready
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         buff_rdy <= 1'b0;
      else if (rx_buff_0_wrtn || rx_buff_1_wrtn)
         buff_rdy <= 1'b1;
      else buff_rdy <= 1'b0;
   end

// block to load receive buffers with accepted message
always @(posedge clk or posedge g_rst)
 begin
  if(g_rst)
   begin
     rx_buff_0_wr_stat <= 1'b1;
     rx_buff_0_wrtn <= 1'b0;
     rx_buff0[0] <= 8'd0;
     rx_buff0[1] <= 8'd0;
     rx_buff0[2] <= 8'd0;
     rx_buff0[3] <= 8'd0;
     rx_buff0[4] <= 8'd0;
     rx_buff0[5] <= 8'd0;
     rx_buff0[6] <= 8'd0;
     rx_buff0[7] <= 8'd0;
     rx_buff0[8] <= 8'd0;
     rx_buff0[9] <= 8'd0;

     rx_buff_1_wr_stat <= 1'b1;
     rx_buff_1_wrtn <= 1'b0;
     rx_buff1[0] <= 8'd0;
     rx_buff1[1] <= 8'd0;
     rx_buff1[2] <= 8'd0;
     rx_buff1[3] <= 8'd0;
     rx_buff1[4] <= 8'd0;
     rx_buff1[5] <= 8'd0;
     rx_buff1[6] <= 8'd0;
     rx_buff1[7] <= 8'd0;
     rx_buff1[8] <= 8'd0;
     rx_buff1[9] <= 8'd0;
   end
  else if (rx_buff_0_read)
     begin
       rx_buff0[0] <= 8'd0;
       rx_buff0[1] <= 8'd0;
       rx_buff0[2] <= 8'd0;
       rx_buff0[3] <= 8'd0;
```

```
      rx_buff0[4] <= 8'd0;
      rx_buff0[5] <= 8'd0;
      rx_buff0[6] <= 8'd0;
      rx_buff0[7] <= 8'd0;
      rx_buff0[8] <= 8'd0;
      rx_buff0[9] <= 8'd0;

      rx_buff_0_wr_stat <= 1'b1;
      rx_buff_0_wrtn <= 1'b0;
     end
else if (rx_buff_1_read)
   begin
      rx_buff1[0] <= 8'd0;
      rx_buff1[1] <= 8'd0;
      rx_buff1[2] <= 8'd0;
      rx_buff1[3] <= 8'd0;
      rx_buff1[4] <= 8'd0;
      rx_buff1[5] <= 8'd0;
      rx_buff1[6] <= 8'd0;
      rx_buff1[7] <= 8'd0;
      rx_buff1[8] <= 8'd0;
      rx_buff1[9] <= 8'd0;

      rx_buff_1_wr_stat <= 1'b1;
      rx_buff_1_wrtn <= 1'b0;
     end
else if(acpt_sts)
   begin
      if (rx_buff_0_wr_stat)
          begin
             rx_buff0[0] <= rcvd_msg_id[10:3];
             rx_buff0[1] <= {rcvd_msg_id[2:0], rcvd_rtr, rcvd_dlc};
             rx_buff0[2] <= rcvd_data_frm[63:56];
             rx_buff0[3] <= rcvd_data_frm[55:48];
             rx_buff0[4] <= rcvd_data_frm[47:40];
             rx_buff0[5] <= rcvd_data_frm[39:32];
             rx_buff0[6] <= rcvd_data_frm[31:24];
             rx_buff0[7] <= rcvd_data_frm[23:16];
             rx_buff0[8] <= rcvd_data_frm[15:8];
             rx_buff0[9] <= rcvd_data_frm[7:0];
             rx_buff_0_wrtn <= 1'b1;
             rx_buff_0_wr_stat <= 1'b0;
           end
      else if (rx_buff_1_wr_stat)
          begin
             rx_buff1[0] <= rcvd_msg_id[10:3];
             rx_buff1[1] <= {rcvd_msg_id[2:0], rcvd_rtr, rcvd_dlc};
             rx_buff1[2] <= rcvd_data_frm[63:56];
             rx_buff1[3] <= rcvd_data_frm[55:48];
             rx_buff1[4] <= rcvd_data_frm[47:40];
             rx_buff1[5] <= rcvd_data_frm[39:32];
             rx_buff1[6] <= rcvd_data_frm[31:24];
             rx_buff1[7] <= rcvd_data_frm[23:16];
             rx_buff1[8] <= rcvd_data_frm[15:8];
```

```
            rx_buff1[9] <= rcvd_data_frm[7:0];
            rx_buff_1_wrtn <= 1'b1;
            rx_buff_1_wr_stat <= 1'b0;
          end
     else begin
           rx_buff_1_wr_stat <= rx_buff_1_wr_stat;
           rx_buff_1_wrtn <= rx_buff_1_wrtn;
           rx_buff1[0] <= rx_buff1[0];
           rx_buff1[1] <= rx_buff1[1];
           rx_buff1[2] <= rx_buff1[2];
           rx_buff1[3] <= rx_buff1[3];
           rx_buff1[4] <= rx_buff1[4];
           rx_buff1[5] <= rx_buff1[5];
           rx_buff1[6] <= rx_buff1[6];
           rx_buff1[7] <= rx_buff1[7];
           rx_buff1[8] <= rx_buff1[8];
           rx_buff1[9] <= rx_buff1[9];

           rx_buff_0_wr_stat <= rx_buff_0_wr_stat;
           rx_buff_0_wrtn <= rx_buff_0_wrtn;
           rx_buff0[0] <= rx_buff0[0];
           rx_buff0[1] <= rx_buff0[1];
           rx_buff0[2] <= rx_buff0[2];
           rx_buff0[3] <= rx_buff0[3];
           rx_buff0[4] <= rx_buff0[4];
           rx_buff0[5] <= rx_buff0[5];
           rx_buff0[6] <= rx_buff0[6];
           rx_buff0[7] <= rx_buff0[7];
           rx_buff0[8] <= rx_buff0[8];
           rx_buff0[9] <= rx_buff0[9];
     end
 end
else begin
      rx_buff_1_wr_stat <= rx_buff_1_wr_stat;
      rx_buff_1_wrtn <= rx_buff_1_wrtn;
      rx_buff1[0] <= rx_buff1[0];
      rx_buff1[1] <= rx_buff1[1];
      rx_buff1[2] <= rx_buff1[2];
      rx_buff1[3] <= rx_buff1[3];
      rx_buff1[4] <= rx_buff1[4];
      rx_buff1[5] <= rx_buff1[5];
      rx_buff1[6] <= rx_buff1[6];
      rx_buff1[7] <= rx_buff1[7];
      rx_buff1[8] <= rx_buff1[8];
      rx_buff1[9] <= rx_buff1[9];

      rx_buff_0_wr_stat <= rx_buff_0_wr_stat;
      rx_buff_0_wrtn <= rx_buff_0_wrtn;
      rx_buff0[0] <= rx_buff0[0];
      rx_buff0[1] <= rx_buff0[1];
      rx_buff0[2] <= rx_buff0[2];
      rx_buff0[3] <= rx_buff0[3];
      rx_buff0[4] <= rx_buff0[4];
      rx_buff0[5] <= rx_buff0[5];
```

```
         rx_buff0[6] <= rx_buff0[6];
         rx_buff0[7] <= rx_buff0[7];
         rx_buff0[8] <= rx_buff0[8];
         rx_buff0[9] <= rx_buff0[9];
      end
end

// block to read data from receive bufers
always @ (posedge clk or posedge g_rst)
 begin
   if (g_rst)
     begin
       data_out <= 8'd0;
       rx_buff_0_read <= 1'b0;
       rx_buff_1_read <= 1'b0;
       rx_buff_0_active  <= 1'b0;
       rx_buff_1_active  <= 1'b0;
      end
   else if(read_en)
      begin
        data_out <= 8'd0;
        rx_buff_0_read <= 1'b0;
        rx_buff_1_read <= 1'b0;
        rx_buff_0_active  <= 1'b0;
        rx_buff_1_active  <= 1'b0;
       end
   else if (~read_en)
     begin
      if (rx_buff_0_wrtn && (~rx_buff_1_active))
        begin
          rx_buff_1_read <= 1'b0;
          case (read_count)
             4'd0 :begin
                if (~rx_buff_0_read)
                   begin
                      rx_buff_0_read <= 1'b0;
                      rx_buff_0_active <= 1'b1;
                    end
                else begin
                      rx_buff_0_read <= 1'b0;
                      rx_buff_0_active <= 1'b0;
                   end
              end
             4'd1 :data_out <= rx_buff0[0];
             4'd2 :data_out <= rx_buff0[1];
             4'd3 :data_out <= rx_buff0[2];
             4'd4 :data_out <= rx_buff0[3];
             4'd5 :data_out <= rx_buff0[4];
             4'd6 :data_out <= rx_buff0[5];
             4'd7 :data_out <= rx_buff0[6];
             4'd8 :data_out <= rx_buff0[7];
             4'd9 :data_out <= rx_buff0[8];
             4'd10 :begin
                 data_out <= rx_buff0[9];
```

```
            rx_buff_0_read <= 1'b1;
            rx_buff_0_active <= 1'b0;
            if (rx_buff_1_wrtn)
               rx_buff_1_active  <= 1'b1;
            else rx_buff_1_active  <= 1'b0;
           end
       default :begin
          data_out <= 8'd0;
          rx_buff_0_read <= 1'b0;
          rx_buff_0_active <= 1'b0;
        end
      endcase
  end
else if (rx_buff_1_wrtn && (~rx_buff_0_active))
  begin
    rx_buff_0_read <= 1'b0;
    case (read_count)
       4'd0 :begin
          if (~rx_buff_1_read)
             begin
                rx_buff_1_read <= 1'b0;
                rx_buff_1_active  <= 1'b1;
             end
          else begin
                rx_buff_1_read <= 1'b0;
                rx_buff_1_active <= 1'b0;
             end
         end
       4'd1 :data_out <= rx_buff1[0];
       4'd2 :data_out <= rx_buff1[1];
       4'd3 :data_out <= rx_buff1[2];
       4'd4 :data_out <= rx_buff1[3];
       4'd5 :data_out <= rx_buff1[4];
       4'd6 :data_out <= rx_buff1[5];
       4'd7 :data_out <= rx_buff1[6];
       4'd8 :data_out <= rx_buff1[7];
       4'd9 :data_out <= rx_buff1[8];
       4'd10 :begin
          data_out <= rx_buff1[9];
          rx_buff_1_read <= 1'b1;
          rx_buff_1_active <= 1'b0;
          if (rx_buff_0_wrtn)
             rx_buff_0_active <= 1'b1;
          else rx_buff_0_active  <= 1'b0;
          end
            default:begin
               data_out <= 8'd0;
             rx_buff_1_read <= 1'b0;
             rx_buff_1_active  <= 1'b0;
       end
      endcase
   end
else begin
   data_out  <= 8'd0;
```

```
        rx_buff_0_active <= 1'b0;
        rx_buff_1_active <= 1'b0;
        rx_buff_0_read <= 1'b0;
        rx_buff_1_read <= 1'b0;
       end
    end
  else begin
        data_out  <= 8'd0;
        rx_buff_0_active <= 1'b0;
        rx_buff_1_active <= 1'b0;
        rx_buff_0_read <= 1'b0;
        rx_buff_1_read <= 1'b0;
    end
end

//Block to increment read_count
always @ (posedge clk or posedge g_rst)
   begin
      if (g_rst)
         read_count <= 4'd0;
      else if ((read_count == 4'd10) || ((~rx_buff_1_wrtn)&&
(~rx_buff_0_wrtn)))
         read_count <= 4'd0;
      else if ((rx_buff_0_active) || (rx_buff_1_active))
         read_count<= read_count + 4'd1;
     end
endmodule
```

APPENDIX T

CONTROLLER AREA NETWORK TEST BENCH

```
`timescale 1ns / 1ns
// module to test CAN controllers
module CAN_tst;

reg osc_clk;

reg g_rst;

reg init_err_st;

wire can_bus_in;

wire can_bus_in_0;

wire can_bus_in_1;

wire can_bus_in_2;

wire can_bus_in_3;

reg param_ld;

reg [7:0] data_in_0, data_in_1, data_in_2, data_in_3;

reg tx_buff_ld_0, tx_buff_ld_1, tx_buff_ld_2, tx_buff_ld_3;

reg rd_en_0, rd_en_1, rd_en_2, rd_en_3;

wire tx_buff_busy_0, tx_buff_busy_1, tx_buff_busy_2, tx_buff_busy_3;
wire can_bus_out_0, can_bus_out_1, can_bus_out_2, can_bus_out_3;
wire buff_rdy_0, buff_rdy_1, buff_rdy_2,  buff_rdy_3;
wire [7:0] data_out_0, data_out_1, data_out_2, data_out_3;

assign can_bus_in = (can_bus_out_0 & can_bus_out_1 & can_bus_out_2 &
can_bus_out_3);

assign can_bus_in_0 = can_bus_in;
assign can_bus_in_1 = can_bus_in;
assign can_bus_in_2 = can_bus_in;
assign can_bus_in_3 = can_bus_in;

initial begin
   osc_clk = 1;
   g_rst = 0;
   init_err_st = 0;
   param_ld = 0;

   tx_buff_ld_0 = 0;
   data_in_0 = 0;
```

```
   rd_en_0 = 1;

   tx_buff_ld_1 = 0;
   data_in_1 = 0;
   rd_en_1 = 1;

   tx_buff_ld_2 = 0;
   data_in_2 = 0;
   rd_en_2 = 1;

   tx_buff_ld_3 = 0;
   data_in_3 = 0;
   rd_en_3 = 1;
   #62.5;
end

always #(62.5) osc_clk = ~osc_clk;

always
   begin
         #62.5;
         g_rst = 1;
         #182.5
         g_rst = 0;
         #1000
         param_ld = 1;
         #1000
         data_in_0 = 8'b11111111;
         data_in_1 = 8'b11110111;
         data_in_2 = 8'b11110111;
         data_in_3 = 8'b00111111;
         param_ld = 0;
         #1000
         data_in_0 = 8'b10111111;
         data_in_1 = 8'b11111111;
         data_in_2 = 8'b10111111;
         data_in_3 = 8'b10111111;
         #1000
         data_in_0 = 8'b01111011;
         data_in_1 = 8'b01111101;
         data_in_2 = 8'b01111101;
         data_in_3 = 8'b01111011;
         #1050
         tx_buff_ld_0 = 1;
         tx_buff_ld_1 = 1;
         #1000
         data_in_0 = 8'b11101110;
         data_in_1 = 8'b11110111;
         data_in_2 = 8'b00000000;
         tx_buff_ld_0 = 0;
         tx_buff_ld_1 = 0;
         #1000
         data_in_0 = 8'b11100001;
         data_in_1 = 8'b11100001;
```

```
#1000
data_in_0 = 8'b00110011;
data_in_1 = 8'b00001111;
#1000
data_in_0 = 8'b00000000;
data_in_1 = 8'b00000000;
data_in_3 = 8'b00000000;
#1000
data_in_0 = 8'b00000000;
data_in_1 = 8'b00000000;
data_in_3 = 8'b00000000;
#1000
data_in_0 = 8'b00000000;
data_in_1 = 8'b00000000;
data_in_3 = 8'b00000000;
#1000
data_in_0 = 8'b00000000;
data_in_1 = 8'b00000000;
data_in_3 = 8'b00000000;
#1000
data_in_0 = 8'b00000000;
data_in_1 = 8'b00000000;
data_in_3 = 8'b00000000;
tx_buff_ld_3 = 1;
#1000
data_in_0 = 8'b00000000;
data_in_1 = 8'b00000000;
data_in_3 = 8'b11101110;
tx_buff_ld_3 = 0;
#1000
data_in_0 = 8'b00000000;
data_in_1 = 8'b00000000;
data_in_3 = 8'b11110001;
#1000
tx_buff_ld_2 = 1;
data_in_3 = 8'b00000000;
#1000
data_in_2 = 8'b11110110;
tx_buff_ld_2 = 0;
#1000
data_in_2 = 8'b11110000;
#1000
data_in_2 = 8'b00000000;
#1000
data_in_2 = 8'b00000000;
#1000
data_in_2 = 8'b00000000;
#1000
data_in_2 = 8'b00000000;
#1000
data_in_2 = 8'b00000000;
#1000
data_in_2 = 8'b00000000;
#1000
```

```
        data_in_2 = 8'b00000000;
        #1000
        data_in_2 = 8'b00000000;
        #146400
        rd_en_0 = 0;
        rd_en_1 = 0;
        rd_en_2 = 0;
        rd_en_3 = 0; //Comment out to create overload condition
        # 2000000
        $stop;
    end


CAN can_0 (osc_clk, g_rst, init_err_st, can_bus_in_0, param_ld,
data_in_0, tx_buff_ld_0, rd_en_0, tx_buff_busy_0, can_bus_out_0,
buff_rdy_0, data_out_0);

CAN can_1 (osc_clk, g_rst, init_err_st, can_bus_in_1, param_ld,
data_in_1, tx_buff_ld_1, rd_en_1, tx_buff_busy_1, can_bus_out_1,
buff_rdy_1, data_out_1);

CAN can_2 (osc_clk, g_rst, init_err_st, can_bus_in_2, param_ld,
data_in_2, tx_buff_ld_2, rd_en_2, tx_buff_busy_2, can_bus_out_2,
buff_rdy_2, data_out_2);

CAN can_3 (osc_clk, g_rst, init_err_st, can_bus_in_3, param_ld,
data_in_3, tx_buff_ld_3, rd_en_3, tx_buff_busy_3, can_bus_out_3,
buff_rdy_3, data_out_3);

endmodule
```

APPENDIX U

SYNOPSYS DESIGN CONSTRAINT

```
# Clock = 8 MHz 50% duty cycle
create_clock [ get_ports {osc_clk} ] -period 125.000 -waveform {0 62.5}

# Clock uncertainity of clock period
set_clock_uncertainty -setup 0.500  [ get_clocks {osc_clk} ]

# Clock Latency
set_clock_latency -source 2.000 [ get_clocks {osc_clk} ]

set_clock_latency 1.000 [ get_clocks {osc_clk}]

set_clock_transition 0.2 [ get_clocks {osc_clk}]

# Generated Clock = 1MHz
create_generated_clock -name clk -source [get_ports {osc_clk} ] -
divide_by 8 [ get_pins {/work/CAN/CAN/synchro/clk_cnt_reg[2]/Q} ]

# Clock Latency
#set_clock_latency -source 3.500 [ get_clocks {clk} ]

# Clock uncertainity
set_clock_uncertainty -setup 0.500 [ get_clocks {clk} ]

set_clock_transition 0.2 [ get_clocks {clk}]

# Inputs
# osc_clk  g_rst  init_err_st can_bus_in  param_ld data_in  tx_buff_ld
rd_en

set_driving_cell -lib_cell BUFX8M  -pin Y [ get_ports  {g_rst} ]

set_driving_cell -lib_cell BUFX8M  -pin Y [ get_ports  {can_bus_in} ]

set_driving_cell -lib_cell BUFX8M  -pin Y [ get_ports  {init_err_st} ]

set_driving_cell -lib_cell BUFX8M  -pin Y [ get_ports  {param_ld} ]

set_driving_cell -lib_cell BUFX8M  -pin Y [ get_ports  {tx_buff_ld} ]

set_driving_cell -lib_cell BUFX8M  -pin Y [ get_ports  {rd_en} ]

set_driving_cell -lib_cell BUFX8M  -pin Y [ get_ports  {data_in[*]} ]

set_driving_cell -lib_cell BUFX8M  -pin Y [ get_ports  {osc_clk} ]

set_false_path -from [ get_ports {g_rst}]

set_input_delay  50.00 -clock [ get_clocks {osc_clk} ] [ get_ports
{can_bus_in} ]
```

```
set_input_delay 400.00 -clock [ get_clocks {clk} ] [ get_ports
{init_err_st}]

set_input_delay 400.00 -clock [ get_clocks {clk} ] [ get_ports
{param_ld}]

set_input_delay 400.00 -clock [ get_clocks {clk} ] [ get_ports
{tx_buff_ld}]

set_input_delay 400.00 -clock [ get_clocks {clk} ] [ get_ports {rd_en}]

set_input_delay 400.00 -clock [ get_clocks {clk} ] [ get_ports
{data_in[*]}]

# Outputs
# tx_buff_busy     can_bus_out          buff_rdy        data_out

set_output_delay 100.00 -clock [ get_clocks {clk} ] [ get_ports
{data_out[*]} ]

set_output_delay 100.00 -clock [ get_clocks {clk} ] [ get_ports
{tx_buff_busy} ]

set_output_delay 100.00 -clock [ get_clocks {clk} ] [ get_ports
{buff_rdy}]

set_output_delay 100.00 -clock [ get_clocks {clk} ] [ get_ports
{can_bus_out}]

set_load -pin_load  1 [ get_ports {data_out[*]} ]

set_load -pin_load  1 [ get_ports {can_bus_out} ]

set_load -pin_load  1 [ get_ports {tx_buff_busy} ]

set_load -pin_load  1 [ get_ports {buff_rdy} ]

set_max_transition 0.75 CAN

set_max_fanout 32 CAN
```

APPENDIX V

MAGMA RUN SCRIPT

```
##############Controller Area Network controller Magma flow############

data delete object /

#########################################################################

#importing library volcano

import volcano
/proj0/Sreeram/CAN/magma/volcano/tsmc13g_mvt_6m1t.volcano

set l /tsmc13g_mvt_6m1t

#########################################################################

# Hiding clk buffers and inverters

set clockCellList [list \
                   $l/CLKBUF \
                   $l/CLKINV \
                   $l/CLKBUF_1 \
                   $l/CLKINV_1 \

]

foreach clkcell  $clockCellList {
        force hide $clkcell
}

export volcano ./volcano/hideclkcell.volcano

#########################################################################

# Hiding Cells causing DRC error

set hideCellList [list $l/AO22X_M/AO22XLM $l/AO22X_M_1/AO22XLMTH
$l/NOR3BX_M_1/NOR3BXLMTH $l/NOR3BX_M_1/NOR3BX1MTH]

foreach hidecell $hideCellList {
            force hide $hidecell
            }

export volcano ./volcano/tsmc13g_6mlt.volcano

#########################################################################

#importing rtl
```

```
import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/accp_checker.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/ack_checker.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/bit_destuff.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/bit_monitor.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/bit_stuff.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/can_crc.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/msg_processor.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/dt_rm_frame_gen.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/ovld_err_frm_gen.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/par_ser_conv.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/rx_buff.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/slzd_frm_tx.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/synchronizer.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/tx_buff.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/arbtr_ctrl.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/form_checker.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/crc_checker.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/bit_stuff_monitor.v
```

```
import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/registry.v

import rtl -verbose -verilog -analyze
/proj0/Sreeram/CAN/modelsim/rtl_code/CAN.v \
 -include /proj0/Sreeram/CAN/modelsim/rtl_code/

run rtl elaborate -verbose -verilog CAN

export volcano ./volcano/importrtl.volcano

########################################################################

set m /work/CAN/CAN

########################################################################

#Fixing the rtl

fix rtl $m

export verilog netlist $m /proj0/Sreeram/CAN/netlist/CAN_netlist_frtl.v
-minsize

export volcano ./volcano/frtl.volcano

########################################################################

#Fixing the netlist

fix netlist $m $l

check model $m -print netlist_violations

check model $m -print netlist_violations -file
CAN_netlist_violations.txt

export verilog netlist $m
/proj0/Sreeram/CAN/netlist/CAN_netlist_fnlst.v -minsize

export volcano ./volcano/fnetlist.volcano

########################################################################

#Fix netlist area report

report model $m -file CAN_model_rpt.txt

########################################################################

#Source sdc file

import sdc $m /proj0/Sreeram/CAN/magma/normal/constraints/SDC
```

```
source sdc_translation.tcl

export volcano ./volcano/importsdc.volcano

#########################################################################

#Flatten user specified hierarchy

data flatten $m

#########################################################################

#fixing time

fix time $m $l -effort high

export verilog netlist $m
/proj0/Sreeram/CAN/netlist/CAN_netlist_ftime.v -minsize

#########################################################################

#Before running run bind logical delete all flaoting nets

check model $m -print netlist_violations

check model $m -print netlist_violations -append
CAN_netlist_violations.txt

data delete object $m/BL_ASSIGN_BUF1

data delete object $m/BL_ASSIGN_BUF5

data delete object $m/BL_ASSIGN_BUF3

data delete object $m/BL_ASSIGN_BUF4

data delete object $m/BL_ASSIGN_BUF0

data delete object $m/BL_ASSIGN_BUF2

data delete object $m/BL_ASSIGN_BUF9

data delete object $m/BL_ASSIGN_BUF7

data delete object $m/BL_ASSIGN_BUF11

data delete object $m/BL_ASSIGN_BUF8

data delete object $m/BL_ASSIGN_BUF6

data delete object $m/BL_ASSIGN_BUF10

check model $m -print netlist_violations
```

```
check model $m -print netlist_violations  -append
CAN_netlist_violations.txt

check model $m -print weird_nets

check model $m -print weird_nets -append  CAN_weird_nets.txt

run gate sweep $m

check model $m -print netlist_violations

check model $m -print netlist_violations -append
CAN_netlist_violations.txt

check model $m -print weird_nets

check model $m -print weird_nets -append  CAN_weird_nets.txt

run bind physical $m $l

export volcano ./volcano/ftime.volcano

########################################################################

#Ftime area rpt
report model $m -append CAN_model_rpt.txt

#Ftime timing rpt
config condition case best

report timing summary $m -file CAN_timing_rpt.txt

config condition case worst

report timing summary $m -append CAN_timing_rpt.txt

########################################################################

# Area report for CAN

report area $m

report area $m -file CAN_area_rpt.txt

########################################################################

#floorplan
force model routing layer $m highest METAL6

data create floorplan $m fplan

force floorplan parameters $m/floorplan:fplan -width 325u -height 325u
-left_margin 21u \
-right_margin 21u -bottom_margin 21u -top_margin 21u -row_height 2.87u
```

```
run floorplan size $m/floorplan:fplan -target_util 0.60 -aspect_ratio 1
force plan net VDD $m -usage power -port VDD
force plan net VSS  $m -usage ground -port VSS
run floorplan apply $m
run bind physical $m $l
force plan pin $m/can_bus_in METAL4 1u 2u -side bottom
force plan pin $m/rd_en METAL4 1u 2u -side bottom
force plan pin $m/osc_clk METAL4 1u 2u -side top
force plan pin $m/g_rst METAL4 1u 2u -side top
force plan pin $m/init_err_st METAL4 1u 2u -side top
force plan pin $m/param_ld METAL4 1u 2u -side top
force plan pin $m/tx_buff_ld METAL4 1u 2u -side top
force plan pin $m/data_in[0] METAL3 1u 2u -side left
force plan pin $m/data_in[1] METAL3 1u 2u -side left
force plan pin $m/data_in[2] METAL3 1u 2u -side left
force plan pin $m/data_in[3] METAL3 1u 2u -side left
force plan pin $m/data_in[4] METAL3 1u 2u -side left
force plan pin $m/data_in[5] METAL3 1u 2u -side left
force plan pin $m/data_in[6] METAL3 1u 2u -side left
force plan pin $m/data_in[7] METAL3 1u 2u -side left
force plan pin $m/can_bus_out METAL4 1u 2u -side bottom
force plan pin $m/buff_rdy METAL4 1u 2u -side bottom
force plan pin $m/tx_buff_busy METAL4 1u 2u -side bottom
force plan pin $m/data_out[0] METAL3 1u 2u -side right
force plan pin $m/data_out[1] METAL3 1u 2u -side right
force plan pin $m/data_out[2] METAL3 1u 2u -side right
force plan pin $m/data_out[3] METAL3 1u 2u -side right
```

```
force plan pin $m/data_out[4] METAL3 1u 2u -side right

force plan pin $m/data_out[5] METAL3 1u 2u -side right

force plan pin $m/data_out[6] METAL3 1u 2u -side right

force plan pin $m/data_out[7] METAL3 1u 2u -side right

run plan create pin $m

export volcano ./volcano/fplan.volcano

########################################################################

check model $m -level floorplan -file CAN_fplan_rpt.txt

########################################################################

#powerplan

force route power2 ring $m core_ring -ring {VDD { METAL5 6u 3u
horizontal } { METAL6 6u\
3u vertical }} -ring {VSS { METAL5 6u 12u horizontal } { METAL6 6u 12u
vertical }}

run route power2 ring $m -inner_shape -style separate -specs {
core_ring }  -allow_drc on

force route power2 mesh $m core_mesh  -orientation vertical  -wire {
VDD METAL4 4u 10u \
-extend both } -wire { VSS METAL4 4u 24u -extend both }

run route power2 mesh $m -inner_shape -specs {core_mesh} -group_count 5

run route power2 rail $m

run route power2 pin $m

run route power2 via $m -nets "VDD VSS"

export volcano ./volcano/powerplan.volcano

########################################################################

check model $m -level floorplan -append CAN_fplan_rpt.txt

########################################################################

#fix cell

fix cell $m $l
```

```
export verilog netlist $m
/proj0/Sreeram/CAN/netlist/CAN_netlist_fcell.v

check route drc $m -power_only

export volcano ./volcano/fcell.volcano

########################################################################

#Fcell area report
report model $m -append CAN_model_rpt.txt

#Fcell timing Report
config condition case best

report timing summary $m -append CAN_timing_rpt.txt

config condition case worst

report timing summary $m -append CAN_timing_rpt.txt

########################################################################

check model $m -print weird_nets

########################################################################

#fix clock

foreach {clockCell} $clockCellList {
       clear  hide $clockCell
     }

force plan clock $m -buffer "$l/CLKBUF/CLKBUF_SUPER
$l/CLKBUF_1/CLKBUF_1_SUPER $l/DLY/DLY_SUPER $l/DLY_1/DLY_1_SUPER" -
inverter "$l/CLKINV/CLKINV_SUPER  $l/CLKINV_1/CLKINV_1_SUPER"

fix clock $m $l -weight skew

export verilog netlist $m /proj0/Sreeram/CAN/netlist/CAN_netlist_fclk.v

export volcano ./volcano/fclock.volcano

########################################################################

#Fclock area report
report model $m -append CAN_model_rpt.txt

#Fclock timing report
config condition case best

report timing summary $m -append CAN_timing_rpt.txt

config condition case worst
```

```
report timing summary $m -append CAN_timing_rpt.txt

##########################################################################

#fix wire

fix wire $m $l -effort high

export verilog netlist $m
/proj0/Sreeram/CAN/netlist/CAN_netlist_fwire.v

##########################################################################

#Add filler cells

config gate filler $l -ordering decreasing \
$l/FILL1M/FILL1M $l/FILL1MTH/FILL1MTH \
$l/FILL2M/FILL2M $l/FILL2MTH/FILL2MTH \
$l/FILL4M/FILL4M $l/FILL4MTH/FILL4MTH \
$l/FILL8M/FILL8M $l/FILL8MTH/FILL8MTH \
$l/FILL16M/FILL16M $l/FILL16MTH/FILL16MTH \
$l/FILL32M/FILL32M $l/FILL32MTH/FILL32MTH

run gate filler add $m -allow_drc off

##########################################################################

#Fwire area report
report model $m -append CAN_model_rpt.txt

#Fwire timing report
config condition case best

report timing summary $m  -append CAN_timing_rpt.txt

config condition case worst

report timing summary $m  -append CAN_timing_rpt.txt

##########################################################################

#Final area report
report model $m -append CAN_model_rpt.txt

#Final timing Report
config condition case best

report timing summary $m -append CAN_timing_rpt.txt

config condition case worst

report timing summary $m -append CAN_timing_rpt.txt
```

```
##########################################################################

export verilog netlist $m
/proj0/Sreeram/CAN/netlist/CAN_netlist_final.v

##########################################################################

# Exporting the SPEF
export spef $m CAN.spef

##########################################################################

# Exporting the GDS
export gdsii $m CAN.gds

##########################################################################

#Final area report
report model $m

#Final timing Report
config condition case best

report timing summary $m

config condition case worst

report timing summary $m

##########################################################################
```

APPENDIX W

VERPLEX DO FILE

```
reset

set dofile abort exit

set log file CAN.log -replace

read library /proj0/Sreeram/CAN/src_lib/tsmc13_m.v -verilog

read library /proj0/Sreeram/CAN/src_lib/tsmc13hvt_m.v -verilog -append

set undefined cell black_box -noascend

set undriven signal 0 -both

read design /proj0/Sreeram/CAN/modelsim/rtl_code/CAN_total.v -verilog -
golden

report rule check -verbose > CAN_design_rule_check.rpt

read design /proj0/Sreeram/CAN/magma/normal/netlist/CAN_netlist_final.v
-verilog -revised

set mapping method -name first -unreach -nosensitive -nets -
nobbox_name_match

set flatten model -nodff_to_dlat_feedback -seq_redundant

set system mode lec

add compare points -all

compare

report compare data -noneq

report statistics > CAN_equivalent.rpt

report floating signals -Undriven -Both -All > CAN_floating.rpt

report unmapped points > CAN_unmapped.rpt

report black box -class Full -Both -Module -Hier -Hidden >
CAN_blackbox.rpt

report compare data -noneq > CAN_nonequivalent.rpt

save session lec_session/CAN_verplex
```

PERMISSION TO COPY

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Texas Tech University or Texas Tech University Health Sciences Center, I agree that the Library and my major department shall make it freely available for research purposes. Permission to copy this thesis for scholarly purposes may be granted by the Director of the Library or my major professor. It is understood that any copying or publication of this thesis for financial gain shall not be allowed without my further written permission and that any user may be liable for copyright infringement.

Agree (Permission is granted.)

Sreeram Krishnamoorthy — 27th July 2006

Student Signature — Date

Disagree (Permission is not granted.)

_____ _____

Student Signature — Date