A BITWISE SIMULATION OF THE CONTROLLER AREA NETWORK

By

NATARAJAN S. PENNATHUR

Bachelor of Engineering

University of Mysore

Mandya, India

1990

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1993

OKLAHOMA STATE UNIVERSITY

A BITWISE SIMULATION OF THE

CONTROLLER AREA NETWORK

Thesis Approved:

Thesis Adviser

Mitchell ? Neilsen

Thomas C. Collins

Dean of the Graduate College

ACKNOWLEDGMENTS

I sincerely thank Dr. K. M. George for his able guidance throughout the course of my research work. He has contributed in a great way by making valuable suggestions at critical junctures. Also, it would have not been possible for me to fulfill this research objective without the efforts of Dr. Mitch Neilsen. He was a great source of inspiration during the various phases of my work. He was actively involved in the design, implementation, and analysis of the simulation. I am also thankful to him for meticulously evaluating my thesis report. I also wish to sincerely thank Dr. Marvin Stone for providing me with all the technical details concerning the CAN. Every discussion with him has been a pleasurable experience, and a path to further knowledge. It has been a very exciting experience to have worked with him.

I am deeply indebted to my beloved parents who have provided moral support in all my endeavors. I am grateful to have received their blessings at all times. I have also drawn lots of inspiration from friends and family alike. I wish to make special mention of some close friends and colleagues including Ganesh, Raja, Prasad, John, Jay, Sam, J.D, Kam, Will, Aaron, Daphna, Kim, and Chitti for their constant encouragement. Finally, I wish to thank my stars for having achieved this objective.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Controller area network (CAN) is a real-time serial communication network that is presently being used for in-vehicle networking. In-vehicle communication is used in cars, agricultural trucks, military and construction vehicles, industrial and factory automation communications, and other event control and information sharing systems. The value for such in-vehicle serial communications is to reduce the harness size, manufacturing and maintenance complexity, eliminate sensors, increase diagnosability, and facilitate in-vehicle electronic options. The effort to develop a common protocol standard has been perceived by Robert Bosch GmbH and Intel Corporation. A similar network model is also being researched by Philips, Chrysler, and other automotive companies. Most of the work has been published in a series of papers in the Society of Automotive Engineers (SAE). CAN features include an open system to expand the network without topological changes, high reliability, low cost, minimum CPU burden for communication, maximum transparency, data consistency, and speedy transmission for real-time applications.

The need for an efficient and low cost network for in-vehicle communication has created a wide number of research areas. The necessity to standardize such a network has become essential. One of the primary research areas now is to find efficient protocols over the existing hardware to shape the network into the OSI seven layer reference model. Since CAN is a typical real-time network with its own way of handling collisions, priority arbitration, addressing, and error control, the typical network algorithms do not apply very well for this kind of network. The architectural details, standards, and protocols of the CAN are discussed in Chapter II.

A major concern is the performance of such a network under heavy load. This means that as the message rate increases, the throughput and delay should remain stable. The messages may be either periodic or sporadic. The scheduling of such messages is tricky in a real-time network system, where deadlines need to be met. A thorough investigation of various real-time scheduling algorithms is discussed in Chapter III.

The purpose of this research is to find an efficient way of handling the periodic and sporadic message set that the CAN application presents. The objective is also to schedule messages within the network to reduce transmission delay, and hence achieve the much desired higher network throughput. At present, the network is designed to operate under less than a 30% load. A comprehensive CAN simulation model has been developed to test and analyze the network performance. The distinctive feature of the simulation program is its bitwise trace of the CAN protocols. Also, functions of error management and fault confinement have been included to analyze message error and node failure overheads. The implementation details of the simulation, and the performance evaluation are discussed in Chapter IV. Finally, the thesis concludes with a summary and a brief discussion of future research in Chapter V.

CHAPTER II

OVERVIEW OF THE CONTROLLER AREA NETWORK

Why CAN ?

The following are some of the standard network topologies, and their limitations that make them unsuitable for real-time applications [Phai86].

1. The *star network* topology has a central node, to which are connected several nodes in a star. This arrangement offers waterproof arbitration schemes, but the failure of the central node results in network failure.
2. The *token bus* is another topology that has good configuration flexibility. However, the network does not offer multimastership. The token is held by a single node at a time, and only that node is allowed to transmit messages. The failure of the node holding the token results in a substantial time loss. Recovery from failure requires complex logic.
3. The *token ring* network is similar to the token bus with the difference being in the physical rather than the logical ring structure. These networks are suited for high speed data transfer with token mastership, and priority based access to tokens. Again, the probability of a ring failure is a major drawback.
4. The *bus* topology using Carrier Sense Multiple Access with Collision Detection (*CSMA/CD)* protocols offers multimastership by allowing any node to transmit when the bus is idle. The drawback in these networks is the destruction of messages when a collision occurs, and the retransmission of messages that involves substantial time loss and increased recovery logic.

The CAN has the following properties that make it most suitable for a real-time network.

1. Prioritized bitwise arbitration for fast transmission of high priority messages with a latency time as short as 150 microseconds.
2. Guarantee of latency times.
3. High transmission rates in the range of 1 Mbps for a bus length of 40m.
4. An open system that allows configuration flexibility to add or delete any number of nodes without changing the underlying software or hardware of any node, with the constraints being physical limitations, and electrical load on the bus.
5. Multicast reception with time synchronization enables any number of nodes to receive a message.
6. Object-oriented communication that increases hardware transparency and system wide data consistency.
7. Multimastership that allows any node to start transmission when the bus is free.
8. A powerful error handling and signaling mechanism by means of bus monitoring, cyclic redundancy checks, bit stuffing, and message frame checks.
9. Automatic retransmission of corrupted messages as soon as the bus is idle again.
10. A distinction between temporary and permanent failure of nodes, and autonomous switching off of defective nodes.

CAN Hardware

The CAN architecture is based on a multimaster single bi-directional bus topology as shown in Figure 1. A node forms the point of contact with the communication channel, while the station is where the sensor and its microprocessor resides. All nodes are linked to the station via a communication controller. The stations may be data acquisition sensors or computers. A block implementation of a CAN station is as shown in Figure 2. The I/O devices receive data from acquisition sensors and the station CPU processes the

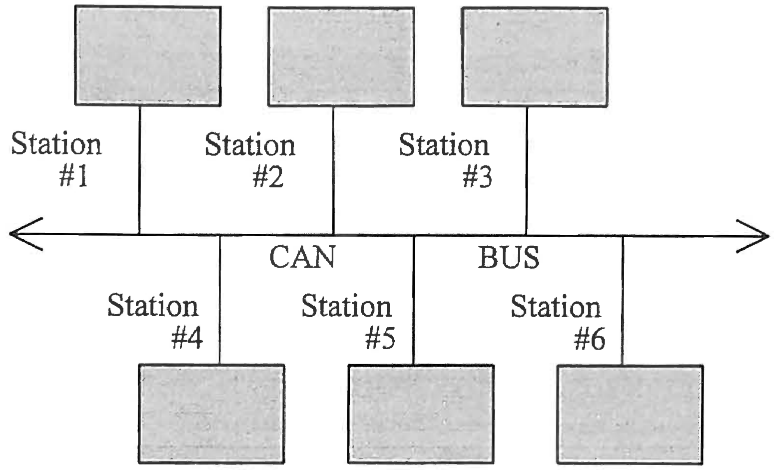


Figure 1. CAN Topology

data. The processed data is stored as communication objects within the shared RAM. The bus interface initiates a message transfer when it senses the bus to be idle. Similarly, a message is received by the bus interface by matching an appropriate communication object within the RAM. Once the data is copied into the RAM, the CPU processes it, and initiates an I/O transfer to the sensor.

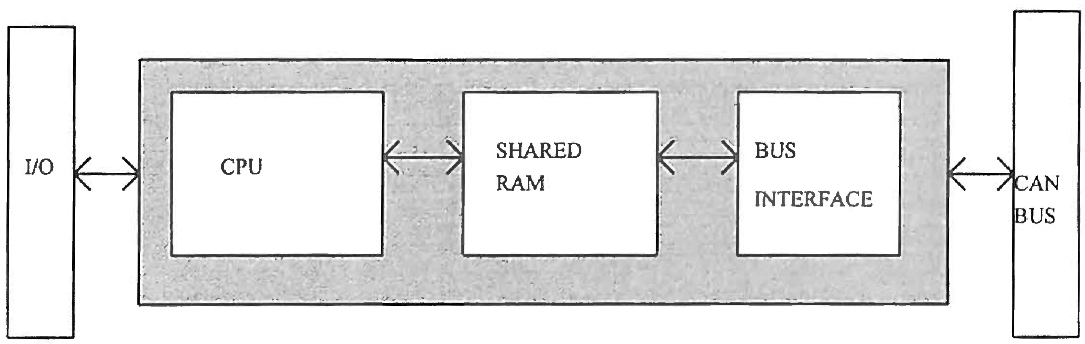


Figure 2. CAN Functional Diagram (Source [Arne87])

The CAN bus is a single bi-directional channel that may have a single wire, two differential wires, or optical fibers with T-junctions. The bus can have two logical values, termed dominant (logical 0) and recessive (logical 1). The recessive bit is represented by a mean voltage level of two voltages, V_{CAN_H} and V_{CAN_L} that are defined with respect to the ground voltages of the electronic control unit (ECU). During the recessive state they are fixed to a mean voltage level. A recessive bit is transmitted during an idle state condition. The dominant state is a differential voltage greater than a minimum threshold, and overwrites the recessive state during arbitration [Bosc91].

The main components of the communication controller include a dual port RAM (DP-RAM), an interface management processor (IMP), and a processor interface unit (PIU). Other components include a bus timing logic (BTL), a transceive logic (TCL), an error management logic (EML), a bit stream processor (BSP), and a clock generator (CG). A block diagram representation is as shown below in Figure 3.

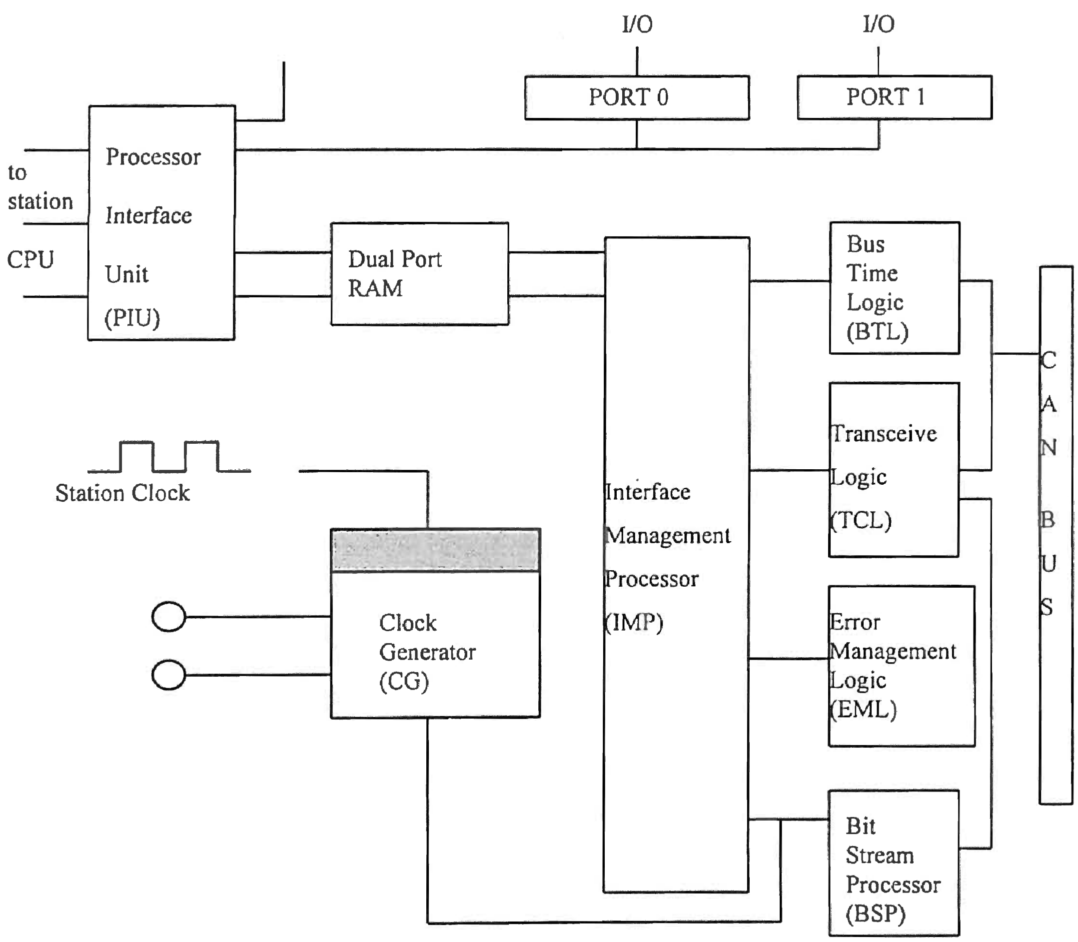


Figure 3. Block Diagram of the CAN Architecture (Source [Phai88])

The DP RAM forms a communication buffer between the station microprocessor and the IMP. Messages are stored as communication objects in the DP RAM. Each communication object consists of an identifier, a control segment, and a data segment. It has a global status register and a control register that help create communication objects to be used by the IMP. The IMP controls the transmission and reception of data between the serial bus and the DP RAM. It performs these tasks by means of acceptance and transmission filtering. This is done by scanning the communication objects in the DP RAM through its data paths. It computes the address for a communication buffer access and manipulates the appropriate control bits to execute the CPU's receive and transmit commands.

The PIU links the DP RAM to the station CPU. It consists of an 8-bit multiplexed data/address bus, read/write control, address latch enable, chip select, interrupt output, external interrupt input, reset, ready output signal, two 8-bit output ports 0 and 1, and 3 chip select output lines to connect additional peripheral devices. The PIU connections to the host microcontrollers is discussed in more detail in [Phai88].

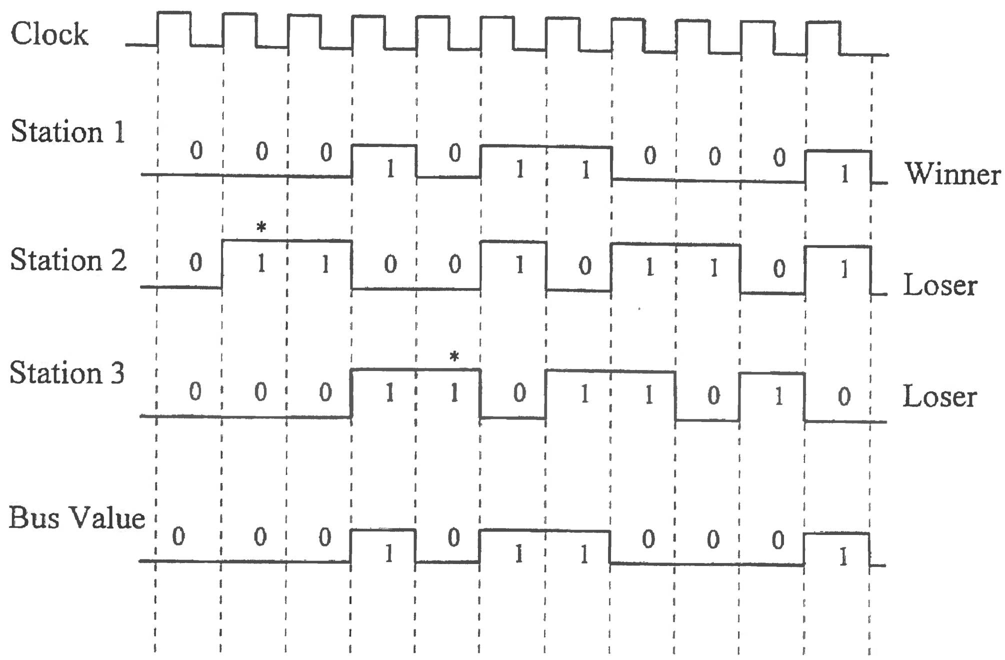
The bus timing logic (BTL) synchronizes the station clock with the signal clock on the bus using a comparator. It also provides programmable time segments to compensate for the propagation delays and phase shifts. The transceive logic (TCL) performs bit stuffing and Cyclic Redundancy Check (CRC) sequence generation using an output driver and several shift registers. The bit stream processor (BSP) controls the flow of bits between the parallel IMP interface and the serial CAN bus interface. It performs bit reception, bitwise arbitration, bit transmission, error signaling and control of TCL. The error management logic (EML) gets error signals from the BSP, and takes action by signaling the BSP, the TCL, and the IMP of error statistics. The clock generator (CG) has an oscillator, a clock divider register, and a driver circuit. The oscillator is driven by an external crystal, or in case of low baud rates by a ceramic resonator. The clock's output is programmable [Phai88].

CAN Standards and Protocols

The characteristic features of the CAN includes its layered structure and physical properties. The CAN implements a serial communication protocol with three well defined layers. The protocol description follows from the layered structure according to the ISO/OSI reference model.

The physical layer performs bit level functions of decoding / encoding, synchronization, timing, high voltage protection, and drive capability. The upper layer being the data link layer is sub-divided into two sub layers, namely the medium access control (MAC) sublayer, and the logical link layer (LLC). The MAC sublayer performs message level functions of fault confinement, message validation, error detection and signaling, acknowledgment, message framing, transfer rate, timing, data encapsulation / decapsulation, serialization / deserialization and arbitration. The LLC sublayer performs object level functions of prioritized message handling, message buffering, overload notification, and recovery management. The almost non existent application layer has controller level functions such as data collection through sensors, request for data from other sensors, and sending messages across the network.

A *bus arbitration protocol* is used as a means of resolving collisions by consensus rather than a central arbiter making decisions. The time required to resolve a conflict is bounded by the number of arbitration bits used. The arbitration is shown by means of square wave forms, where each cycle represents a bit level as seen below in Figure 4. The CAN bus can be viewed as an OR gate whose value is monitored by all nodes connected to the bus. If one can violate the Boolean rule, and assume that a *0* when ORed with a *1* results in a *0*, then the protocol is easily understood. Since every station is synchronized to read the same bit field, whenever a station detects a dominant bus level of *0*, while it actually sent a recessive bit *1*, the station backs off, and thus loses the arbitration. Eventually when all the arbitration bits are sent the winner holds the bus, as the case with station #1 in Figure 4. Hence the arbitration results in the message with the



* represents the point at which the station loses the arbitration

Figure 4. Collision Resolution by Non-Destructive Bitwise Arbitration

highest priority (lowest binary value) winning.

Communication modes are of five types, namely command, request, proprietary, sleep / wakeup, and acknowledgment [Bosc92]. Command mode provides the capability to send commands to nodes to take necessary actions. Addressing a destination may be explicit with a destination address, or implicit with an extended data content. The request mode facilitates information request globally from all nodes, or from a specific destination. This mode provides messages to be sent to devices that can distinguish them properly without conflicts. The source address field of the message may have the sender's address when transmitting a message, or the receiver's address when the message is a destination specific request. The acknowledgment mode provides for a positive acknowledgment (ACK) for an error free message transfer, or a negative acknowledgment (NACK) for an erratic message transfer that results in an automatic retransmission. A sleep mode enables the CAN device to be in an inactive state, reducing power consumption as the bus drivers are disconnected. The internal activity gets restarted by a wake-up signal.

Message transfer for the CAN 2.0 version provides an extended frame in addition to the standard frame defined in the CAN 1.0/1.2 version. Both a standard message format with a 11 bit identifier, and an extended frame format with 29 bits have been incorporated in the CAN 2.0 version. This is to make the CAN 2.0 version compatible with the CAN 1.0/1.2 versions. The extended frame format allows the CAN to address a large implicit data content address. This way CAN performs functional addressing using the data content rather than the physical address itself [Phai86].

CAN performs message passing using communication objects. Information from sensors is written into the data segment of the proper communication object within the DP RAM. A transfer is initiated by a transmission request in the control segment. Transmission and error handling is then performed without the CPU involvement. This helps to fire and forget messages [Kien86]. Message reception is performed by reading the data segment onto an already set up communication object. There are four kinds of frames in the CAN namely, a data frame that carries data from transmitters to receivers, a remote frame to request the transmission of a data frame with the same identifier, an error frame to signal a bus error, and an overload frame to provide an extra delay between succeeding data or remote frames. Data and remote frames may be used in both standard as well as extended frame formats.

A data frame is composed of seven fields: START OF FRAME (SOF), ARBITRATION (ARB), CONTROL (CTR), DATA, CRC, ACK, and END OF FRAME (EOF) as shown in Figure 5. The SOF field consists of a single dominant bit to mark the beginning of the message frame. The ARB field for the standard frame format has an 11 bit identifier, and a Remote Transmission Request (RTR) bit. The extended format ARB field has a 29 bit identifier, a Substitute Remote Request (SRR) bit, an Identifier Extension (IDE) bit, and an RTR bit. In both formats the first 11 bits represent the base id, ID1, that defines the base priority of the message, while the 18 additional bits forming the extended id, ID2, in the extended format represent data content implicitly. In both

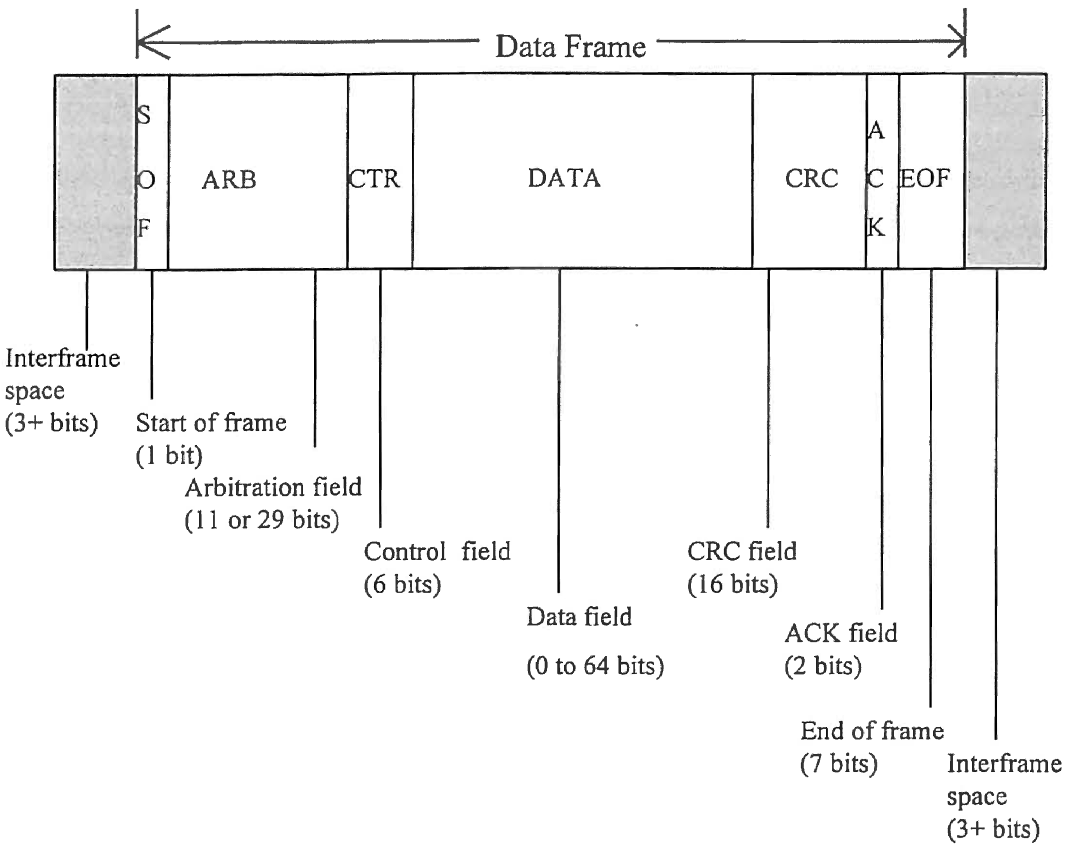


Figure 5. Data Frame Format (Source [Bosc91])

formats, the RTR bit is dominant for a data frame, while it is recessive for a remote frame. This bit notifies the network that the message is a remote request. The SRR bit is placed in the RTR bit field position in the extended frame, and is recessive to ensure that the standard frame prevails over the extended frame in the event of a collision, when the base identifiers of these dissimilar frames is the same. This bit tells the network that the message is in an extended frame format. The CTR field has six bits. For the standard format it has an IDE bit, a reserved bit r0, and a four bit data length code (DLC). The IDE bit is in the control field for standard format and is dominant, while it is recessive in the extended format. The DLC represents the length of the data bytes in binary. The standard frame format for the ARB and the CTR fields is as shown in Figure 6.

In the extended format two reserved bits r0, and r1 are followed by a four bit DLC. Both the reserved bits are sent dominant in an extended frame. The ARB and CTR fields

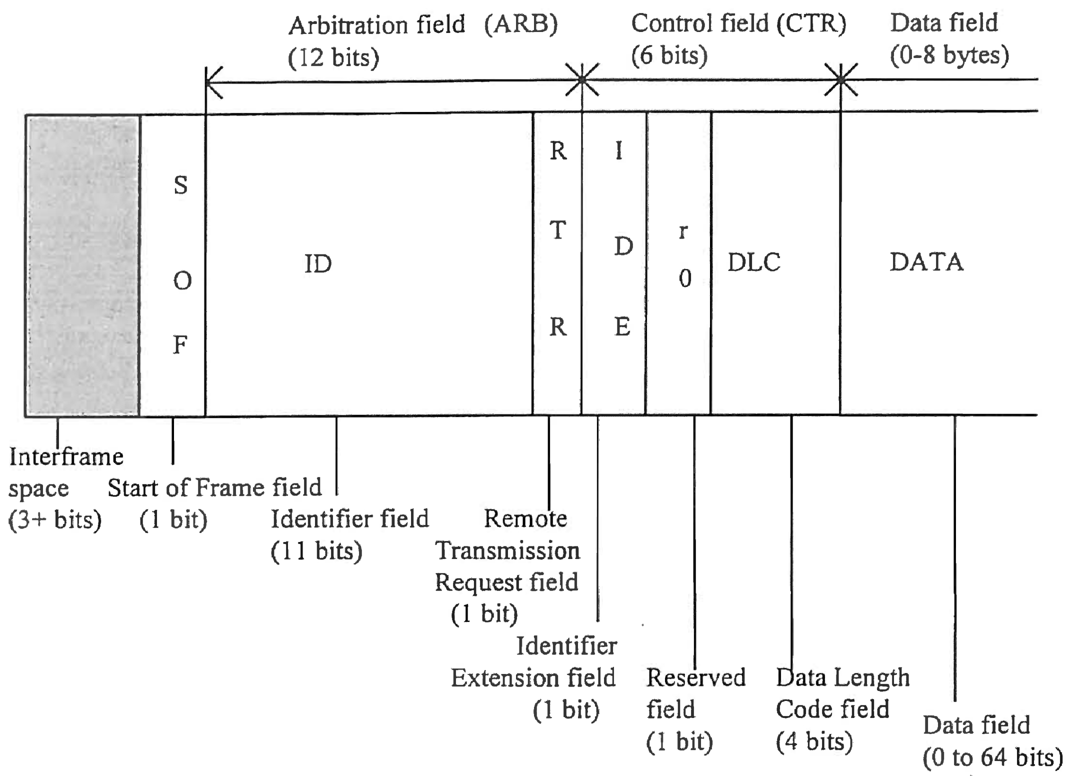


Figure 6. Standard Format for ARB and CTR Fields (Source [Bosc91])

for an extended frame are as shown in Figure 7. The DATA field has 0 to 8 bytes of data that are transferred MSB first. The CRC field has 16 bits, containing a 15 bit CRC sequence followed by a CRC delimiter bit that is recessive. The ACK field is two bits long, and contains the ACK slot, and the ACK delimiter that is recessive. A positive acknowledgment of reception of data is reported by super scribing the recessive ACK slot bit with a dominant bit by the receiving stations. Finally a seven bit EOF field is used to mark the end of the message frame. All seven bits are recessive.

A remote frame is used by a receiver to initiate the transmission of data to the source node. A remote frame contains the address of the transmitter. It is void of the DATA field. The RTR bit is set to recessive, to indicate a remote transmission request.

An error frame has two fields consisting of a six equal bit ERROR FLAG that is a superposition of error flags contributed by various stations, and an eight bit ERROR

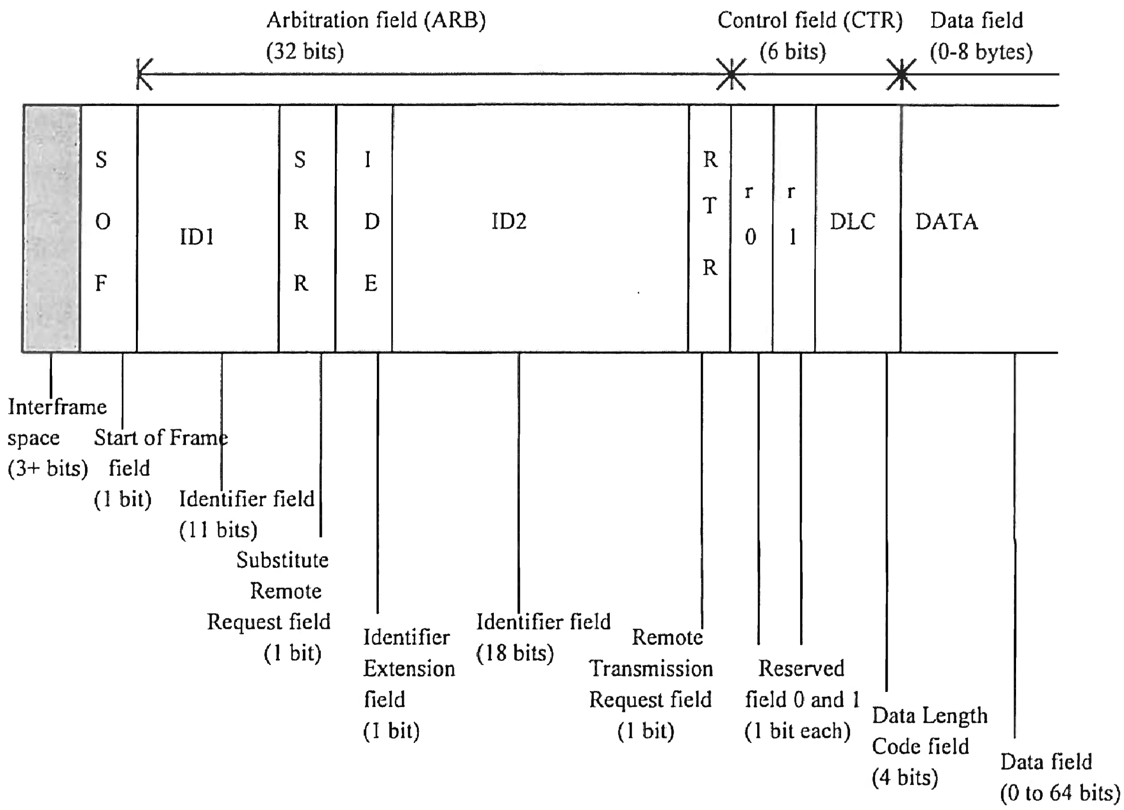


Figure 7. Extended Format for ARB and CTR Fields (Source [Bosc91])

delimiter that are all recessive. All active nodes send an ACTIVE ERROR FLAG that consists of six dominant bits, while the passive nodes send a PASSIVE ERROR FLAG that consists of six recessive bits. An ERROR FLAG violates the bit stuffing rule, and hence all other nodes on the bus detect an error condition, and in turn signal errors.

An overload frame has two fields consisting of six OVERLOAD FLAG bits that are dominant, and eight OVERLOAD delimiter bits that are all recessive. An overload condition may occur when the delay of the next data or remote frame falls short of the interframe space, or when a dominant bit is detected at the first, and second bit of intermission, or when a dominant bit is detected at the eighth (last) bit of an error frame or an overload frame.

An interframe space has two fields namely, a three bit INTERMISSION field in which all bits are recessive, followed by an arbitrary number of bits in the BUS IDLE field. In addition to the above, an error passive station that was a transmitter of the last message has an eight bit SUSPEND TRANSMISSION field following the INTERMISSION field in which all bits are recessive. The overload and error frames are not preceded by a interframe space. Any dominant bit detected during the BUS IDLE period is interpreted as a SOF of a new message.

Message Filtering is used by a station to receive a message that belongs to it, and hence implement a multicast network. This is achieved by having optional mask registers that allow any identifier bit to be set 'don't care' for message filtering, and may be used to select a group of identifiers to be mapped into the attached receive buffers. The mask registers may be programmed, to be enabled or disabled for message filtering. The length of the mask register can comprise the whole identifier or only part of it.

Every node on the network checks the message identifier on the bus to see if it matches with the object identifier in the DP RAM. If a match occurs, the message is copied into the proper communication object in the DP RAM.

Message validation is performed by both the transmitter and the receivers of the message. A message is valid for the transmitter if it does not detect an error at the end of the EOF bits. The message is valid for a receiver, if no error is detected until the penultimate bit of EOF is received. Corrupted messages result in automatic retransmission.

Error detection and signaling is performed by the error management logic (EML) that is connected to the bus. All global errors, local errors, 5 randomly distributed errors in a message, burst errors of length less than 15 in a message, and errors of any odd number in a message are detected [Gupt88]. The total residual error probability for undetected corrupted messages is less than the message error rate which is $(4.7 * 10^{-11})$ [Bosc91]. The message recovery time after detection of an error is about 29 bit times. Five different types of errors are detected, namely bit errors, stuff errors, CRC errors, form errors, and acknowledgment errors.

A bit error is detected if a transmitter detects a bus value that is different from the bit value it sent. Bit stuffing, ACK flagging, and overwriting of passive error flags are exceptions to the rule.

A stuff error is detected when there are six consecutive equal bits that violate the law of bit stuffing for a CAN. The exceptions to the rule are the ERROR FLAGS, and OVERLOAD FLAGS that send six consecutive dominant or recessive bits.

A CRC error occurs when the CRC sequence computed by the receiver does not match the sequence sent by the transmitter.

A form error is detected when a fixed form bit field has an illegal bit. For example, if the SOF bit is received as a recessive bit, a form error occurs.

An acknowledgment error is detected by a transmitter, when it does not read a positive acknowledgment in the form of a dominant bit in the ACK slot field.

Fault confinement is implemented by having two error counts, namely a TRANSMIT ERROR COUNT, and a RECEIVER ERROR COUNT at each node. Initially, all nodes start out as active nodes with zero error counts. When a transmitter or a receiver detects an error its corresponding error count is incremented by one. If a transmitter, or a receiver detects an error condition during transmission of an error flag, the corresponding error count is incremented by eight. Successful transmission, and reception of a message results in decrementing the corresponding error count by one. If either of a node's error count exceeds 127, it becomes an error passive node. Similarly if both the node's error counts become less than 128, then it becomes an error active node. An error active node signals errors with an ACTIVE ERROR FLAG consisting of dominant bits, while an error passive node uses a PASSIVE ERROR FLAG consisting of recessive bits. An error active node is hence used as a better judge of an error occurrence, while an error passive node's error signals may be overridden. If the TRANSMIT ERROR COUNT of a node exceeds 255, it becomes bus off. A bus off node is inactive on the bus. This feature enables the CAN to isolate a faulty node.

Bit timings for the nominal bit time is divided into separate non-overlapping time segments namely, synchronization segment (SYN_SEG) that is 1 time quanta long, propagation segment (PROP_SEG) that is 1 to 8 time quanta long, phase buffer segment 1 (PHASE_SEG1) that is 1 to 8 time quanta long, and phase buffer segment 2 (PHASE_SEG2) that is the maximum of PHASE_SEG1 and information processing time [Bosc91]. The information processing is less than or equal to 2 time quanta long. The total time quanta in a bit time is programmable to between 8 to 25. Synchronization is achieved by hard synchronization and resynchronization that are described in [Bosc91].

CAN Enterprise

CAN networks have been on the scene since the need for an electronic network for the highly competitive automotive industry was required. Also, CAN provides a real-time and multimaster support with nondestructive collision resolution. The American Trucking Association (ATA), the Society of Automotive Engineers (SAE), and the International Standards Organization (ISO), along with various automotive and semiconductor manufacturers worked toward developing an in-vehicle network. The CAN components, like the Intel 82526, have been on the market since 1988. Many automobile corporations like Chrysler, and Robert Bosch GmbH have been perceiving the design and implementation of such real-time distributed systems for their cars. Also, much interest is being generated in the aviation and earth moving equipment industries. An Inter Controller Area Network (ICAN) was proposed in SAE J1583 by the Intel Corp. Intel's 82526 integrated the IMP, DP RAM and PIU units into one single chip. Chrysler

Corp. came up with their Chrysler Collision Detection (CCD) for serial data communication multiplex bus. In 1985, Robert Bosch GmbH and Intel joined together to develop an in-vehicle network device with CAN specifications [Iver88]. Philips built various components to support testing and design of CANs [Eyho89]. Motorola developed a single chip microcontroller MC68HC04 for a basic CAN architecture [Jord88]. The difference in its implementation was that communication between the CPU and the CAN interface is via a dual register with a context switch. This has a limitation in that it can receive a small number of messages at the full data rate.

Also, since an onboard CAN simulation package exists, efforts can be made to test protocols on the CAN hardware itself. Also, Philips provides a NetSim PC-based simulator to which a CAN network must be described in terms of number of nodes, transmission speed, message identifiers, message length, and a noise margin. The output provides results of the simulation, such as network delay, network throughput, and bus load. Robert Bosch GmbH has provided an on board simulator, with which some specifications and performance measures can be obtained. All of these are presented in their draft of J1939 in the SAE Recommended Practices. The CAN 2.0 high speed proposal for an International Draft Standard, (September, 1991) focuses mainly on the CAN's data link layer and its differences with previous versions.

At Oklahoma State University, research was perceived by Dr. Marvin Stone and Dr. Huizhu Lu's student Mr. Zhengou Wang on a priority exchange algorithm to schedule sporadic message generations with a maximum arrival rate. The assumption made here was that the arrival rates of messages are Poisson. The message priority assignment algorithm as they called it made a worst case analysis by considering the transmission time and an allowed transmission delay for each message type as the parameter to assign priorities. Priorities were exchanged as and when the service time of a message exceeded the allowed transmission delay of that message from the time it arrived.

CHAPTER III

REAL-TIME ENVIRONMENT

Concepts

Real-time computing implies the use of a computer in conjunction with an external process. The concept of a real-time system is more specifically defined as the ability of a computer to respond to stimuli from an external event in a timely fashion. The computer needs to be fast enough to complete the execution of the process. In a real-time network this translates to the speed of communication between processors or sensors.

A real-time environment is one in which responses to events should occur before a deadline. In a hard real-time system violation of such critical timing constraints result in material and/or human disasters. In contrast, a soft real-time system is one in which the real-time constraints are relaxed, and violation of deadlines do not result in catastrophies. It is obvious from its nature of operation that the CAN is a hard real-time environment where deadlines must be met. For example, a failure to signal a braking action in an automobile could lead to a fatal accident. Hence, one of the chief concerns is to minimize delays within the network.

One of the major hurdles in achieving system reliability, in such hard real-time systems, is finding an efficient way to schedule the events. I have considered the real-time scheduling as my research basis, since it is adaptable into a CAN type of environment. It has been my endeavor to pursue system configurations that are representative of the CAN. The following discussion provides the various analogies that can be related from the typical real-time computer systems to the CAN. First of all, a non-preemptive process scheduling aptly represents CAN messages since they cannot be removed from the bus once they are placed on it. Secondly, a uniprocessor machine can be easily viewed as the single channel of communication that the CAN adopts for message transfer. Scheduling overhead is assumed to be negligible in a real-time computer system. Also, exclusive access to the CAN bus is guaranteed once arbitration resolves bus contention. Finally, the processes arriving in a computer system can be readily equated to the messages generated in the CAN.

Sporadic and periodic messages

CAN messages are both periodic as well as sporadic in nature. Hence, a translation needs to be performed to have one type of message. The sporadic messages that are asynchronous in nature can be easily transformed to their periodic counterparts [Jeff91]. A periodic message is one that is generated repetitively in fixed time intervals. A typical periodic message M_p is defined as $M_p = (c, p)$ where 'c' is the communication cost, and 'p' is the period. A message M_p arriving at time t_k has the following rules of generation:-

- the (k + 1)-th generation of message M_p will occur at time $t_{k+1} = t_k + p$.
- the k-th transfer of the message M_p cannot start before t_k and must be completed no later than its deadline $t_k + p$. That is, the transmission time needs to be in the interval $(t_k, t_k + p)$.

A typical sporadic message is one that is generated in response to an internal or external event. A sporadic message M_s is defined as $M_s = (c, p)$, where the 'c' is the communication cost, and 'p' is the least interval of time before the next generation of such a message. A message M_s arriving at time t_k has the following rules of generation:-

- the (k + 1)-th generation of message M_s will occur no earlier than $t_k + p$; that is, $t_{k+1} >= t_k + p$.

- the k-th transfer of message M_s cannot start before t_k, and must complete no later than its deadline at $t_k + p$.

Thus, the two message types differ only by the first rule. A periodic rate can be imposed on the sporadic message by using the period 'p', that is the shortest interval of time in which a sporadic message arrives. Hence, any scheduling scheme for periodic messages can be used to schedule sporadic messages as well. Also, since the CAN messages are mostly periodic, it is convenient to use the above convention to define messages. A feasible schedule involves ordering messages in such a manner that all messages meet their deadlines.

Approaches to Scheduling

Two distinct approaches to scheduling messages are on-line (dynamic scheduling) and off-line (static scheduling). Since most messages are periodic, and their characteristics are known in advance, off-line scheduling is more suitable. A schedule length equal to the least common multiple of all message periods can be used to decide if the message set is schedulable or not [Xu93]. Also, it seems to be the only practical means of providing predictability in a real-time system.

Two parameters that can be used in the CAN message scheduling are message deadlines and message priorities. If message deadlines are equated to the corresponding message periods plus their previous deadlines, then an optimal priority assignment scheme can be used to resolve collisions during arbitration. Since priorities on the CAN are programmable, a priority assignment strategy based on a pre-computed schedule can be implemented.

Scheduling Issues

One of the chief concerns in a hard real-time system using pre-run-time scheduling is satisfying relevant timing constraints. The objective also is to minimize the schedule length which is the longest time taken to transfer all messages. Two main theorems are discussed in detail in [Jeff91]. In terms of message scheduling, if $M = \{M_1, ..., M_n\}$ is a set of periodic messages, where $M = (c_i, p_i)$, then the messages are in increasing order of periods. In other words, for all messages, M_i and M_j, $i > j$ implies $p_i >= p_j$. The two necessary conditions for this message set to be schedulable are:-

- The overall bus utilization cannot exceed 100%; that is,

$$\sum_{i=1}^{n} (c_i/p_i) \leq 1$$

- For any i between 1 and n, and L between p_1 and p_i,

$$L \geq (c_i + \Sigma ((L-1) / p_j) * c_j)$$

This suggests a non-preemptive schedule with no inserted idle time. The right hand side of the equation gives the bus utilization that can be realized in the interval L, starting at the generation of message M_i and ending before its deadline.

Real-time scheduling algorithms

Following are real-time scheduling algorithms proposed in the literature:

Earliest deadline first (EDF) scheduling, which is also called *relative urgency (RU)* scheduling, has been proven to be universal for sporadic and periodic message sets. A concrete message translated from concrete task is one that has a release time associated with it. In [Jeff91], it has been proven that non preemptive scheduling of concrete periodic tasks is NP-hard in the strong sense.

In EDF scheduling, a message is assigned the highest priority if the deadline of its current request is the earliest [Liu73]. Scheduling decisions are made at the time of each message generation. Thus, this suits a dynamic scheduling scheme, where priorities are assigned based on the current request.

The *rate monotonic priority assignment algorithm* when translated to message scheduling says that messages with higher generation rates get higher priorities [Mok83]. This essentially means that a message with the highest priority has the maximum arrival rate. That is the message with the shortest period has the highest priority. The heuristics here are based on the fact that the most important, or time critical message is the one that is generated most often. Hence, the algorithm is suited for static priority assignments, where priorities are decided based on message periods that are known in advance.

The *mixed scheduling algorithm* provides a mixed approach that can schedule a set of messages with shorter periods by using a fixed priority schedule that is static, and the remaining set of messages with larger periods by an EDF schedule that is dynamic. Hence, this type of scheduling takes the potential advantages of both on-line as well as off-line scheduling techniques to provide an optimal schedule.

A *priority exchange algorithm* has been discussed in [Wang92] for a CAN real-time environment. It assigns priority by increasing order of transmission times of the messages. Priorities are then exchanged based on their deadline requirements until the messages are ensured of meeting their deadlines. This study was based on a maximum arrival rate analysis of the messages. A Poisson distribution of message generation, and exponential transfer times, was considered for this purpose. The results showed that under heavier loads the system experienced larger delays for lower priority messages, whereas under lighter loads it remained stable.

Earliest deadline first with dynamic deadline modification (EDF / DDM) was studied by [Jeff92]. This scheme is used to dynamically alter deadlines of resource requesting tasks. This is more suitable for a process scheduling scenario rather than message scheduling.

The *least slack algorithm* is another on-line scheduling technique where preemption is allowed [Mok83]. The slack time of a message is defined as the time interval remaining between the message transfer completion time and its deadline. It is taken to be zero if the message misses its deadline. Intuitively it turns out to be the maximum time a process can be delayed before it is bound to miss its current deadline. In a least slack algorithm, at any point of time the message with the least slack time is scheduled next. Hence, it is essentially an on-line scheduling scheme.

From the above discussions about various scheduling schemes, one of the key considerations in making a choice is to look at the system configuration. If the on-line scheduler is going to burden the system resources with a high scheduling overhead, then dynamic scheduling would be a bad choice. Another viewpoint is that if the message generation is highly unpredictable resulting in a lot of deadlines being missed, then an off-line scheduler is not helpful. Hence, a careful assessment of what a priori knowledge of the message set is available can determine which type of algorithm should be used. The most important characteristics to look for in a message set would be periodicity, release times, and deadlines requirements.

The CAN message set is known, and most messages are periodic in nature. Hence, an off-line scheduler is most preferable. Also, on-line scheduling involves additional scheduling overhead to perform scheduling functions while the network is running. Another potential disadvantage is the requirement of additional hardware required to support an on-line scheduler. A modified version of the rate monotonic priority assignment algorithm is well suited for scheduling the messages in CAN. Priorities are assigned by increasing order of periods. Ties in message priorities are broken arbitrarily. An example of an priority assignment is as shown in Figure 8.

Message Periods	20	10	5	100	70	500	40
Default Priority	1	2	3	4	5	6	7

Message Periods	5	10	20	40	70	100	500
Assigned Priority	1	2	3	4	5	6	7

Figure 8. A Rate Monotonic Priority Assignment

The main difference encountered in message scheduling as opposed to process scheduling is in the occurrence of error conditions resulting in retransmissions. Of course, message scheduling does not involve process synchronization, precedence relations, or interprocess communication as in process scheduling.

CHAPTER IV

SIMULATION OF THE CAN

Model

Simulation offers a flexible approach for performance evaluation of the CAN, and any computer network in general. It requires few assumptions and approximations of the network details. A detailed modeling of the CAN is useful to explore the various design aspects. It also aids in predicting changes in network performance, and comparing alternate designs. Analytical and graphical results can aid the network designer in creating a prototype model. The major drawback is the inability to predict the system reliability.

Various modeling approaches including queuing models, Petri nets, and finite state machines have been used in the past. A queuing network model does not represent the protocol aspects of the CAN, while the finite state machine model cannot handle the topological features of the CAN. Petri net models can be used to verify the CAN protocols. A more simplistic model for discrete event simulation of the CAN is presented in Figure 9.

Design

The program design was made in three phases. The three phases are specification of the protocols, specification of the topology, and specification of the nodes. The first phase was to make a detailed study of the CAN protocols. The *protocol specifications* includes the rules of communication dictated by protocols within the network. This part

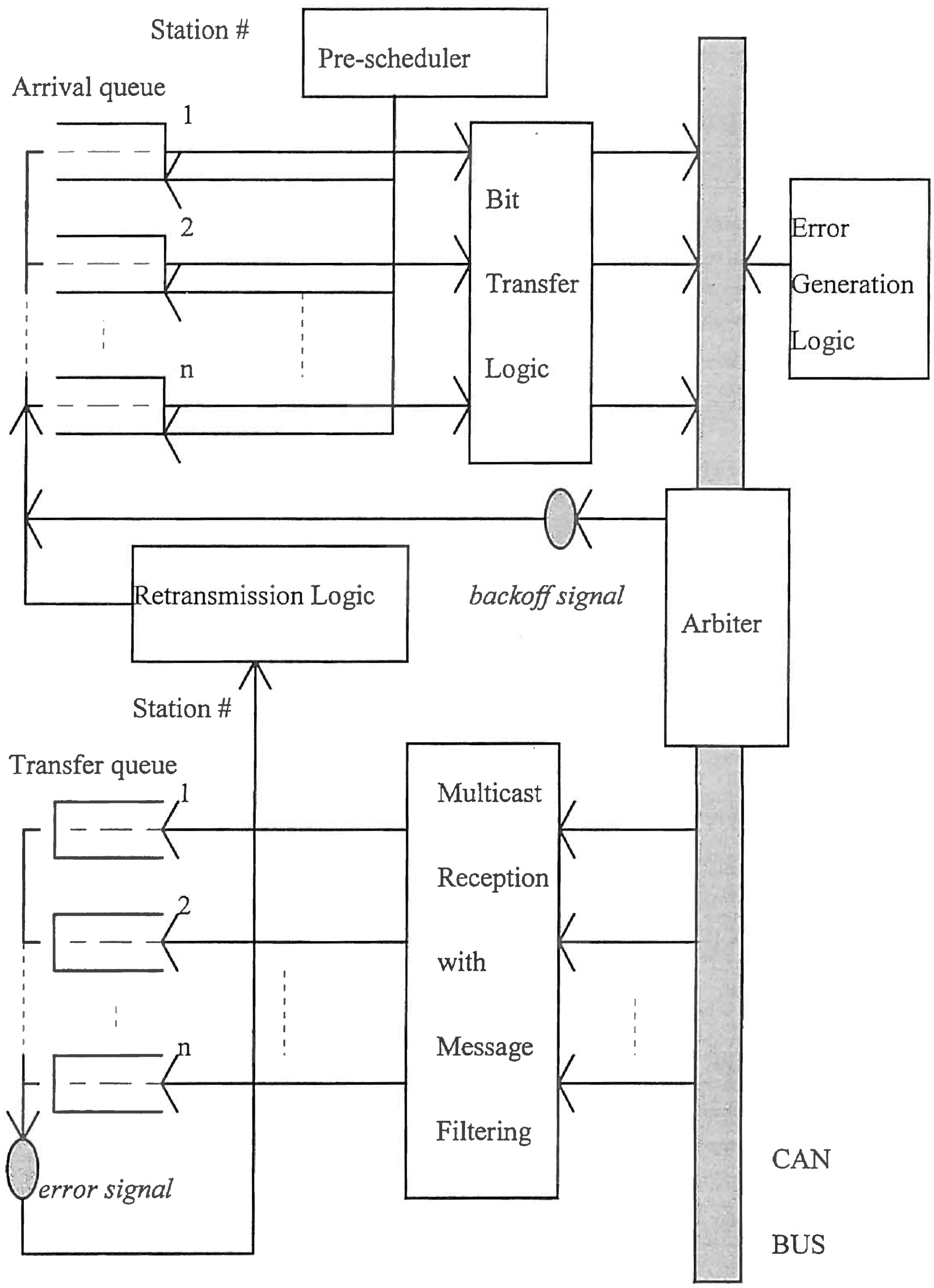


Figure 9. CAN Simulation Model

forms the core of the simulation, as it represents the flow of control within the simulation program itself. The important CAN protocols to be studied are the arbitration, message transfer, error detection, error signaling, and retransmission. Since the CAN is a real-time network, and messages are mostly periodic, an off-line scheme is adopted as suggested in the previous chapter. As the arbitration, error checking and message transfer operations are bitwise, a bit by bit simulation methodology is used (i.e., bit transfers are simulated instead of message transfers).

The second phase involves the *topological module specification*. This essentially determines the physical layout and the physical transmission characteristics under which the network operates. The layout specification is simply the way in which all nodes on the network are connected to the single CAN bus as in Figure 1. The bus topology of the CAN offers multimastership and multicast reception. Thus, any node that has a message to be transmitted simply transmits it, bit by bit, on the bus. If a collision occurs, arbitration is used to determine the winner. All nodes receive all messages in the simulation because the mask register functionality does not affect the network performance. The most important physical characteristic of the CAN medium is the baud rate. A CAN bus with a transfer rate of 250 kbps is selected. The unit of time in the simulation is assumed to be one bit time; that is, the time taken to transmit a single bit. In physical terms, one bit time is 4 μs for a 250 kbaud bus. So all times within the simulation are converted to bit time by dividing the simulation bit time (in μs) by 4. The electrical characteristics of the CAN are significant only with respect to voltage fluctuations that result in error conditions.

The final phase of the design is the *node module specification*. This includes the specifications of attributes of all nodes connected to the bus. The primary goals of a node are message generation, message transfer, and message reception. All other station details are less important. The following are the main features to look for in a node. The first feature is the type of messages it generates, whether periodic or sporadic. All sporadic messages are translated to periodic messages by the simple technique described in Chapter III. The second feature in a node is its message characteristics. This includes message length, message representation (standard/extended), message mode (data/remote/error/overload), message release times, and message default priority, if any. Finally, each node receives messages depending on the kind of objects it has. This feature is ignored as all nodes on the network in the simulation receive all messages.

Implementation

CAN is a dedicated network being used for specific real-time applications. It has its own distinct protocols, and standards that define its operation. We develop a simulation package using a bottom-up design. The CAN simulator is coded in the *C* programming language, and presented in Appendix B. The implementation details of the program are described below.

One of the key issues in a simulation is to map the physical time to the simulation time within the program. This parameter indicates the total simulation time for which the trace has been generated. A global clock forms the simulation time, and it maps to each unit of time spent in the network. Initialization of all node parameters, after reading input values is performed first. The periods of all messages are then tested for the two real-time constraints mentioned in Chapter III.

Priorities are assigned to messages based on their schedule order. The simulation gets underway with the arrival of a new message. The first arrival of a message is determined by its release time. Once a message is released it arrives at its periodic rate. If no message is arrives, then the bus is in an idle state. Each idle state results in the incrementing of an idle time counter. When one or more messages are generated at the same point in time, a message cycle is started. If more than one message arrives, an arbitration process is initiated to resolve the conflict. The eventual transmitter of the message starts a message transfer. A conceptual flow diagram of the bit transfer,

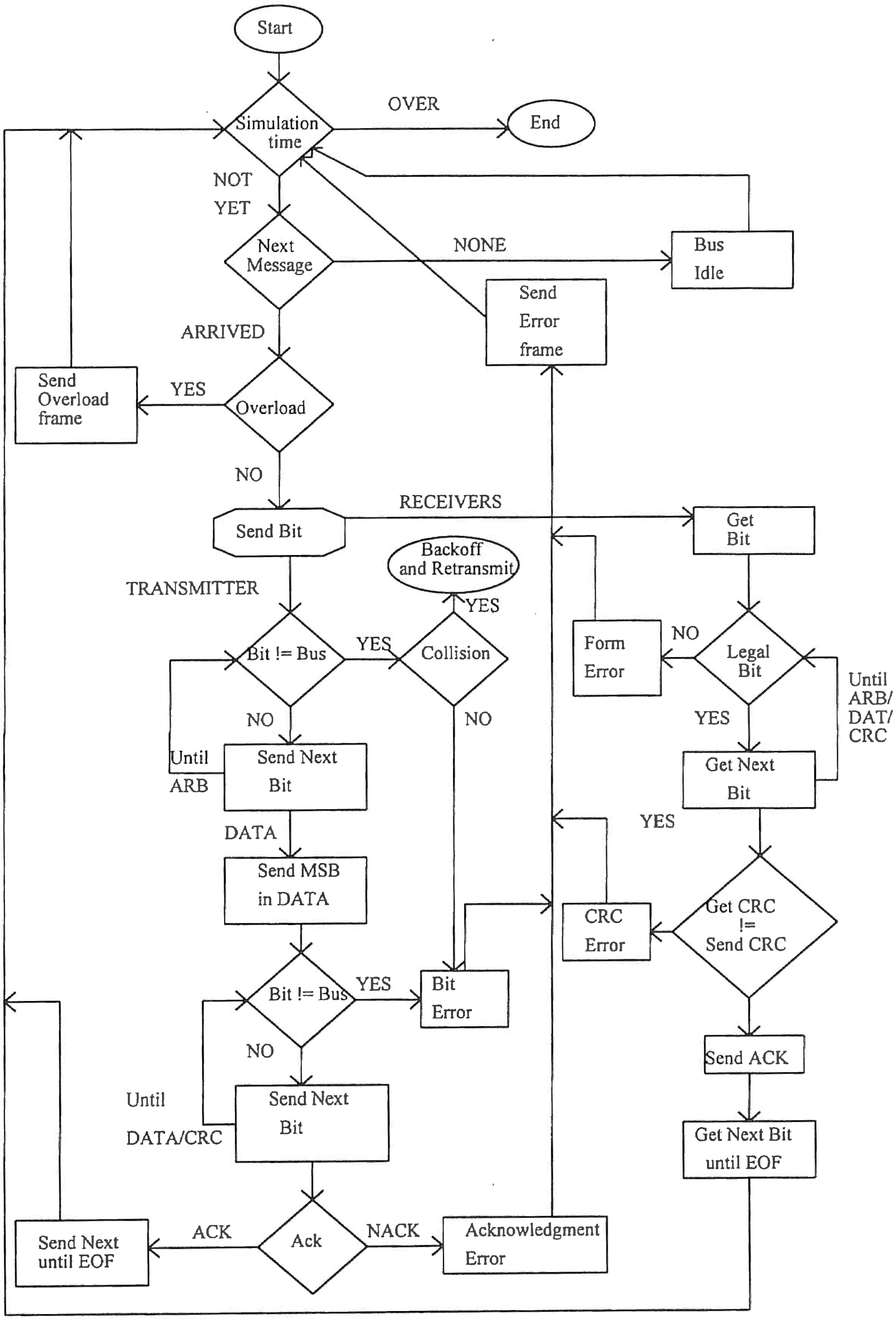


Figure 10. Bit Transfer Flow Diagram

including the arbitration protocol, is shown in Figure 10. Every bit put onto the CAN bus is a logical value, found by testing the appropriate bit position within the transmitted message using Boolean logic. Every bit transfer results in an additional bit time being spent in the simulation. After the lapse of a bit time the bit is received by all receivers simultaneously. If more than five consecutive bits of equal value are sent, then a bit stuff is simulated by incrementing the simulation time by one. Thus, all bits within a CAN message are sent until an EOF or an error condition is detected.

Errors are generated at random times. Error value is determined using the following formula:

random_value = (r * c) *mod* error_rate

error_point = random_value + simulation_clock

where 'r' is a random number generated by a random number generator, $c = 10^n$, such that 'n' is the required number of digits for the random value, and error_rate is used to vary error points within the simulation.

When an error occurs, a bit being transmitted is complemented to produce an error. Every bit is monitored for an erroneous transmission by the transmitter and all receivers. The transmitter detects bit errors and acknowledgment errors, while the receivers detect frame errors, CRC errors, and stuff errors.

Messages are generated at each station in conformance with the frame formats. The extended frame is taken to be the basic data structure. The standard frame is built over the extended frame by ignoring the extended identifier fields during transmission. The first six bits are used for priority assignments for a total of 63 messages The extended data content has not been used as proposed, since its content is not required, and does not affect the simulation in any way. The data for each frame consists of 0 to 8 bytes, and is generated randomly in a byte by byte fashion. The data details are not considered as their functional value is immaterial.

A 15-bit frame check sequence is derived using the code given in [Bosc91]. A 15-bit shift register is used to perform polynomial division using a polynomial generator, and the remainder of this computation is the CRC sequence. This value is computed for the bits ranging from the SOF field to the end of the DATA field. The receivers on their part compute a similar frame check sequence, using the same code. A check is made to see if the receiver's CRC value matches with that of the transmitter. A CRC error is signaled if a mismatch occurs. All of the receivers flag a positive acknowledgment by overwriting the recessive ACK SLOT with a dominant bit. After all EOF bits are sent, control is returned to the message cycle routine that keeps checking for the next message arrival until all of the simulation time has elapsed. At the end of the run, various statistics are calculated and output. The parameters under study are throughput, latency, time, response, error, and collision characteristics. A detailed discussion of the graphical, and statistical analysis ensues.

Statistical and Graphical Analysis

The input data for the simulation is selected from the CAN specification manual [Bosc92]. This set is used because it represents a real-life CAN situation. Also, additional data has been included by modifying the original CAN set in [Bosc92] to facilitate testing, and obtain various network performance measures. The input data format that is used in the input data file is as follows:

Simulation time (in milliseconds)
Bandwidth (in number of bits per bit time)
Error rate for the random error generation
Node_name Number_of_messages Number_of_objects

Msg_name	*Release time*	*Priority*	*Period*	*No_of data_bytes*	*Data(1)/ Remote(0)*	*Standard(1)/ Extended(0)*
Objects						

Simulation runs have been performed in the time range of 100 ms to 10 seconds. This is done to accommodate for load variations, error rates, and irregularity in message generation times. The bandwidth is used to increase the simulation length to produce the effect. The error rate parameter is used as a means of varying error generation points and rates. Each node is defined with a certain number of messages and objects. A node may use more than one type of message. Any node can generate only one message at a time. So, an upper bound of message arrivals is the number of nodes on the network. Each message in turn has a period (in ms), a release time, a default priority in the range of 0 to 63, a number of data bytes in the range 0 to 8, a remote or data flag, and a standard or extended frame flag. Since the first 6 bits in the arbitration field have been used for priority assignment, only 63 messages can be input to the program. Since most messages on the CAN are periodic, all messages have been taken to be of that nature. All message release times are 0. The objects may be used to simulate the message filtering functionality or a destination specific transmission.

Verification of the simulation is performed on a single node with a single message. The message is an 8-byte extended data frame, with a period of 10 ms. A 100 ms run, with no errors and no collisions, produced the following results:-

Total number of messages transmitted = 10

Idle time = 94.82 ms	Busy time = 5.18 ms	Error time = 0 ms
Load = 5.18%	Throughput = 100 msgs/s	

It is obvious that a message with a period of 10 ms arrives 10 times in a 100 ms run, and so, 10 messages are transmitted. This also leads to a throughput of 100 msgs/s (10 messages * 10 such runs). The sum of the idle and busy times gives the total simulation time. The message consists of 128 bits inclusive of the interframe space to give a total transmission time of 5.12 ms ((128 bits * 4 μs/bit * 10 messages) / 1000). So, the network load over a 100 ms period is (5.12 / 100) * 100, that is 5.12% ≅ 5.18%. The small difference is due to some additional bits being sent at the end of the simulation.

The simulation is performed on 8 different input message sets, with 2, 3, 10, 17, 20, 30, 40, and 50 nodes corressponding to 3, 5, 17, 24, 27, 39, 50, and 60 messages respectively. The above load conditions are labelled 1, 2, 3, 4, 5, 6, 7, and 8 respectively for curves and load points in the graphs. This gives a variation in the offered load on the network. The input files for the 8 different message sets are given in Appendix C. The message specifications were adopted from [Bosc92]. The statistics are computed and output at 5 sampling intervals within the trace. The statistics at the end of each run is presented in Appendix D, with inputs being numbered in Roman numerals. The output includes network and node statistics that help in making a comprehensive performance evaluation of the network. The following discussion analyzes the results obtained out of the statistics using some representative graphs.

Network load is defined as the ratio of utilized bus time to the total bus time; that is,

Load = (Busy time + Error overhead time) / Total bus time

where Total bus time is the total simulation run time.

The utilized time includes useful message transmission, as well as error message transmission time. The 5 different runs produce graphs as shown in Figure 11.

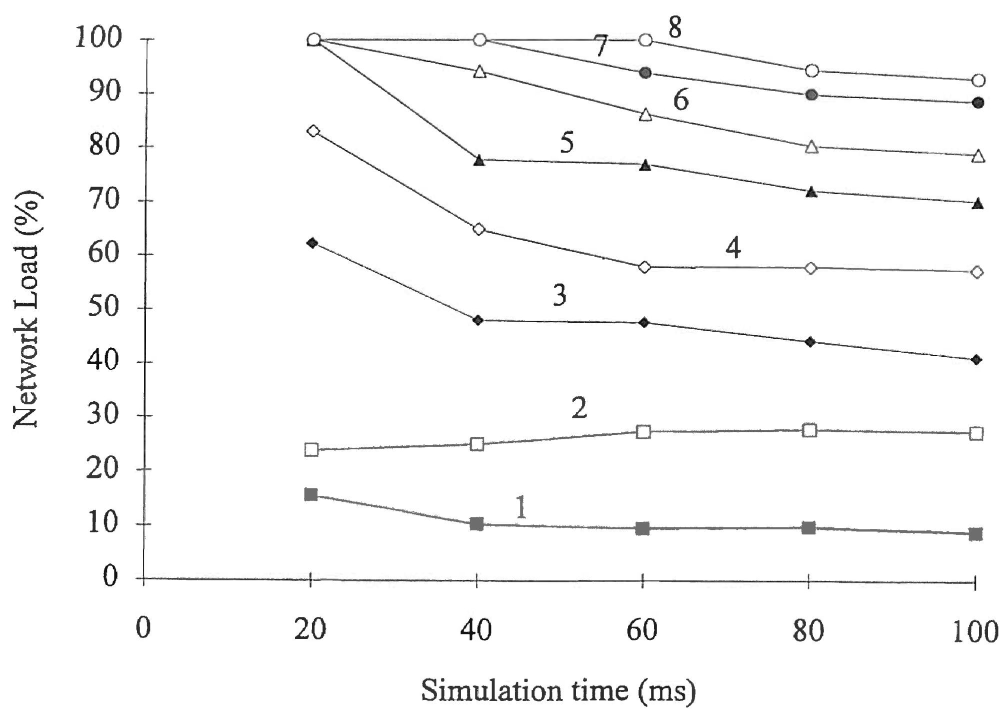


Figure 11. Load Characteristics

Initially, all runs have a high load signifying the simultaneous release of messages by all nodes on the network. Then, there is a near exponential decrease in the load as the distribution of message generation times is more varied. Towards the end, the graphs tend to become horizontal curves representing a more steady state system behavior. It can be observed that as the message set gets larger the exponential decrease lessens. This signifies that as the message set increases, more transmissions are getting clustered together.

Network throughput is defined as the total number of messages transmitted per second, and is the given by the formula:

Throughput = Total number of messages transmitted / Total bus time

The throughput versus load graph is as shown in Figure 12. A predicted behavior is seen in the form of a linear shaped graph, but at the second sampling point a sharper rise occurs. This may be attributed to the fact that more messages get transmitted as the load is increased. Also a lower number of errors for this load point increases productive transmission. This observation is made from the error graph in Figure 16.

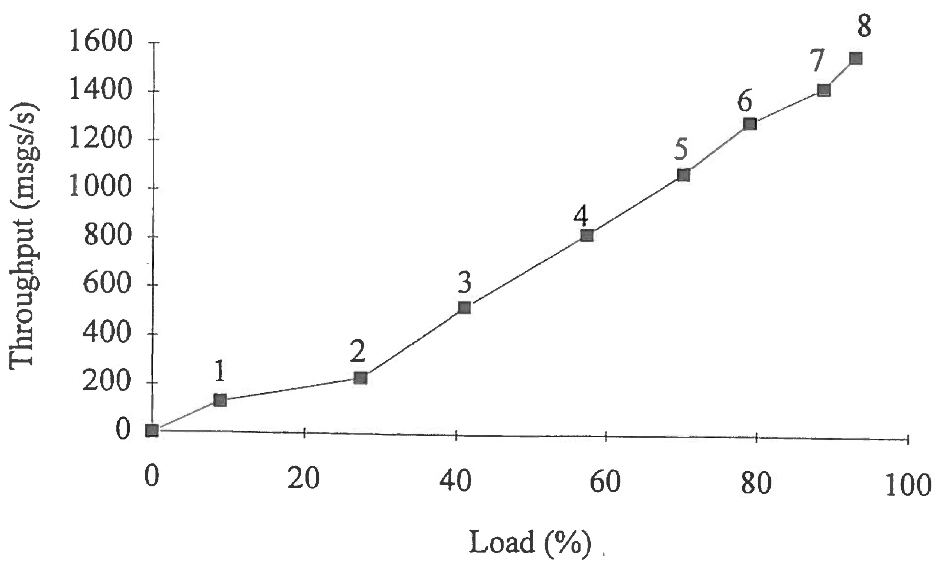


Figure 12. Throughput vs Load

The *time analysis* graphs of Figure 13 shows the three major time parameters analyzed in the simulation with respect to the load. It can be seen that the sum of all three time quantities is equal to the simulation time which is 100 ms in this case. Idle time is the time for which no transmission takes place, that is the bus is in an idle state.

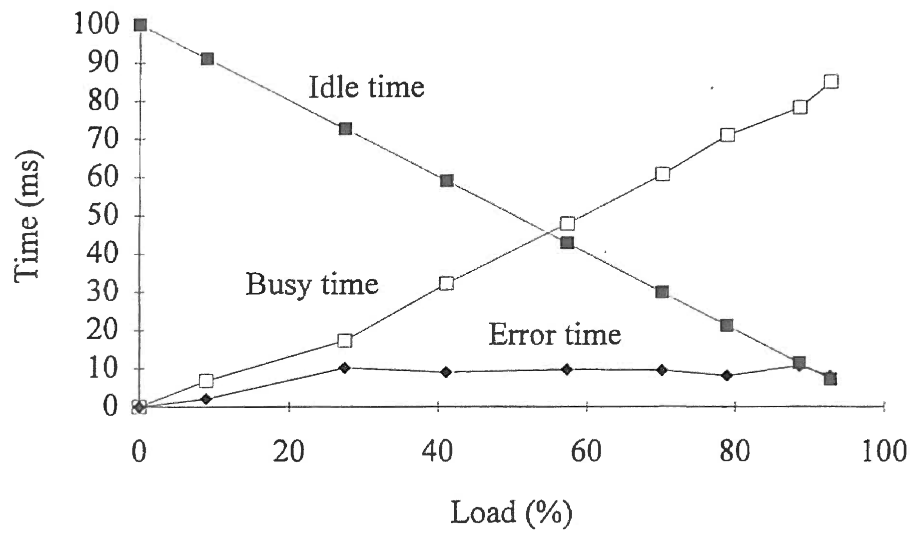


Figure 13. Time Characteristics

The idle time graph is linearly decreasing with increased load. This is obvious from the fact that as more messages are generated the bus is free for a smaller amount of time. Busy time is the time for which the bus is busy due to transmission of a data or a remote frame. A variation in the linear behavior after the second load point is due to the greater number of message transfers as discussed earlier. Similar reasoning can be used to attribute the cause of the decline in error times after the third load points. Error time is the time for which the bus is utilized to transmit error messages. The unevenness of this graph is because of the random distribution of error generations.

The *average response time* in a network is the average amount of time taken by all messages to gain bus access, once they are generated. The graphs in Figure 14 shows the changes in the average response times at various sampling points within the simulation trace. The graphs for the 8 different topological conditions present interesting characteristics of the CAN. The first two load conditions have negligible amount of average response times. Fewer nodes that offer a lower load produce a more stable

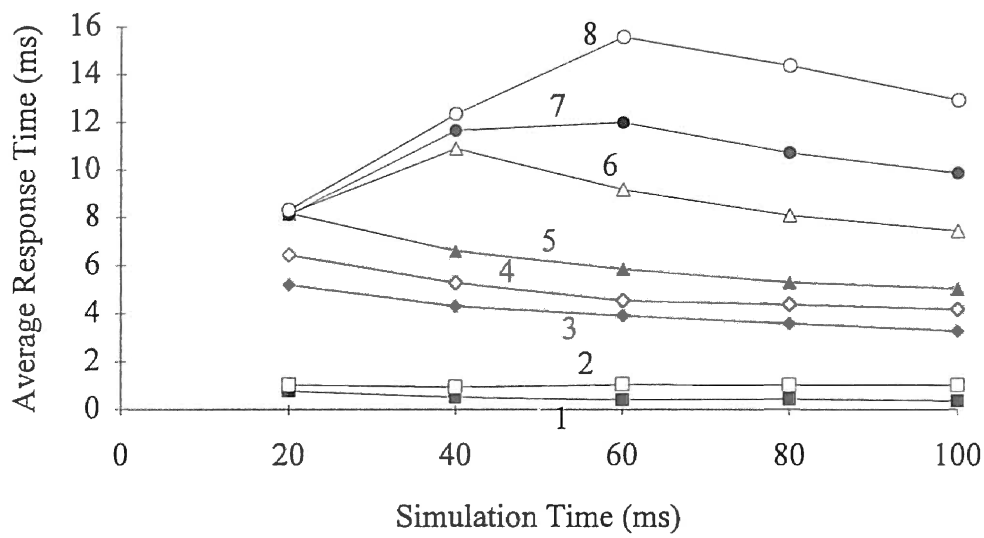


Figure 14. Response Time Characteristics

response time, while more than 20 nodes offering greater loads show a sharp increase in the response times until the initial overload is accommodated. This shows a slow response for a maximum arrival rate at the beginning, resulting in delayed service.

One of the key parameters under study in a network is its delay or latency characteristics. The network delay in terms of the transmission time is not a useful performance measurement quantity. This is because of the insignificant time involved in message transmission. The *average latency,* defined for all messages missing their deadlines, is the average time elapsed between message deadlines and their actual completion of transfer. The graphs of Figure 15 trace latency characteristics. As expected, lower loads produce lower latency times, while larger loads have higher latencies. The first four load conditions have 0 latencies throughput as no message misses its deadline. The last configuration has a greater slope due to large number of messages missing their deadlines at the beginning.

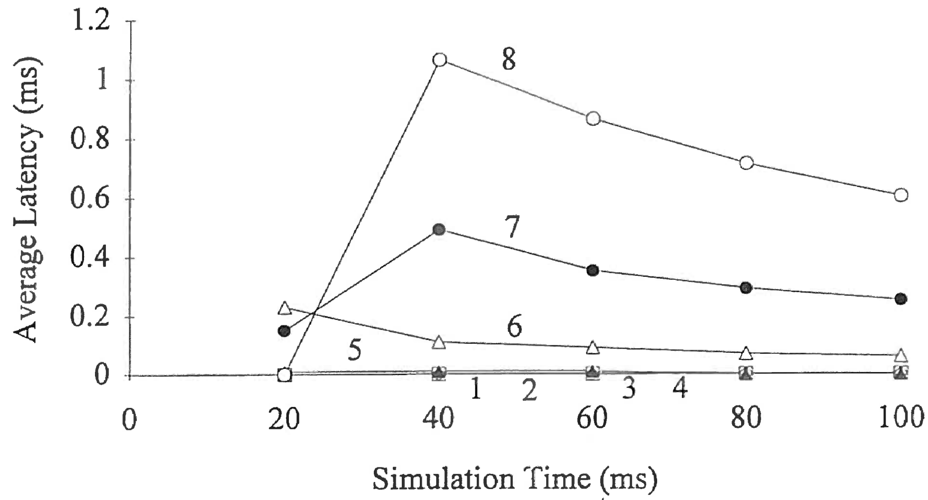


Figure 15. Latency Characteristics

The graphs of Figure 16 shows the *number of errors* produced at different load points. It shows a random distribution, and signifies its effect on the system behavior as

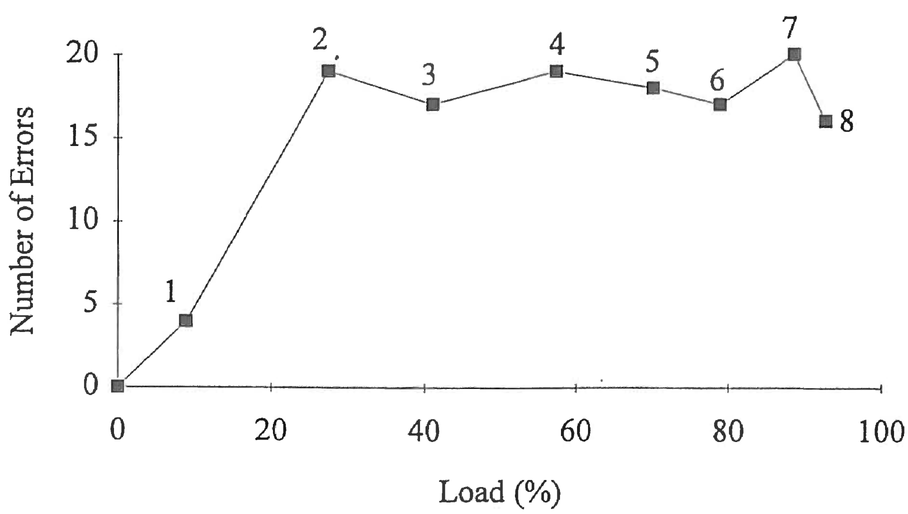


Figure 16. Error Characteristics

discussed in the previous graph analysis.The variations in error rates may also be attributed to the nature of the message set. If more messages have larger periods, then lesser number of errors hit the messages, while smaller periods cause a higher error rate. Changing the error distributions produces graphs almost similar to the one in Figure 16.

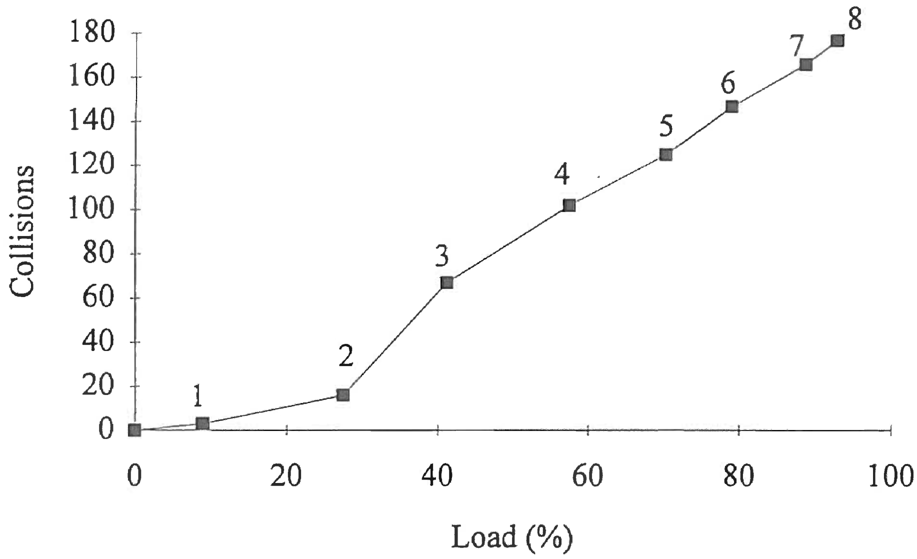


Figure 17. Collisions vs Load

The graph of Figure 17 shows the number of collisions at different load points. This is a linear shaped graph with a break at the second load point. So, loads above 30% produce greater number of collisions. It is interesting to compare this almost linear nature to that of the Ethernet. Since, Ethernet uses a 1-persistent CSMA/CD, collisions become more rampant as each collision results in destruction of all messages in contention. This scenario is aggravated with the arrival of more messages. Thus, throughput characteristics of an Ethernet is a rapid linear rise in throughput for light load conditions, as most of the channel idle time is avoided [Stal88]. After a peak load condition of around 20%, an exponential decay in the throughput occurs for heavier loads. This gives the CAN a definitive edge over Ethernet for real-time operations under heavy loads.

CHAPTER V

CONCLUSIONS

Summary of Results

The Controller Area Network (CAN) has already been proven to work well under loads of 30%. The present simulation has load conditions varying from 0 to 100%. Most results show good performance under loads of 40%. Although loads under 30% produce a very low throughput. The inclusion of errors in the simulation produce measures of error tolerance. A stable system operates with a maximum of around 20 errors over a 100 ms period. There is no latency for load conditions reaching 60%. A topology with 50 nodes and 60 messages results in a greater number of messages missing their deadlines.

On the whole, the network performs admirably with a load as high as 40%. The throughput achieved for this scenario is around 500 messages/sec. From the time characteristics it is clear that there is a large idle time that can provide for some additional loading if necessary. Response times are faster for loads lesser than 60%, with the average response times being less than 6 ms over a 100 ms run.

Conclusions

From the analysis, it is clear that the network begins to degrade under severe loads. Since no message can afford to miss its deadline, guaranteed performance is mandatory. An added load of 10% to the original CAN specifications does not affect the network performance. An error rate of 1 error every 1250 bits provides near optimal performance. Considering the fact that a pessimistic approach is chosen in terms of release times, a better performance can be assured for a more typical CAN environment.

The rate monotonic priority assignment algorithm seams to work well with the predominantly periodic message set. Some refinements to the scheduling algorithms can be used to test the network performance. The test for the two real-time constraints is vital in deciding if a schedule is safe or not. It is also evident that the CAN's versatile arbitration protocol enhances performance by avoiding delays due to message destruction. Thus, the CAN has proven to be an efficient real-time network.

Future Research

Since CAN is a developing network, research findings are essential for its growth. Every network goes through an evolution, and its success depends on how well a performance evaluation is made before formally setting standards. This thesis opens up several avenues for research. Studies in the changes of key network parameters has revealed some network limitations. The simulation program has been crucial in arriving at the above conclusions. The bit by bit logic within the simulation has aided in error analysis. It is also useful for making design changes in the message formats. A full scale investigation of different topologies could be made. This could give an insight into a wide range of design issues. One of the aspects left out of the study was fault tolerance. Although the fault confinement logic has been implemented within the simulation, it could not provide adequate results. This was due to the fact that enough errors were not generated to create faulty nodes. Modifying the simulation could give some fault tolerance measures.

REFERENCES

[Arne87] Arnett, J. D., A High Performance Solution for in Vehicle Networking - Controller Area Network(CAN). Earthmoving Industry Conference, Peoria, IL, April 7, 1987, SAE paper #870823.

[Bosc91] CAN Specification Version 2.0, Robert Bosch GmbH, September, 1991.

[Bosc92] J1939 Recommended Practices for a Serial Control and Communications Vehicle Network (Class C), Robert Bosch GmbH, December, 1992.

[Eyho89] Ey, Horst., Controller Area Network (CAN) Components, Electronic Component and Application, vol. 9, no. 3, 1989, pp. 155-158.

[Gupt88] CAN Facilitates In-Vehicle Serial Communications, On Board Communication for Machinery and Control, ASAE paper #88-1649.

[Harb91] Harbison, Samuel P., and Steele, Guy L. Jr., C: A Reference Manual, Third Edition, Prentice Hall Software Series, Englewood Cliffs, NJ, 1991.

[Inte88] Intel Corp. Auto-Communication Chip Replace Bulky Wires, Design News, vol. 44, August 1988, pp. 120.

[Iver88] Iversen, Wesley R., Intel Gets a Jump on the Auto Multiplex Market, Electronics, vol. 61, March 88, pp. 31-32.

[Jeff91] Jeffay, K., Stanat, D. F., and Martel, C. V., On Non-Preemptive Scheduling of Periodic and Sporadic Tasks, Proceedings of the Twelfth IEEE Real Time Systems Symposium, San Antonio, Texas, December 1991, pp. 129 - 139.

[Jeff92] Jeffay, K., Scheduling Sporadic Tasks with Shared Resources in Hard Real Time Systems, Proceedings of the Thirteenth IEEE Real Time Systems Symposium, Phoenix, Arizona, December 1992, pp. 89 - 99.

[John86] Johnson, W and Volk, J., A Proposal for a Vehicle Network Protocol Standard, Feb. 1986, SAE paper #860392.[Jord88] Jordan, Pat., Controller Area Network, Electronics and Wireless World, vol. 94, Aug. 88, pp. 816-819.

[Jurg86] Jurgen, R. K., Coming from Detroit: Networks on Wheels, IEEE Spectrum, June 86, pp. 53-59.

[Kien86] Kiencke, U., Dais, S. and Litschel, M., Automotive Serial Controller Area Network, International Congress and Exposition, Detroit, Michigan, Feb. 24-29, 1986, SAE paper #860391.

[Liu73] Liu, C. L., and Layland, J. W., Scheduling algorithms for Multiprogramming in a Hard Real Time Environment, Journal of the Association for Computing Machinery, Vol. 20, No. 1, January 1973, pp. 46 - 61.

[Mies86] Miesterfeld, F. O. R., Chrysler Collision Detection (C^2D) - A Revolutionary Vehicle Network, International Congress and Exposition, Detroit, Michigan, February 24-28, 1986, SAE paper #860389.

[Mok83] Mok, A. K., Fundamental Design Problems of Distributed Systems for the Hard Real Time Environment, Ph. D. Thesis, Massachusetts Institute of Technology, MIT/LCS/TR-297, May 1983.

[Phai86] Phail, F. H., Arnett, D. J., In-Vehicle Networking - Serial Communication Requirements and Directions, SAE paper #860390.

[Phai88] Phail, F. H., Controller Area Network - An In-Vehicle Network Solution, 1988 International Summer Meeting of the American Society of Agricultural Engineers, paper #88-3021.

[Stal88] Stallings, William., Data and Computer Communications, Second Edition, Macmillan Publishing Company, New York, 1988.

[Xu93] Xu, J., and Parnas, D. L., On Satisfying Timing Constraints in Hard Real Time Systems, IEEE Transactions on Software Engineering, Vol. 19, No. 1, January 1993, pp. 70 - 84.

[Wang91] Wang, Z., Analysis and Design of Controller Area Networks, MS. Thesis, Oklahoma State University, December 1991.

[Wang92] Wang, Zhengou., Stone, Marvin., and Lu, Huizhu., A Message Priority Assignment Algorithm for CAN Based Networks, Proceedings of the 20-th Annual Computer Science Conference, Kansas City, MO, March 3-5, 1992, pp. 25-32.

APPENDIX A

GLOSSARY

1-persistent CSMA/CD is a scheme where a node detecting a collision backs off for a random time, and retransmits with a probability of 1; that is whenever the bus is idle.

Bandwidth is the maximum possible data rate within the network in terms of bits per second.

Bit time is the unit time taken to transmit a bit across the bus.

Broadcast networks are those where messages may be received by all stations.

Bus off node is one that has an error count greater than 256.

Bus topology has a single communication channel to which all nodes are connected.

CSMA/CD is a carrier sense multiple access collision detection mechanism.

Collisions are the result of overlapping transmission of messages by more than one node.

Contention is a dispute between more than one node for access to a common channel at the same time.

Error active node is one that has both its error counts less than or equal to 127.

Error passive node is one that has either error count greater than 127.

Information processing time is the time segment starting from the sample point, and begins a bit level.

Message is information sent on a bus with a fixed format.

Multicast networks are those where more than one node may receive a message.

Nominal bit rate is the number of bits transmitted per second.

Nominal bit time is a reciprocal of the nominal bit rate, and is divided into four segments as below.

Peak load is the maximum load conditions occurring in the network.

Phase segment1 and Phase segment2 are used to compensate for edge phase errors.

Propagation segment is used to compensate for physical delays within the network.

Protocol is a formal set of conventions governing the format and relative timing of message exchange.

Receiver is a communicating device that receives a message from an alien device.

Ring topology has a circular channel to which nodes are connected.

Sample point is the point of time at which the bus level is read and interpreted as the value of that bit.

Star topology has a central node to which all other nodes are linked.

Station is a device that processes, sends, and receives data over a network.

Synchronization segment is used by the nodes on the bus to synchronize.

Time quantum is a fixed unit of time derived from the oscillator period.

Transmitter is a communicating device that sends out a message to one or more alien devices.

Tree has a hierarchical structure with a root node and several layers of nodes below it.

APPENDIX B

SIMULATION PROGRAM

Procedure call representation:

```
main()
    sys_init()
        packs_init()
        packr_init()
    get_parm()
    node_addressing()
    edf_schedule()
    msg_cycl()
        packet_gen()
            crc_gen()
        arbitrate()
        msg_transfr()
            transmit()
                get_bit()
                    rand_error()
                msg_filter()
                    receive()
                        crc_check()
                    bit_stuff()
                send_error_frame()
                send_overload_frame()
statistics()
```

```
#include <stdio.h>
#include <fcntl.h>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <malloc.h>

#define M_MSGS              10
#define TOT_MSGS            100
#define MAX_NODES           60
#define MAX_NAME            10
#define MAX_OBJS            60
#define MAX_OVERHEAD        67
#define MAX_MSG_LEN         132
#define SAMPLES             4
#define RAND_RATE           1000
#define YES                 1
#define NO                  0
#define BUSY                1
#define IDLE                0
#define OVER                0
#define FAILURE             0
#define SUCCESS             1
#define OVERLOAD_ERROR      2
#define FORM_ERROR          3
#define CRC_ERROR           4
#define ACK_ERROR           5
#define ACTIVE              1
#define PASSIVE             0
#define BUS_OFF             -1

int busy_time, latency, slack_time, response_time, remote;
int losers, idle_time, errors, msg_time, error_overhead;
int sample_count, sim_cnt, count, data_frame, standard;
long int tic, finish, sample_time, sample_period;
int rand_rate, error_period, bandwidth, random_error;
```

```
int collisions, transmitted, missed, retransmitted;
char bus, bus_flag;
int max_period, min_period, periodic_error;
int nr, n, m, total_msgs, total_nodes, simulation_time;
int no_of_receivers, ones, zeros, prv_bit, pos;
int ack_count, overload_count, crc_count, form_count;
FILE *ip, *op, *st;

typedef struct {  /* formatted message */
  unsigned char eof;
  unsigned char ack;
  unsigned char crc[2];
  unsigned char dat[8];
  unsigned char ctr;
  unsigned char arb[4];
  unsigned char sof;
  unsigned char isp;
  } PACKET ;

typedef struct { /*  message parameters */
  int data_len, period, release, deadline, prior;
  int msg_format, msg_mode, arb_lost, error_flag;
  float  trans_time;
  char msg_name[MAX_NAME];
  } MESSAGE;

typedef struct { /* node parameters */
  PACKET packs, packr;
  MESSAGE msg[M_MSGS];
  int curr_msg, no_of_msgs;
  char node_name[MAX_NAME];
  int no_of_objs, object[MAX_OBJS];
  unsigned char ovrhd_delim, err_flag, err_delim;
  unsigned char ovrhd_flag,  address;
  char prv_bus, prv_bus_flag;
  int status, bit_val, transfer, receive;
```

```
  int dead_count, recv_err_cnt, trans_err_cnt;
  int error_count, lost_count, trans_count;
  } NODE;

NODE node[MAX_NODES];

typedef struct { /* scheduling parameters */
  int dead, node, msg;
  float  trans;
  } ORDER;

ORDER order[TOT_MSGS];
```

/* This function performs initialization of the transmitter packet when new message is created. */

```
void packs_init(int S)
{
 int j, i;
 i = S;
 node[i].packs.eof = 0;
 node[i].packs.ack = 0;
 node[i].packs.crc[0] = 0;
 node[i].packs.crc[1] = 0;
 for (j = 0;j < 8;j++)
   node[i].packs.dat[j] = 0;
 node[i].packs.ctr = 0;
 for (j = 0;j < 4;j++)
   node[i].packs.arb[j] = 0;
 node[i].packs.sof = 0;
 node[i].packs.isp = 0;
 return;
}
```

```
/* This function performs initialization of the receiver packet when a new message is
created. */
void packr_init(int R)
{
 int j, i;
 i = R;
 node[i].packr.eof = 0;
 node[i].packr.ack = 0;
 node[i].packr.crc[0] = 0;
 node[i].packr.crc[1] = 0;
 for (j = 0;j < 8;j++)
  node[i].packr.dat[j] = 0;
 node[i].packr.ctr = 0;
 for (j = 0;j < 4;j++)
  node[i].packr.arb[j] = 0;
 node[i].packr.sof = 0;
 node[i].packr.isp = 0;
 return;
}


/* This function performs initialization of the simulation parameters when a new run is
started up. */
void sys_init()
{
int i, j, k, temp;
 ones = 0;
 zeros = 0;
 count = 0;
 prv_bit = 1;
 idle_time = 0;
 busy_time = 0;
 response_time = 0;
 slack_time = 0;
 collisions = 0;
 losers = 0;
```

```
latency = 0;
remote = 0;
missed = 0;
error_overhead = 0;
msg_time = 0;
errors = 0;
total_msgs = 0;
pos = 0;
tic = 0;
retransmitted = 0;
transmitted = 0;
data_frame = 1;
standard = 1;
sample_count = 1;
overload_count = 0;
form_count = 0;
ack_count = 0;
crc_count = 0;
/* total simulation time is represented in terms of bit times */
/* transmission speed is assumed to be 250 kbps */
finish = (bandwidth) * ((simulation_time * 1000)/4);
sample_period = finish / SAMPLES;
sample_time = sample_period;
/* periodic error rate */
error_period = (error_period * 1000) / 4;
periodic_error = error_period;
random_error = rand_error(tic);
bus = 1;
bus_flag = IDLE;
for (i = 0;i < MAX_NODES;i++) {
  for (j = 0;j < M_MSGS; j++) {
    node[i].msg[j].data_len = 0;
    node[i].msg[j].period = 0;
    node[i].msg[j].release = 0;
    node[i].msg[j].deadline = 0;
    node[i].msg[j].trans_time = 0;
```

```
      node[i].msg[j].prior = 0;
      node[i].msg[j].arb_lost = 0;
      node[i].msg[j].error_flag = 0;
      }
    packs_init(i);
    packr_init(i);
    node[i].no_of_msgs = 0;
    for (j = 0;j < MAX_OBJS;j++)
      node[i].object[j] = 0;
    node[i].curr_msg = 0;
    node[i].prv_bus = 1;
    node[i].prv_bus_flag = IDLE;
    node[i].transfer = NO;
    node[i].receive = YES;
    node[i].bit_val = 1;
    node[i].ovrhd_flag = 0;
    node[i].ovrhd_delim = 0;
    node[i].err_flag = 0;
    node[i].err_delim = 0;
    node[i].status = ACTIVE;
    node[i].recv_err_cnt = 0;
    node[i].trans_err_cnt = 0;
    node[i].trans_count = 0;
    node[i].lost_count = 0;
    node[i].error_count = 0;
    node[i].dead_count = 0;
    }
}


/*  Statistics for the simulation at specified sampling points and at end of the simulation
run. */
void statistics()
{
  int i, denom;
  float Total_busy_time, Total_time;
```

```
float Idle_time, Busy_time, Error_overhead;
float Response_time, Slack_time, Latency;
float Throughput, Load, Success;
if (sample_count == 1) {
  fprintf(st,"\n\tNumber of nodes = %d\t\t",total_nodes);
  fprintf(st,"Number of messages = %d\n\n",total_msgs);
  fprintf(st,"\t..............................");
  fprintf(st,"..............................\n");
  }
fprintf(st, "\n");
fprintf(st,"\t-------------------------------");
fprintf(st,"-------------------------------\n");
fprintf(st,"\tNetwork Statistics\t\t");
fprintf(st,"Sampling Point %d at %d ms\n",sample_count, (tic*4)/1000);
fprintf(st,"\t-------------------------------");
fprintf(st,"-------------------------------\n");
fprintf(st,"\n\tTotal number of messages transmitted\t= %10d\n",  transmitted);
fprintf(st,"\n\tTotal number of remote messages \t= %10d\n", remote);
fprintf(st,"\n\tTotal number of collisions \t\t= %10d\n", collisions);
fprintf(st,"\n\tTotal number of msgs losing arbitration\t= %10d\n",losers);
fprintf(st,"\n\tTotal number of errors encountered \t= %10d\n", errors);
fprintf(st,"\n\tTotal number of overload errors \t= %10d\n",overload_count);
fprintf(st,"\n\tTotal number of acknowledgement errors \t= %10d\n", ack_count);
fprintf(st,"\n\tTotal number of form errors \t\t= %10d\n",form_count);
fprintf(st,"\n\tTotal number of crc errors \t\t= %10d\n",crc_count);
fprintf(st,"\n\tTotal number of msgs resent\t\t= %10d\n",retransmitted);
Idle_time = ((float)idle_time * 4.0) / 1000.0;
fprintf (st,"\n\tIdle time in the network \t\t= %10.2f ms\n",Idle_time);
Busy_time = ((float)busy_time * 4.0) / 1000.0;
fprintf(st,"\n\tBusy time in the network \t\t= %10.2f ms\n", Busy_time);
Error_overhead = ((float)error_overhead * 4.0) / 1000.0;
fprintf(st,"\n\tError overhead time\t\t\t= %10.2f ms\n",Error_overhead);
Response_time = (((float)response_time*4.0)/1000.0)/(float)transmitted;
fprintf(st,"\n\tAverage response time\t\t\t= %10.2f ms\n",Response_time);
Slack_time = (((float)slack_time * 4.0)/1000.0)/(float)transmitted;
fprintf(st,"\n\tAverage slack time\t\t\t= %10.2f ms\n",Slack_time);
```

```
Latency = (((float)latency * 4.0) / 1000.0) / (float) transmitted;
fprintf(st,"\n\tAverage latency time\t\t\t= %10.2f ms\n",Latency);
Total_busy_time = Busy_time + Error_overhead;
Total_time = Busy_time + Error_overhead + Idle_time;
fprintf(st,"\n\tSimulation time\t\t\t\t= %10.2f ms\n",Total_time);
Load = (Total_busy_time / Total_time) * 100.0;
fprintf(st,"\n\tNetwork load \t\t\t\t= %10.2f %%\n", Load);
Throughput = ((float)transmitted / Total_time) * 1000.0;
fprintf(st,"\n\tNetwork throughput\t\t\t= %10.2f msgs/s\n",Throughput);
fprintf(st,"\n\t-------------------------------");
fprintf(st,"-------------------------------\n");
fprintf(st,"\n\tNode Statistics\n");
fprintf(st,"\t---------------\n\n");
fprintf(st,"\t-------------------------------");
fprintf(st,"---------------\n");
fprintf(st,"\tNode      No of  No of  No of     Percent  \n");
fprintf(st,"\t          msgs   arbits deadlines sucess   \n");
fprintf(st,"\t          sent   lost   missed    in trans.\n");
fprintf(st,"\t-------------------------------");
fprintf(st,"---------------\n");
for (i = 0;i < total_nodes;i++) {
  fprintf(st,"\n\t%-10s",node[i].node_name);
  fprintf(st,"%5d",node[i].trans_count);
  fprintf(st,"%8d",node[i].lost_count);
  fprintf(st,"%8d",node[i].dead_count);
  denom = node[i].trans_count + node[i].error_count;
  if (denom != 0) {
    Success = (float)node[i].trans_count / (float)denom;
    fprintf(st,"%11.2f",Success);
    }
  else
    fprintf(st,"      --");
  }
fprintf(st,"\n\t-------------------------------");
fprintf(st,"---------------\n");
}
```

```
/* This function gives a random point at which an error may be generated within the
simulation. The random generator function uses the linear congruential algorithm. The
seed value is specified by the global simulaiton clock 'tic'. */
int rand_error(long seed)
{
 int rand_val;
 srand48 (seed);
 rand_val = (int)(drand48() * 1000000) % rand_rate;
 return(rand_val);
}


/* This function generates addresses for the nodes. */
void node_addressing()
{
 int i;
 unsigned char base_address;
 base_address = 0;
 for (i = 0;i < total_nodes;i++) {
  node[i].address = base_address + i;
  }
 return;
}


/* This function obtains the input parameters for the simulation from an input file named
input#, where # gives the order of the file. The parameters include node, and message
data such as node name, number of objects, number of messages, message name, message
period, message release time, message priority, message data length, type of format
(standard/extended), message mode (data/remote), and node objects. */
void get_parm()
{
 int i, j, num;
 static char line[82];
 total_nodes = 0;
 i = 0;
```

```
  while (fgets(line, 80, ip) != NULL) {
   sscanf(line,"%s %d %d",&node[i].node_name,&node[i].no_of_msgs,
        &node[i].no_of_objs);
   num = node[i].no_of_msgs;
   for (j = 0;j < num;j++) {
    fgets(line, 80, ip);
    sscanf(line, "%s %d %d %d %d %d %d %d", &node[i].msg[j].msg_name,
        &node[i].msg[j].period, &node[i].msg[j].release, &node[i].msg[j].prior,
        &node[i].msg[j].data_len, &node[i].msg[j].msg_format,
        &node[i].msg[j].msg_mode);
    node[i].msg[j].period = (node[i].msg[j].period * 1000)/ 4;
    }
   for (j = 0;j < node[i].no_of_objs;j++) {
    fgets(line, 80, ip);
    sscanf(line, "%d", &node[i].object[j]);
    }
   total_nodes++;
   i++;
   }
}


/*  This function receives the bit sent over the CAN bus.  It is implemented in such a way
that all nodes receive the message.  The receivers detect and signal errors to the
transmitter, to initiate a retransmission. */
int receive(int r_bit, int br, int indr, int nr)
{
 int j;
 unsigned arr_crc = 0, check;
 switch (pos) {
  case 0:  /* reception of the interframe space bits */
          /* Overload condition if less than 3 recessive bits */
        if (r_bit != 1) {
         return(OVERLOAD_ERROR);
         }
        node[nr].packr.isp |= (r_bit << br);
```

```
        return(SUCCESS);
        break;

    case 1:  /* reception of the SOF bit */
          /* form violation if a recessive SOF is sent */
        if (r_bit != 0) {
         return(FORM_ERROR);
         }
        node[nr].packr.sof |= (r_bit << br);
        return(SUCCESS);
        break;

    case 2:  /* reception of the arbitration bits */
        node[nr].packr.arb[indr] |= (r_bit << br);
          /* remote message sensing */
        if (!data_frame &&
          (((standard) && (indr == 2) && (br == 4) && (r_bit == 1)) ||
          ((!standard) && (indr == 0) && (br == 0) && (r_bit == 1))))
         for (j = 0;j < node[nr].no_of_objs;j++)
          if (node[nr].object[j] == node[n].address) {
           node[nr].msg[j].deadline = tic + 150;
           return(SUCCESS);
           }
        return(SUCCESS);
        break;

    case 3:  /* reception of the control bits */
        node[nr].packr.ctr |= (r_bit << br);
        return(SUCCESS);
        break;

    case 4:  /* reception of the data bits */
        node[nr].packr.dat[indr] |= (r_bit << br);
        return(SUCCESS);
        break;
```

```
case 5:  /* reception of the CRC sequence bits */
        node[nr].packr.crc[indr] |= (r_bit << br);
         /* CRC sequence check by receivers */
        if ((indr == 0) && (br == 0)) {
         arr_crc = node[nr].packr.crc[0];
         arr_crc |= (node[nr].packr.crc[1] << 8);
         check = crc_check(nr);
         if (check != arr_crc) {
          return(CRC_ERROR);
          }
         }
        return(SUCCESS);
        break;

case 6:  /* reception of the acknowledgement bits */
        if (br == 0)
         node[nr].packr.ack |= (r_bit << br);
        else {
         /* acknowledgement posting by receivers */
         node[nr].packr.ack |= (0 << br);
         if (node[nr].recv_err_cnt != 0) { /* fault confinement */
          node[nr].recv_err_cnt--;
          if ((node[nr].trans_err_cnt < 128) &&
            (node[nr].recv_err_cnt < 128))
          node[nr].status = ACTIVE;
          }
         }
        /* negative acknowledgement detected by the transmitter */
        if ((br == 0) && (node[nr].packr.ack != 1)) {
         return(ACK_ERROR);
        }
        return(SUCCESS);
        break;

case 7:  /* reception of the EOF bits */
         /* form violation in EOF bits with
```

```
          the detection of dominant bit */
        if (r_bit != 1) {
          return(FORM_ERROR);
        }
        node[nr].packr.eof |= (r_bit << br);
        return(SUCCESS);
        break;
        }
  return(OVER); /* end of message reception */
  }
```

```
/* This routine is the core of the bit by bit simulation. It represents a dominant bit on the
bus with a logical 0, and a recessive bit with a logical 1. It also creats an error condition
by complementing the true vale at the appropriate error time. */
int get_bit(unsigned char g_val, int bits)
{
 if (g_val & (1 << bits)) {
   if (tic == random_error) {
     random_error = tic + rand_error(tic);
     return(0); /* send errorneous bit */
     }
   if (tic == periodic_error) {
     periodic_error += error_period;
     return(0); /* send errorneous bit */
     }
   return(1); /* send the correct bit */
   }
 else {
   if (tic == random_error) {
     random_error = tic + rand_error(tic);
     return(0); /* send errorneous bit */
     }
   if (tic == periodic_error) {
     periodic_error += error_period;
     return(0); /* send errorneous bit */
```

```
    }
   return(0); /* send the correct bit */
   }
}
```

/* This function transmits a bit over the bus. The sender station transmits bit by bit. The sender station also monitors the CAN bus for potential errors during transmission. Successful transmission of a bits continues unless an error condition is detected. */

```
int transmit(int n)
{
 int i, b, bit_val, bit_flag;
 unsigned char val;
 /* bus is held by the current transmitter */
 node[n].prv_bus_flag = BUSY;
 bus_flag = BUSY;

 while (pos <= 8) {
  switch (pos) {
   case 0: /* transmission of interframe space bits */
         val = node[n].packs.isp;
         for (b = 2;b >= 0;b--) {
         bit_val = get_bit(val, b);
         if ((bit_flag = msg_filter(bit_val,b,0,0)) != SUCCESS)
          return(bit_flag);
         }
         printf("  isp %2X\n", node[n].packr.isp);
         pos++;
         break;

   case 1: /* transmission of SOF bit */
         val = node[n].packs.sof;
         bit_val = get_bit(val, 0);
         if ((bit_flag = msg_filter(bit_val,b,0,0)) != SUCCESS)
          return(bit_flag);
         printf("  sof %2X\n", node[n].packr.sof);
```

```
        pos++;
        break;

    case 2:  /* transmission of arbitration bits */
        for (i = 3;i >= 0;i--) {
          val = node[n].packs.arb[i];
          for (b = 7;b >= 0;b--) {
            bit_val = get_bit(val, b);
            if ((bit_flag = msg_filter(bit_val,b,i,0)) != SUCCESS)
              return(bit_flag);
            }
          printf("  arb %d  %02X\n",i, node[n].packr.arb[i]);
          }
        pos++;
        break;

    case 3:  /* transmission of control bits */
        val = node[n].packs.ctr;
        for (b = 5;b >= 0;b--) {
          if ((bit_flag = msg_filter(bit_val,b,0,0)) != SUCCESS)
            return(bit_flag);
          }
        printf("  ctr %02X\n", node[n].packr.ctr);
        pos++;
        break;

    case 4:  /* transmission of data bits */
        for (i = 7;i >= 0;i--) {
          val = node[n].packs.dat[i];
          for (b = 7;b >= 0;b--) {
            bit_val = get_bit(val, b);
            if ((bit_flag = msg_filter(bit_val,b,i,0)) != SUCCESS)
              return(bit_flag);
            }
          printf("  dat %d  %02X\n",i, node[n].packr.dat[i]);
          }
```

```
        pos++;
        break;

    case 5:  /* transmission of CRC sequence bits */
        for (i = 1;i >= 0;i--) {
          val = node[n].packs.crc[i];
          for (b = 7;b >= 0;b--) {
            bit_val = get_bit(val, b);
            if ((bit_flag = msg_filter(bit_val,b,i,0)) != SUCCESS)
              return(bit_flag);
            }
          printf("  crc %d  %02X\n",i, node[n].packr.crc[i]);
          }
        pos++;
        break;

    case 6:  /* transmission of acknowledgment bits */
        val = node[n].packs.ack;
        for (b = 1;b >= 0;b--) {
          bit_val = get_bit(val, b);
          if ((bit_flag = msg_filter(bit_val,b,0,0)) != SUCCESS)
            return(bit_flag);
          }
        printf("  ack %2X\n", node[n].packr.ack);
        pos++;
        break;

    case 7:  /* transmission of EOF bits */
        val = node[n].packs.eof;
        for (b = 6;b >= 0;b--) {
          bit_val = get_bit(val, b);
          if ((bit_flag = msg_filter(bit_val,b,0,0)) != SUCCESS)
            return(bit_flag);
          }
        printf("  eof %2X\n", node[n].packr.eof);
        pos++;
```

```
            break;

        case 8:  /* end of message transmission */
            return(OVER);
      }
    }
}
```

```
/* This function checks if the current time is a sampling point, and if so, the statistics
routine is invoked to compute and output the statistics at that point in time. */
void sample()
{
  if (tic == sample_time) {
    statistics();
    sample_time += sample_period;
    sample_count++;
    }
}
```

```
/* This module simulates the bit stuffing function by adding a bit time whenever 5
consecutive bits of equal value are detected. */

void bit_stuff(int bit_rd)
{
  if (bit_rd == 1) {
    if (prv_bit == 1) {
      ones++;  /* track recessive bits */
      if (ones > 5) {
        msg_time++;
        tic++;  /* a complement bit is stuffed */
        sample();
        ones = 0;
        }
      }
```

```
    else {
     zeros = 0;
     prv_bit = 1;
     }
    }
  else {
    if (prv_bit == 0) {
     zeros++;  /* track dominant bits */
     if (zeros > 5) {
      msg_time++;
      tic++;  /* a complement bit is stuffed */
      sample();
      zeros = 0;
      }
     }
    else {
     ones = 0;
     prv_bit = 0;
     }
    }
  return;
}
```

```
/* This function performs message filtering within the CAN nodes. The broadcast bit is
sent to nodes that find a match with their communication objects. */
int msg_filter(int bit_val, int bm, int indm, int nm)
{
  int bit_read;
  while (nm < total_nodes) {
    if (node[nm].receive == YES) {
     bit_read = receive(bit_val,bm,indm,nm);
     if (bit_read != SUCCESS) {
      node[nm].recv_err_cnt++;
      if (node[nm].recv_err_cnt >= 128)
       node[nm].status = PASSIVE;
```

```
      return(bit_read);
      }
     }
    nm++;
    }
  if (!(((pos == 4) && (indm < (8 - node[n].msg[m].data_len))) || ((pos == 2) &&
        (standard) && ((indm < 2) || ((indm == 2) && (bm < 4)))))) {
    msg_time++;
    /* global simulation clock that keeps ticking at each bit transmission */
    tic++;
    sample();
    bit_stuff(bit_read);
    }
  return(SUCCESS);
}


/*  This function computes the CRC sequence for the frame at the receiving station.  The
CRC sequence is generated by a polynomial division algorithm, using a 15-bit shift
register. */
int crc_check(int nr)
{
 int i, j, k;
 unsigned crc_seq;
 unsigned char crc_nxt;
 unsigned char nxt_bit;
 unsigned char msb_bit;
 crc_seq = 0;
 msb_bit = 0;
 nxt_bit = 0;
 msb_bit = (1 & (crc_seq >> 14));
 crc_nxt = node[nr].packr.sof ^ msb_bit;
 crc_seq <<= 1;
 crc_seq &= 0x00007fff;
 if (crc_nxt)
  crc_seq ^= 0x4599;
```

```
if (standard)
 k = 2;
else
 k = 0;

for (i = 3;i >= k;i--) {
 for (j = 7;j >= 0;j--) {
   if (node[nr].packr.arb[i] & (1 << j))
     nxt_bit = 1;
   else
     nxt_bit = 0;
   if (crc_seq & (1 << 14))
     msb_bit = 1;
   else
     msb_bit = 0;
   crc_nxt = nxt_bit ^ msb_bit;
   crc_seq <<= 1;
   crc_seq &= 0x00007fff;
   if (crc_nxt)
     crc_seq ^= 0x4599;
   /*  accept the first 12 bits for a standard frame */
   if ((standard) && (i == 2) && (j == 4))
     break;
   }
 }

for (i = 7;i >= (8 - node[n].msg[m].data_len);i--) {
 for (j = 7;j >= 0;j--) {
   if (node[nr].packr.dat[i] & (1 << j))
     nxt_bit = 1;
   else
   nxt_bit = 0;
   if (crc_seq & (1 << 14))
     msb_bit = 1;
   else
     msb_bit = 0;
```

```
    crc_nxt = nxt_bit ^ msb_bit;
    crc_seq <<= 1;
    crc_seq &= 0x00007fff;
    if (crc_nxt)
     crc_seq ^= 0x4599;
    }
   }
  crc_seq |= 1;
  return(crc_seq);
}
```

```
/* This function performs arbitration during bus contention by more than one message.
The first 6 bits of the arbitration field are used to determine the winner, depending on
their priorities.  A dominant bit overrides a recessive bit during arbitration.  The station
that detects the bus value to be different from its bit value backs off from transmission. */
int arbitrate()
{
  int i, j, k, b, bus_val = 1, bit_flg, l;
  unsigned char val;
  pos = 0;
  val = node[n].packs.isp;
  for (b = 2;b >= 0;b--) {
    node[n].bit_val = get_bit(val, b);
    if ((bit_flg = msg_filter(node[n].bit_val,b,0,0)) != SUCCESS) {
      for (l = 0;l < total_nodes;l++)
        if (node[l].transfer == YES)
          node[l].msg[node[l].curr_msg].error_flag= YES;
      return(bit_flg);
      }
    }
  printf("\n  ISP %2X\n", node[n].packr.isp);
  pos++;
  val = node[n].packs.sof;
  node[n].bit_val = get_bit(val, 0);
  if ((bit_flg = msg_filter(node[n].bit_val,0,0,0)) != SUCCESS) {
```

```
  for (l = 0;l < total_nodes;l++)
   if (node[l].transfer == YES)
    node[l].msg[node[l].curr_msg].error_flag= YES;
  return(bit_flg);
  }
printf("  SOF %2X\n", node[n].packr.sof);
pos++;
fprintf(op,"\t(Message,Node) = ");
for (j = 3;j >= 0;j--) {
  for (b = 7;b >= 0;b--) {
   /* all stations place their bit value on the bus */
   for(i = 0;i < total_nodes;i++)
    if (node[i].transfer == YES) {
     val = node[i].packs.arb[j];
     node[i].bit_val = get_bit(val, b);
     bus_val &= node[i].bit_val;
     }
   /* every station checks if the bit it placed on the bus is the same as the bit that is being
       transmitted. */
   for(i = 0;i < total_nodes;i++)
    if (node[i].transfer == YES) {
     if ((node[i].bit_val != bus_val) && (node[i].bit_val == 1)) {
      node[i].transfer = NO;
      k = node[i].curr_msg;
      node[i].msg[k].arb_lost = YES;
      node[i].lost_count++;
      fprintf(op,"(%d,%d), ",k,i);
      losers++;
      count--;
      }
     else {
      n = i;
      }
     }

   if((bit_flg=msg_filter(node[n].bit_val,b,j,0))!=SUCCESS) {
```

```
      for (l = 0;l < total_nodes;l++)
        if (node[l].transfer == YES)
          node[l].msg[node[l].curr_msg].error_flag= YES;
      return(bit_flg);
      }
    bus_val = 1;
    }
   printf("  ARB %d  %02X\n",j, node[n].packr.arb[j]);
   }
  fprintf(op,"lost arbitration by time %d\n\n",tic);
  /* more than one message has the same priority assigned to it */
  if (count > 1) {
   printf("\n Error in priority assignment, Quits\n");
   exit(0);
   }
  return(SUCCESS);
}


/*  This function transmits an overload frame when an overload error occurs. */
int send_overload_frame()
{
 int i, b, bit_val;
 unsigned char val;
 val = 0x00;
 for (b = 5;b >= 0;b--) { /* six overload flags */
  bit_val = get_bit(val, b);
  if (bit_val)
   return(FAILURE);
  for (i = 0;i < total_nodes;i++)
   if (node[i].receive == YES)
    node[i].ovrhd_flag |= (bit_val << b);
  tic++;
  sample();
  msg_time++;
  }
```

```
 printf("\n  ovld_flag %2X\n", node[n].ovrhd_flag);
 val = 0xff;
 for (b = 7;b >= 0;b--) { /* eight overload delimiters */
  bit_val = get_bit(val, b);
  if (!bit_val)
   return(FAILURE);
  for (i = 0;i < total_nodes;i++)
   if (node[i].receive == YES)
    node[i].ovrhd_delim |= (bit_val << b);
  tic++;
  sample();
  msg_time++;
  }
 printf("  ovld_delim %2X\n\n", node[n].ovrhd_delim);
 return(SUCCESS);
}
```

```
/* This function transmits an error frame when an ACK error, CRC error, form error, or
bit error occurs. */
int send_error_frame(int ne)
{
 int i, b, bit_val;
 unsigned char val;
 /* transmitter sending error frame */
 if ((ne == n) && (node[n].status == ACTIVE)) {
  node[n].trans_err_cnt += 8;
  if (node[n].trans_err_cnt >= 128) /* error passive node */
   node[n].status = PASSIVE;
  if (node[n].trans_err_cnt >= 256) /* faulty node */
   node[n].status = BUS_OFF;
  }
 if (node[ne].status == ACTIVE)
  val = 0x00;
 else
  val = 0x37;
```

```
  for (b = 5;b >= 0;b--) { /* six error flags */
   bit_val = get_bit(val, b);
   for (i = 0;i < total_nodes;i++)
    if (node[i].receive == YES)
     node[i].err_flag |= (bit_val << b);
   if (node[ne].err_flag != val)
    return(FAILURE);
   tic++;
   sample();
   msg_time++;
   }
  printf("\n  err_flag %2X\n", node[n].err_flag);

  val = 0xff;
  for (b = 7;b >= 0;b--) { /* eight error delimiters */
   bit_val = get_bit(val, b);
   /* receiver detects the first bit to be dominant */
   if ((!bit_val) && (b == 8)) {
    node[nr].recv_err_cnt += 8;
    if (node[nr].recv_err_cnt >= 128)
     node[nr].status = PASSIVE;
    return(FAILURE);
    }
   for (i = 0;i < total_nodes;i++)
    if (node[i].receive == YES)
     node[i].err_delim |= (bit_val << b);
   tic++;
   sample();
   msg_time++;
   }
  printf("  err_delim %2X\n", node[n].err_delim);
  return(SUCCESS);
}
```

```
/* This module performs the initiation of a message transfer. The message may contain a
data frame, remote frame, error frame, or an overload frame. Transmission is completed
successfully or an error condition is reported. */
void msg_transfer(int mode)
{
 int i, k;
 while (mode < 6) {
 switch (mode) {

    case 0: /* action after a successful message transfer */
          fprintf(op,"\tMessage %d of Node %d TRANSMITTED ",m,n);
          fprintf(op,"at time %d\n\n",tic);
          busy_time += msg_time;
          response_time += (tic - msg_time) - node[n].msg[m].deadline;
          msg_time = 0;
          packs_init(n);
          for (i = 0;i < total_nodes;i++) {
           packr_init(i);
           }
          node[n].trans_count++;
          node[n].transfer = NO;
          node[n].prv_bus = 1;
          node[n].prv_bus_flag = IDLE;
          node[n].msg[m].error_flag = NO;
          node[n].msg[m].deadline += node[n].msg[m].period;
          slack_time += node[n].msg[m].deadline - tic;
          /* check node status */
          if (node[n].trans_err_cnt != 0) {
           node[n].trans_err_cnt--;
           if ((node[n].trans_err_cnt < 128) && .
               (node[n].recv_err_cnt < 128))
             node[n].status = ACTIVE;
           }
          if (node[n].msg[m].deadline < tic) {
           fprintf(op,"\n\tMsg %d of node %d MISSED deadline by ", m, n);
           fprintf(op,"%d bit times\n\n", tic - node[n].msg[m].deadline);
```

```
          node[n].dead_count++;
          latency += tic - node[n].msg[m].deadline;
          missed++;
          }
        if (data_frame)
          transmitted++;
        else
          remote++;
        return;

case 1:  /* initiation of a data / remote transfer */
        mode = transmit(n);
        break;

case 2:  /* action after an overload error occurs */
        for (i = 0;i < total_nodes;i++)
          packr_init(i);
        retransmitted++;
        errors++;
        overload_count++;
        fprintf(op,"\tOVERLOAD ERROR in Message %d of Node %d ", m, n);
        fprintf(op,"at time %d\n\n",tic);
        node[n].error_count++;
        node[n].msg[m].error_flag = YES;
        if (!send_overload_frame()) {
          fprintf(op,"\tError in Overload frame at %d\n\n", tic);
          mode = 3;
          break;
          }
        else {
          error_overhead += msg_time;
          msg_time = 0;
          return;
        }
```

```
case 3:  /* action after a form error occurs */
        for (i = 0;i < total_nodes;i++)
          packr_init(i);
        retransmitted++;
        errors++;
        form_count++;
        fprintf(op,"\tFORM ERROR in Message %d of Node %d ",m, n);
        fprintf(op,"at time %d\n\n",tic);
        node[n].error_count++;
        node[n].msg[m].error_flag = YES;
        if (!send_error_frame(nr)) {
          fprintf(op,"\tError in ERROR frame at %d\n\n", tic);
          mode = 2; /* Error in Error frame */
          break;
          }
        else {
          error_overhead += msg_time;
          msg_time = 0;
          return;
          }

case 4:  /* action after a CRC error occurs */
        for (i = 0;i < total_nodes;i++)
          packr_init(i);
        retransmitted++;
        errors++;
        crc_count++;
        fprintf(op,"\tCRC ERROR in Message %d ", m);
        fprintf(op,"of Node %d at time %d\n\n", n, tic);
        node[n].error_count++;
        node[n].msg[m].error_flag = YES;
        if (!send_error_frame(nr)) {
          fprintf(op,"\tError in ERROR frame at %d\n\n", tic);
          mode = 2; /* Error in Error frame */
          break;
          }
```

```
        else {
          error_overhead += msg_time;
          msg_time = 0;
          return;
          }

    case 5:  /* action after a ACK error occurs */
        for (i = 0;i < total_nodes;i++)
          packr_init(i);
        retransmitted++;
        errors++;
        ack_count++;
        fprintf(op,"\tACK ERROR in Message %d Node %d ",m,n);
        fprintf(op,"at time %d\n\n",tic);
        node[n].error_count++;
        node[n].msg[m].error_flag = YES;
        if (!send_error_frame(n)) {
          fprintf(op,"\tError in ERROR frame at %d\n\n", tic);
          mode = 2; /* Error in Error frame */
          break;
          }
        else {
          error_overhead += msg_time;
          msg_time = 0;
          return;
          }

    default:  printf("\n Error in message transfer mode, Quits \n ");
        exit(0);
  }
 }
}
```

```
/* This function generates a CRC sequence for the message. The CRC sequence is
computed for the SOF, arbitration, control, and data fields in that order. */
void crc_gen(int Node, int Msg)
{
 int i, j, k;
 unsigned crc_reg;
 unsigned char crc_nxt;
 unsigned char nxt_bit;
 unsigned char msb_bit;

 crc_reg = 0; /* initialize shift register */
 msb_bit = 0;
 nxt_bit = 0;
 msb_bit = (1 & (crc_reg >> 14));
 crc_nxt = node[Node].packs.sof ^ msb_bit;
 crc_reg <<= 1;
 crc_reg &= 0x00007fff;
 if (crc_nxt)
  crc_reg ^= 0x4599;
 if (standard)
  k = 2;
 else
  k = 0;

 for (i = 3;i >= k;i--) {
  for (j = 7;j >= 0;j--) {
   if (node[Node].packs.arb[i] & (1 << j))
    nxt_bit = 1;
   else
    nxt_bit = 0;
   if (crc_reg & (1 << 14))
    msb_bit = 1;
   else
    msb_bit = 0;
   crc_nxt = nxt_bit ^ msb_bit;
   crc_reg <<= 1;
```

```
      crc_reg &= 0x00007fff;
      if (crc_nxt)
        crc_reg ^= 0x4599;
      /* stop after 12 th bit for standard frames */
      if ((standard) && (i == k) && (j == 4))
        break;
      }
    }

  for (i = 7;i >= (8 - node[Node].msg[Msg].data_len);i--) {
    for (j = 7;j >= 0;j--) {
      if (node[Node].packs.dat[i] & (1 << j))
        nxt_bit = 1;
      else
        nxt_bit = 0;
      if (crc_reg & (1 << 14))
        msb_bit = 1;
      else
        msb_bit = 0;
      crc_nxt = nxt_bit ^ msb_bit;
      crc_reg <<= 1;
      crc_reg &= 0x00007fff;
      if (crc_nxt)
        crc_reg ^= 0x4599;
      }
    }
  node[Node].packs.crc[0] = crc_reg & 0x00ff;
  node[Node].packs.crc[1] = (crc_reg & 0xff00) >> 8;
  node[Node].packs.crc[0] |= 1; /* crc delimiter */
  return;
}
```

```
/* This function generates a packet for each message that arrives at a node. A packet is
created in conformance with the frame format in the CAN 2.0 version. Both standard and
extended frames are developed. The basic structure is that of a extended frame. Standard
frames are built over the extended frame. */
void packet_gen(int node_no, int msg_no)
{
 int i, j;
 /* interframe space consists of 3 recessive bits */
 node[node_no].packs.isp |= 0x7;
 printf("\n isp = %2X", node[node_no].packs.isp);
 /* start of frame is a single dominant bit */
 node[node_no].packs.sof = 0;
 printf("\n sof = %2X", node[node_no].packs.sof);
 /* following 5 bits are used to represent node address */
 node[node_no].packs.arb[3] |= (node[node_no].address >> 3);
 /* first 6 arbitration bits are used for priority */
 node[node_no].packs.arb[3] |= (node[node_no].msg[msg_no].prior << 2);
 node[node_no].packs.arb[2] |= (node[node_no].address << 5);

 if (standard) {
  if (data_frame)
   node[node_no].packs.arb[2] |= (0x0); /* RTR bit is dominant*/
  else /* if remote frame */
   node[node_no].packs.arb[2] |= (0x1 <<4);/*RTR 12th bit*/
  node[node_no].packs.arb[2] |= (0x0f); /* 13th onward bits*/
  node[node_no].packs.arb[1] |= (0xff);
  node[node_no].packs.arb[0] |= (0xff);
  }
 else { /* extended format */
  node[node_no].packs.arb[2] |= (0x1 << 4); /* SRR bit */
  node[node_no].packs.arb[2] |= (0x1 << 3); /* IDE bit */
  node[node_no].packs.arb[1] &= (0x00); /* extended ID to be set*/
  node[node_no].packs.arb[0] &= (0x00);
  if (data_frame)
   node[node_no].packs.arb[0] |= (0x0); /* RTR bit domi*/
  else /* if remote frame */
```

```
    node[node_no].packs.arb[0] |= (0x1);/*RTR 32th bit*/
   }
  printf("\n arb = ");
  for (i = 3;i >= 0;i--)
  printf("%02X", node[node_no].packs.arb[i]);
  /* last 4 control bits give the binary value of data length in bytes */
  if (standard)
    node[node_no].packs.ctr |= (0x1 << 4); /* IDE and r0 bits */
  else
    node[node_no].packs.ctr |= (0x0 << 4); /* r0 and r1 bits */
  node[node_no].packs.ctr |= node[node_no].msg[msg_no].data_len;
  printf("\n ctr = %02X", node[node_no].packs.ctr);

  /* data bytes are generated randomly */
  for (i = 7;i >= (8-node[node_no].msg[msg_no].data_len);i--) {
    node[node_no].packs.dat[i] = (rand() % 256);
   }
  for (i = (7-node[node_no].msg[msg_no].data_len);i >= 0;i--)
    node[node_no].packs.dat[i] = 0xff; /* dont care bytes */
    printf("\n dat = ");
  for (i = 7;i >= 0;i--)
    printf("%02X", node[node_no].packs.dat[i]);

  /* a 15-bit CRC sequence is obtained */
  crc_gen(node_no, msg_no);
  printf("\n crc = %2X",node[node_no].packs.crc[1]);
  printf("%02X\n",node[node_no].packs.crc[0]);

  /* ack bits are recessive before transmission  */
  node[node_no].packs.ack |= 3;
  printf(" ack = %2X\n", node[node_no].packs.ack);
  /* 7 end of frame bits are recessive */
  node[node_no].packs.eof |= 0x7f;
  printf(" eof = %02X\n", node[node_no].packs.eof);
  return;
}
```

/* This function performs the message cycle. It checks for new message arrivals, initiates arbitration if more than one message has arrived, then calls the message transfer routine to transmit the message. */

```
void msg_cycl()
{
 int i, j, k, l;
 static unsigned long int next_arrival = 0;
 int mode, filter;
 long int next;
 int IS_THERE_A_MSG;
 bus_flag = IDLE;
 transmitted = 0;
 pos = 0;
 /* initial message deadlines are their release times */
 for (i = 0;i < total_nodes;i++)
  for (j = 0;j < node[i].no_of_msgs;j++)
   node[i].msg[j].deadline = node[i].msg[j].release;

 /* cycle until end of the simulation run */
 while (tic <= finish) {
  /* check each node for message arrivals */
  for (j = 0;j < total_nodes;j++) {
   next = 100000000;
   IS_THERE_A_MSG = NO;
   for (l = 0;l < node[j].no_of_msgs;l++)
    if (node[j].msg[l].deadline <= next) {
     IS_THERE_A_MSG = YES;
     k = l;
     next = node[j].msg[l].deadline;
     }
   /* process each node message */
   if ((IS_THERE_A_MSG == YES) &&
     (node[j].msg[k].deadline <= tic)) {
    node[j].transfer = YES;
    node[j].curr_msg = k;
    if (node[j].msg[k].msg_mode)
```

```
      data_frame = 1;
    else
      data_frame = 0; /* remote request */
    if (node[j].msg[k].msg_format == 1)
      standard = 1;
    else
      standard = 0; /* extended frame format*/
    /* generate a packet is message is already not there */
    if ((node[j].msg[k].arb_lost != YES) &&
       (node[j].msg[k].error_flag != YES))
      packet_gen(j, k);
    n = j;
    m = k;
    count++;  /* number of messages arrivals */
    }
  }
msg_time = 0;
mode = 1;
/* arbitrate to resolve bus contention */
if (count > 1) {
  collisions++;
  mode = arbitrate(count);
  m = node[n].curr_msg;
  pos = 3;
  }
node[n].msg[m].arb_lost = NO;
if (node[n].msg[m].msg_format == 1)
  standard = 1;
else
  standard = 0; /* extended frame format*/
if (node[n].msg[m].msg_mode == 1)
  data_frame = 1;
else
  data_frame = 0; /* remote request */
/* no message has arrived, bus is idle state */
if (count == 0) {
```

```
      tic++;
      sample();
      idle_time++;
      if (tic >= periodic_error)
        periodic_error += error_period;
      if (tic >= random_error)
        random_error = tic + rand_error(tic);
      count = 0;
      }
    /* initiate a message transfer */
    else {
      count = 0;
      if ((bus_flag == IDLE) && (node[n].status != BUS_OFF)) {
        msg_transfer(mode);
        if (tic >= finish)
          return;
        bus_flag = IDLE;
        pos = 0;
        }
      }
    }
}
```

```
/*  This function performs a priority assignment using the rate monotonic priority
assignment algorithm.  Higher priorities are assigned to message with smaller periods.
The message set is also tested for the two real time constraints before assigning priorities.
*/
void prior_assign()
{
  int i, j, k, l, temp;
  float cost_fn, schedulables;
  int p_max, interval_L;

  cost_fn = 0;
  /* test for the first real time constraint */
```

```
/* total cost function is less than unity */
for (i = 0;i < total_nodes;i++)
  for (j = 0;j < node[i].no_of_msgs;j++) {
    node[i].msg[j].trans_time = (node[i].msg[j].data_len*8.0+MAX_OVERHEAD);
    cost_fn += node[i].msg[j].trans_time / node[i].msg[j].period;
    total_msgs++;
    }
printf("cost function is %f\n",cost_fn);
if (cost_fn >= 1.0) {
  printf("No schedule for this message set, Quits\n\n");
  exit(0);
  }
/* message ordering by message periods */
l = 0;
for (i = 0;i < total_nodes;i++) {
  if (node[i].no_of_msgs == 1) {
    order[l].msg = 0;
    order[l].node = i;
    order[l].dead = node[i].msg[0].period;
    order[l].trans = node[i].msg[0].trans_time;
    l++;
    }
  else {
    for (k = 0;k < node[i].no_of_msgs;k++) {
      order[l].msg = k;
      order[l].node = i;
      order[l].dead = node[i].msg[k].period;
      order[l].trans = node[i].msg[k].trans_time;
      l++;
      }
    }
  }

for (i = 0;i < total_msgs - 1;i++)
  for (j = 0;j < total_msgs - 1;j++)
    if (order[j].dead > order[j+1].dead) {
```

```
      temp = order[j].dead;
      order[j].dead = order[j+1].dead;
      order[j+1].dead = temp;
      temp = order[j].msg;
      order[j].msg = order[j+1].msg;
      order[j+1].msg = temp;
      temp = order[j].node;
      order[j].node = order[j+1].node;
      order[j+1].node = temp;
      temp = order[j].trans;
      order[j].trans = order[j+1].trans;
      order[j+1].trans = temp;
      }
  for (i = 0;i < total_msgs;i++) {
   j = order[i].node;
   k = order[i].msg;
   node[j].msg[k].prior = i;
   }

  max_period = order[total_msgs-1].dead;
  min_period = order[0].dead;
  /* test for second real time constraint */
  /* no inserted idle time */
  p_max = order[total_msgs-1].dead;
  interval_L = p_max - 10;
  schedulables = order[total_msgs-1].trans;
  for (i = total_msgs - 2;i >= 0;i--)
   schedulables += floor(((interval_L-1)/order[i].dead))*order[i].trans;
  if (interval_L < schedulables) {
   printf("No schedule for this message set, Quits\n\n");
   exit(0);
   }
}
```

```
/* This function is the main routine that controls the flow within the program. It also
invokes 5 simulation runs for 5 different message sets from input file named input#. */
main()
{
 static char buff[82];
 char infile[10], statfile[10], outfile[10];
 rand_rate = RAND_RATE;
 system("tput clear");
  sim_cnt = 1;
  while (sim_cnt <= 8) {
   sprintf(outfile, "output%1d", sim_cnt);
   if ((op = fopen(outfile,"w")) == NULL) {
    printf("Error:  %s file not created\n\n", outfile);
    exit(1);
    }
   sprintf(statfile, "statistix%1d", sim_cnt);
   if ((st = fopen(statfile,"w")) == NULL) {
    printf("Error:  %s file not created\n\n", statfile);
    exit(1);
    }
   sprintf(infile, "input%1d", sim_cnt++);
   if ((ip = fopen(infile,"r")) == NULL) {
    printf("Error:  input file %s not created\n\n", infile);
    exit(1);
    }
   fgets(buff, 80, ip);
   sscanf(buff,"%d", &simulation_time);
   printf("\n\n Simulation time %d milli seconds\n\n", simulation_time);
   fgets(buff, 80, ip);
   sscanf(buff,"%d", &bandwidth);
   printf("Bandwidth %d bits per bit time\n\n", bandwidth);
   fgets(buff, 80, ip);
   sscanf(buff,"%d", &error_period);
   printf("Error period %d error/ms\n\n", error_period);
   fprintf(st,"\t..............................");
   fprintf(st,"..............................\n");
```

```
    fprintf(st,"\n\t\t\tSTATISTICS OF SIMULATION RUN %d\n\n", (sim_cnt-1));
    fprintf(op,"\n\n\t..............................");
    fprintf(op,"..............................\n");
    fprintf(op,"\n\t\t\tEVENTS OF SIMULATION RUN %d\n\n",(sim_cnt-1));
    fprintf(op,"\t..............................");
    fprintf(op,"..............................\n\n");
    sys_init();
    get_parm();
    node_addressing();
    prior_assign();
    msg_cycl();
    fprintf(op,"\n\n\t..............................");
    fprintf(op,"..............................\n");
    fprintf(st,"\t..............................");
    fprintf(st,"..............................\n");
    printf("\n\n\n END OF SIMULATION RUN %d\n\n",(sim_cnt-1));
    fclose(ip);
    }
   fclose(st);
   fclose(op);
   system("tput clear");
  printf("\n\n\n END OF SIMULATION\n\n");
  printf("\n\n\n ADIOS ! BYE ! SAYONARA!\n\n\n\n");
  return;
}


stop()
{
 fflush(stdin);
 fflush(stdout);
 printf("\n                                        Continue ...");
 getchar();
}
```

APPENDIX C

INPUT DATA

Input file I

```
100
1
1
engine1    2     3
MSG11     10     1    31    8    1    1
MSG12     50     0    30    8    0    1
2
0
0
engine2    1     1
MSG13    250     0    29    8    1    1
0
```

Input file II

```
100
1
1
engine    3     3
MSG11     10    1     31    8     1     1
MSG12     50    0     30    8     0     1
MSG13     250   0     29    8     1     1
2
0
0
torque    1     2
MSG21     10    0     28    8     1     1
0
0
trans1    1     1
MSG31     10    0     27    8     1     0
0
```

Input file III

```
100
1
1
engine     3     3
MSG11     10     1    31    8    1    1
MSG12     50     0    30    8    0    1
MSG13    250     0    29    8    1    1
2
0
0
torque     1     2
MSG21     10     0    28    8    1    1
0
0
trans1     1     1
MSG31     10     0    27    8    1    0
0
trans2     3     1
MSG41     10     0    26    8    1    1
MSG42    100     0    25    8    1    1
MSG43   1000     0    24    8    1    1
0
brake      2     1
MSG51    100     0    23    8    1    1
MSG52   1000     0    22    8    1    1
0
retarder   2     1
MSG61    100     0    21    8    1    1
MSG62   1000     0    20    8    1    1
0
brk_ctrl   1     1
MSG71     50     0    19    8    1    1
0
axle       2     1
MSG81     30     0    18    8    1    1
MSG82   1000     0    17    8    1    1
0
eng_con    1     1
MSG91   5000     0    16    8    1    1
0
ind        1     1
MSG101    20     0    15    8    1    1
0
```

Input file IV

```
100
1
1
engine      3     3
MSG11      10     1    31    8    1    1
MSG12      50     0    30    8    0    1
MSG13     250     0    29    8    1    1
2
0
0
torque      1     2
MSG21      10     0    28    8    1    1
0
0
trans1      1     1
MSG31      10     0    27    8    1    0
0
trans2      3     1
MSG41      10     0    26    8    1    1
MSG42     100     0    25    8    1    1
MSG43    1000     0    24    8    1    1
0
brake       2     1
MSG51     100     0    23    8    1    1
MSG52    1000     0    22    8    1    1
0
retarder    2     1
MSG61     100     0    21    8    1    1
MSG62    1000     0    20    8    1    1
0
brk_ctrl    1     1
MSG71      50     0    19    8    1    1
0
axle        2     1
MSG81      30     0    18    8    1    1
MSG82    1000     0    17    8    1    1
0
eng_con     1     1
MSG91    5000     0    16    8    1    1
0
trans_con   1     1
MSG101     10     0    15    8    1    1
0
```

retr_con 1 1
MSG111 10 0 14 8 1 1
0
eng_fluid 1 1
MSG121 1000 0 13 8 1 1
0
eng_temp 1 1
MSG131 1000 0 12 8 1 1
0
eng_hrs 1 1
MSG141 10 0 11 8 1 1
0
pto_def 1 1
MSG151 100 0 10 8 1 1
0
idle_pto 1 1
MSG161 1000 0 9 8 1 1
0
speed 1 1
MSG171 100 0 8 8 1 1
0

Input file V

```
100
1
1
engine     3     3
MSG11     10     1    31    8    1    1
MSG12     50     0    30    8    0    1
MSG13    250     0    29    8    1    1
2
0
0
torque     1     2
MSG21     10     0    28    8    1    1
0
0
trans1     1     1
MSG31     10     0    27    8    1    0
0
trans2     3     1
MSG41     10     0    26    8    1    1
MSG42    100     0    25    8    1    1
MSG43   1000     0    24    8    1    1
0
brake      2     1
MSG51    100     0    23    8    1    1
MSG52   1000     0    22    8    1    1
0
retarder   2     1
MSG61    100     0    21    8    1    1
MSG62   1000     0    20    8    1    1
0
brk_ctrl   1     1
MSG71     50     0    19    8    1    1
0
axle       2     1
MSG81     30     0    18    8    1    1
MSG82   1000     0    17    8    1    1
0
eng_con    1     1
MSG91   5000     0    16    8    1    1
0
trans_con  1     1
MSG101    10     0    15    8    1    1
0
```

```
retr_con    1    1
MSG111   10    0    14    8    1    1
0
eng_fluid    1    1
MSG121 1000    0    13    8    1    1
0
eng_temp    1    1
MSG131 1000    0    12    8    1    1
0
eng_hrs    1    1
MSG141   10    0    11    8    1    1
0
pto_def    1    1
MSG151  100    0    10    8    1    1
0
idle_pto    1    1
MSG161 1000    0    9    8    1    1
0
speed    1    1
MSG171  100    0    8    8    1    1
0
calib    1    1
MSG181   10    0    7    8    1    1
0
miles    1    1
MSG191   10    0    6    8    1    1
0
ind    1    1
MSG201   20    0    5    8    1    1
0
```

Input file VI

```
100
1
1
engine      3     3
MSG11      10     1    31    8    1    1
MSG12      50     0    30    8    0    1
MSG13     250     0    29    8    1    1
2
0
0
torque      1     2
MSG21      10     0    28    8    1    1
0
0
trans1      1     1
MSG31      10     0    27    8    1    0
0
trans2      3     1
MSG41      10     0    26    8    1    1
MSG42     100     0    25    8    1    1
MSG43    1000     0    24    8    1    1
0
brake       2     1
MSG51     100     0    23    8    1    1
MSG52    1000     0    22    8    1    1
0
retarder    2     1
MSG61     100     0    21    8    1    1
MSG62    1000     0    20    8    1    1
0
brk_ctrl    1     1
MSG71      50     0    19    8    1    1
0
axle        2     1
MSG81      30     0    18    8    1    1
MSG82    1000     0    17    8    1    1
0
eng_con     1     1
MSG91    5000     0    16    8    1    1
0
trans_con   1     1
MSG101     10     0    15    8    1    1
0
```

retr_con	1	1				
MSG111	10	0	14	8	1	1
0						
eng_fluid	1	1				
MSG121	1000	0	13	8	1	1
0						
eng_temp	1	1				
MSG131	1000	0	12	8	1	1
0						
eng_hrs	1	1				
MSG141	10	0	11	8	1	1
0						
pto_def	1	1				
MSG151	100	0	10	8	1	1
0						
idle_pto	1	1				
MSG161	1000	0	9	8	1	1
0						
speed	1	1				
MSG171	100	0	8	8	1	1
0						
calib	1	1				
MSG181	10	0	7	8	1	1
0						
miles	1	1				
MSG191	10	0	6	8	1	1
0						
fuel	2	1				
MSG201	200	0	5	8	1	1
MSG202	10	0	4	8	1	1
0						
ind	1	1				
MSG211	100	0	3	8	1	1
0						
tire	1	1				
MSG221	10000	0	37	8	1	1
0						
amby	1	1				
MSG231	1000	0	38	8	1	1
0						
exhst	1	1				
MSG241	1000	0	39	8	1	1
0						
power	1	1				
MSG251	1000	0	40	8	1	1

0						
fluids	1	1				
MSG261	1000	0	41	8	1	1
0						
dash	1	1				
MSG271	10000	0	43	1	1	1
0						
water	1	1				
MSG281	10000	0	45	7	1	1
0						
diag	2	1				
MSG291	600	0	46	3	1	1
MSG292	700	0	47	3	1	1
0						
ind	1	1				
MSG301	20	0	48	8	1	1
0						

Input file VII

```
100
1
1
engine      3     2
MSG11      10     1    31    8    1    1
MSG12      50     0    30    8    0    1
MSG13     250     0    29    8    1    1
2
0
torque      1     1
MSG21      10     0    28    8    1    1
0
trans1      1     1
MSG31      10     0    27    8    1    0
0
trans2      3     1
MSG41      10     0    26    8    1    1
MSG42     100     0    25    8    1    1
MSG43    1000     0    24    8    1    1
0
brake       2     1
MSG51     100     0    23    8    1    1
MSG52    1000     0    22    8    1    1
0
retarder    2     1
MSG61     100     0    21    8    1    1
MSG62    1000     0    20    8    1    1
0
brk_ctrl    1     1
MSG71      50     0    19    8    1    1
0
axle        2     1
MSG81      30     0    18    8    1    1
MSG82    1000     0    17    8    1    1
0
eng_con     1     1
MSG91    5000     0    16    8    1    1
0
trans_con   1     1
MSG101     10     0    15    8    1    1
0
retr_con    1     1
MSG111     10     0    14    8    1    1
```

0						
eng_fluid	1	1				
MSG121	1000	0	13	8	1	1
0						
eng_temp	1	1				
MSG131	1000	0	12	8	1	1
0						
eng_hrs	1	1				
MSG141	10	0	11	8	1	1
0						
pto_def	1	1				
MSG151	100	0	10	8	1	1
0						
idle_pto	1	1				
MSG161	1000	0	9	8	1	1
0						
speed	1	1				
MSG171	100	0	8	8	1	1
0						
calib	1	1				
MSG181	10	0	7	8	1	1
0						
miles	1	1				
MSG191	10	0	6	8	1	1
0						
fuel	2	1				
MSG201	200	0	5	8	1	1
MSG202	10	0	4	8	1	1
0						
ind	1	1				
MSG211	100	0	3	8	1	1
0						
tire	1	1				
MSG221	10000	0	37	8	1	1
0						
amby	1	1				
MSG231	1000	0	38	8	1	1
0						
exhst	1	1				
MSG241	1000	0	39	8	1	1
0						
power	1	1				
MSG251	1000	0	40	8	1	1
0						
fluids	1	1				

MSG261	1000	0	41	8	1	1
0						
dash	1	1				
MSG271	10000	0	43	1	1	1
0						
water	1	1				
MSG281	10000	0	45	7	1	1
0						
diag	2	1				
MSG291	600	0	46	3	1	1
MSG292	700	0	47	3	1	1
0						
ind	1	1				
MSG301	800	0	36	8	1	1
0						
ind2	1	1				
MSG311	800	0	35	8	1	1
0						
ind3	1	1				
MSG321	700	0	34	8	1	1
0						
ind4	1	1				
MSG331	600	0	33	8	1	1
0						
ind5	1	1				
MSG341	500	0	32	8	1	1
0						
ind6	1	1				
MSG351	400	0	63	8	1	1
0						
ind7	1	1				
MSG361	300	0	1	8	1	1
0						
ind8	1	1				
MSG371	30	0	1	8	1	1
0						
ind9	1	1				
MSG381	40	0	1	8	1	1
0						
ind10	1	1				
MSG391	50	0	1	8	1	1
0						
ind11	1	1				
MSG401	60	0	1	8	1	1
0						

Input file VIII

```
100
1
1
engine      3     3
MSG11       10    1    31    8    1    1
MSG12       50    0    30    8    0    1
MSG13       250   0    29    8    1    1
2
0
0
torque      1     2
MSG21       10    0    28    8    1    1
0
0
trans1      1     1
MSG31       10    0    27    8    1    0
0
trans2      3     1
MSG41       10    0    26    8    1    1
MSG42       100   0    25    8    1    1
MSG43       1000  0    24    8    1    1
0
brake       2     1
MSG51       100   0    23    8    1    1
MSG52       1000  0    22    8    1    1
0
retarder    2     1
MSG61       100   0    21    8    1    1
MSG62       1000  0    20    8    1    1
0
brk_ctrl    1     1
MSG71       50    0    19    8    1    1
0
axle        2     1
MSG81       30    0    18    8    1    1
MSG82       1000  0    17    8    1    1
0
eng_con     1     1
MSG91       5000  0    16    8    1    1
0
trans_con   1     1
MSG101      10    0    15    8    1    1
```

```
0
retr_con     1     1
MSG111      10     0     14     8     1     1
0
eng_fluid    1     1
MSG121    1000     0     13     8     1     1
0
eng_temp     1     1
MSG131    1000     0     12     8     1     1
0
eng_hrs      1     1
MSG141      10     0     11     8     1     1
0
pto_def      1     1
MSG151     100     0     10     8     1     1
0
idle_pto     1     1
MSG161    1000     0      9     8     1     1
0
speed        1     1
MSG171     100     0      8     8     1     1
0
calib        1     1
MSG181      10     0      7     8     1     1
0
miles        1     1
MSG191      10     0      6     8     1     1
0
fuel         2     1
MSG201     200     0      5     8     1     1
MSG202      10     0      4     8     1     1
0
ind          1     1
MSG211     100     0      3     8     1     1
0
tire         1     1
MSG221   10000     0     37     8     1     1
0
amby         1     1
MSG231    1000     0     38     8     1     1
0
exhst        1     1
MSG241    1000     0     39     8     1     1
0
power        1     1
```

MSG251	1000	0	40	8	1	1
0						
fluids	1	1				
MSG261	1000	0	41	8	1	1
0						
dash	1	1				
MSG271	10000	0	43	1	1	1
0						
water	1	1				
MSG281	10000	0	45	7	1	1
0						
diag	2	1				
MSG291	600	0	46	3	1	1
MSG292	700	0	47	3	1	1
0						
ind	1	1				
MSG301	800	0	36	8	1	1
0						
ind2	1	1				
MSG311	800	0	35	8	1	1
0						
ind3	1	1				
MSG321	700	0	34	8	1	1
0						
ind4	1	1				
MSG331	600	0	33	8	1	1
0						
ind5	1	1				
MSG341	500	0	32	8	1	1
0						
ind6	1	1				
MSG351	400	0	63	8	1	1
0						
ind7	1	1				
MSG361	300	0	1	8	1	1
0						
ind8	1	1				
MSG371	30	0	1	8	1	1
0						
ind9	1	1				
MSG381	40	0	1	8	1	1
0						
ind10	1	1				
MSG391	50	0	1	8	1	1
0						

ind11	1	1				
MSG401	60	0	1	8	1	1
0						
ind12	1	1				
MSG411	70	0	1	8	1	1
0						
ind13	1	1				
MSG421	80	0	1	8	1	1
0						
ind14	1	1				
MSG431	90	0	1	8	1	1
0						
ind15	1	1				
MSG441	100	0	1	8	1	1
0						
ind16	1	1				
MSG451	110	0	1	8	1	1
0						
ind17	1	1				
MSG461	120	0	1	8	1	1
0						
ind18	1	1				
MSG471	130	0	1	8	1	1
0						
ind19	1	1				
MSG481	140	0	1	8	1	1
0						
ind20	1	1				
MSG491	150	0	1	8	1	1
0						
ind21	1	1				
MSG501	160	0	1	8	1	1
0						

APPENDIX D

STATISTICS

Input Number	I	II	III	IV	V	VI	VII	VIII
Number of Nodes	2	3	10	17	20	30	40	50
Number of Messages	3	5	17	24	27	39	50	60
Number of Messages Transmitted	13	23	52	82	107	128	142	155
Number of Remote Transmissions	0	10	10	10	10	10	10	10
Number of Collisions	3	16	67	102	125	147	166	177
Number of Messages Losing Arbitration	3	18	204	521	828	1528	2216	3049
Number of Errors	4	19	17	19	18	17	20	16
Number of Overload Errors	0	0	0	1	0	2	0	1
Number of Acknowledgment Errors	0	4	2	5	3	2	4	3
Number of Form Errors	0	4	3	3	1	2	0	1
Number of CRC Errors	4	11	12	10	14	11	16	11
Number of Messages Resent	4	19	17	19	18	17	20	16
Idle Time (in ms)	91.06	72.51	58.86	42.65	29.84	21.11	11.35	7.19
Busy Time (in ms)	6.83	17.28	32.11	47.65	60.61	70.81	78.06	84.79
Error Time (in ms)	2.11	10.20	9.03	9.70	9.55	8.08	10.59	8.01
Average Response Time (in ms)	0.37	1.03	3.28	4.20	5.05	7.45	9.88	12.96
Average Slack Time (in ms)	33.72	26.48	195.53	163.15	125.88	380.82	372.88	346.62
Average Latency (in ms)	0.00	0.00	0.00	0.00	0.00	0.06	0.25	0.60
Network Load (in %)	8.94	27.49	41.14	57.35	70.16	78.89	88.65	92.81
Network Throughput (in msgs / second)	130	230	520	820	1070	1280	1420	1550

VITA

Natarajan S. Pennathur

Candidate for the Degree of

Master of Science

Thesis: A BITWISE SIMULATION OF THE CONTROLLER AREA NETWORK

Major Field: Computer Science

Biographical:

Personal Data: Born in Chidambaram, India, on March 16, 1967, to Sundaresan, P. S., and Shyamala, S.

Education: Received high school certificate from Lindsay Memorial, Kolar Gold Fields, India, in May 1983; completed undergraduate studies in Computer Science and Engineering, with a Bachelor of Engineering Degree from the University of Mysore, India, January 1990; completed requirements for the Master of Science Degree at Oklahoma State University, Stillwater, December 1993.

Professional Experience: Lecturer, Department of Computer Science, Bangalore University, India, January 1990 to August 1991.