HARD REAL-TIME COMMUNICATIONS IN CONTROLLER AREA NETWORK

By

WEIQING LI

Bachelor of Engineering
Shandong Engineering Institute
Jinan, China
1982

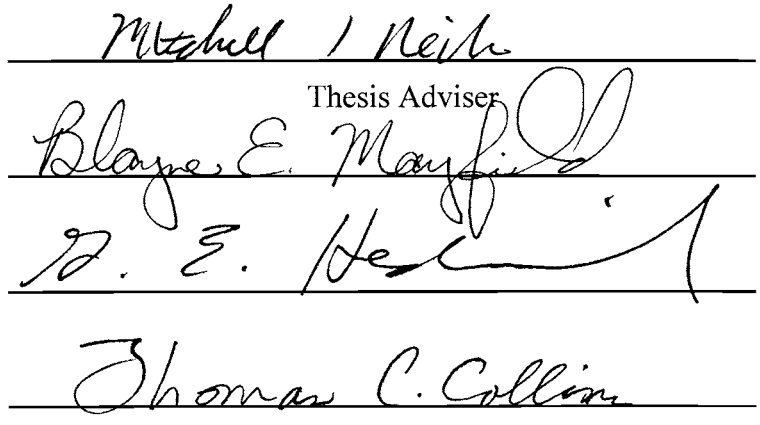
Master of Engineering
Shandong Polytechnic University
Jinan, China
1988

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, 1995

HARD REAL-TIME COMMUNICATIONS IN

CONTROLLER AREA NETWORK

Thesis Approved:



Thesis Adviser

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my appreciation to my advisor, Dr. Mitch Neilsen, for his intelligent supervision, constructive guidance, inspiration and friendship throughout my graduate study and writing of this thesis. My sincere appreciation extends to my other committee members, Dr. G. E. Hedrick and Dr. Blayne E. Mayfield, whose guidance, assistance, encouragement, and friendship were also invaluable.

I would also like to give special appreciation to my wife, Lihong Sun, for her precious suggestions to my research, her strong encouragement at times of difficulty, and love and understanding throughout this whole process. My deepest appreciation is extended to my daughter, Minghui Li, for her patience and love.

I am deeply indebted to my beloved parents who have provided moral support and constant encouragement in all my endeavors.

Finally, I would like to thank the Department of Computer Science for providing me with their generous financial support during my study.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

CHAPTER I

INTRODUCTION

There are two types of real-time systems, namely, *soft* real-time systems and *hard* real-time systems. In soft real-time systems, tasks are performed by the system as fast as possible, but they are not constrained to finish by specific times. On the other hand, in hard real-time systems, tasks have to be performed not only correctly, but also in a timely fashion. Otherwise, there might be severe consequences.

A hard real-time system is often composed of a number of periodic and sporadic tasks which communicate their results by passing messages; in a distributed system these messages are sent between processors across a communication device. In order to guarantee that the timing requirements of all tasks are met, the communication delay, between a sending task queuing a message and a receiving task being able to access that message, must be bounded. This total delay is often termed the *end-to-end communication* delay — the time between a message being queued by the sending task and the message fully arriving at the receiving task. The end-to-end communication delay is made up of four major components (Figure 1):

1. the *generation delay*: the time taken for the application task to generate and queue the message,
2. the *queuing delay*: the time taken by the message to gain access to the communiction device after being queued,
3. the *transmission delay*: the time taken by the message to be transmitted on the communication device, and
4. the *delivery delay*: the time taken to process the message at the destination processor before finally delivering it to the destination task.

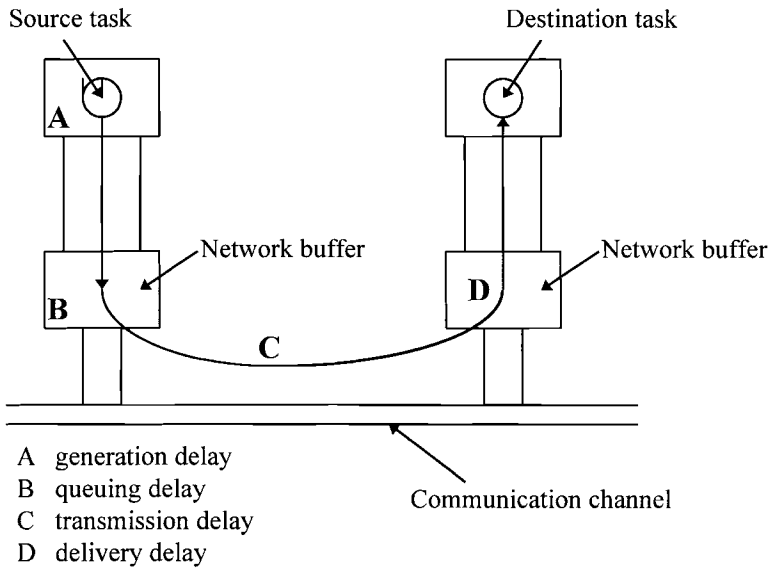


Figure 1: The Components of Communications

The generation delay is the worst-case time taken between the arrival of the sender task and the queuing of the message. This represents some element of application processing to generate the contents of the message, and the time taken to queue the message. The queuing delay is the time the message spends waiting to be removed from the queue by the communications device. With a point-to-point communication link, the message must contend with other messages sent from the same processor; with a shared communications link, the message must also contend with messages sent from other processors. The transmission delay is the time taken for the message to be sent once it has been removed from the queue. The delivery delay is the amount of time taken to process the incoming data and deliver it to destination tasks. This work includes such functions as decoding packed headers, re-assembling multi-packet messages, copying message data between buffers, and notifying the dispatcher of the arrival of a message. This latter function is important, since the destination task may be blocked awaiting the

arrival of the message. In practice the delivery delay can form a significant part of the end-to-end communications delay.

The most important aspect of hard real-time applications is that a message generated at the source station must be received at the destination station within a given amount of time after its generation at the sending station. If a message's delay exceeds this time constraint, the message is considered lost, regardless of whether it is ever received at the destination station.

The message flow for hard real-time communication applications is shown in Figure 2. The messages generated by a real-time application (A) at a sending station are first buffered or stored (B) within the sending station. Once a message has been buffered, the network access mechanism (C) eventually decides whether the message should be transmitted into the communication subsystem (E) or should be explicitly discarded at the sending station (D). If a message is eventually received successfully at the destination station but its delay exceeds its time constraint, the message is lost (G); otherwise it is passed on to the real-time application (H) at the destination station.

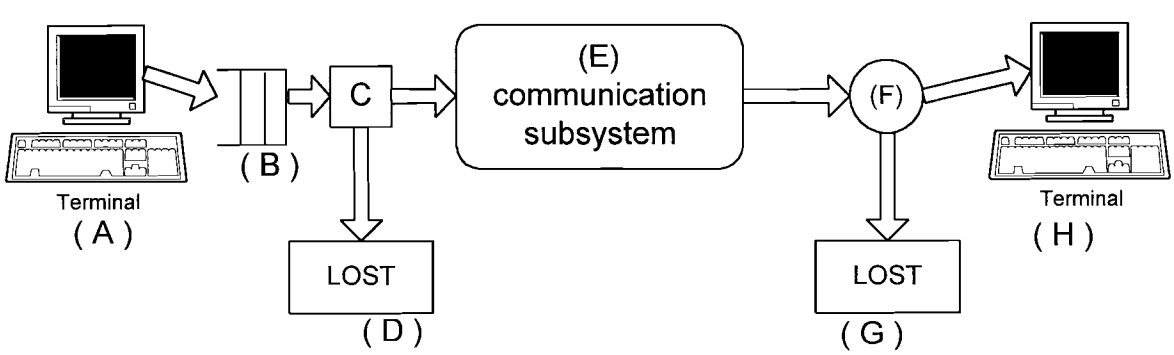


Figure 2. Real-time communication in a network environment.

There are several important communication applications having such real-time characteristics. One application is packetized voice, in which the human voice is digitized, packetized at the sending station, transmitted over the network subsystem, and reconstructed and played out synchronously at the destination station [12]. Since excessive delays can have seriously disruptive effects on human conversation, voice packets are usually constrained to arrive at the destination station within a given amount of time after their generation at the sending station. Those packets that do not arrive within the time bound are considered lost; a small number of lost packets has been shown to have little, if any, effect on human speech intelligibility.

A second application requiring real-time communication is distributed vehicle-monitoring applications in which distributed stations attempt to track a moving object using their local observations and the communicated observations of the other stations [2]. Since the position of the object is continually changing, only a small amount of time is available to fix its current location or trajectory. The distributed observations necessary to determine the current location thus must be communicated within this amount of time. A small amount of message loss due to excessive delays may be tolerable, but may result in uncertainty in the object's calculated position. A third class or real-time applications is real-time control applications, in which stations must initiate some action at remote devices within a specified amount of time.

Most research into hard real-time communications has concentrated on protocols bounding the access delay to shared communications media. For example, the MARS project uses a simple TDMA protocol to resolve communications media contention between processors [30]. A simple priority queue can be used to resolve contention between local messages. Strosnider *et al* [5], and late Pleinevaux [35], apply rate monotonic analysis to periodic and aperiodic messages sent across an 802.5 token ring. Agrawal *et al* [2] also apply the rate monotonic scheduling approach to the FDDI access protocol.

The 802.5 token ring protocol is an example of a global priority scheme — packets sent on the bus are assigned a priority, and the highest priority packed in any node is transmitted. This priority arbitration is carried out by a *reservation protocol*, whereby each node 'bids' for the right to transmit the next packet; the node with the highest priority packet wins the bidding.

Existing fixed priority schedulability analysis (such as rate monotonic analysis [22, 38]) has suffered from restrictions on deadlines: the rate monotonic approach requires task deadlines to be equal to their periods, and thus applying this analysis to message schedulability gives the restriction that a message must arrive at the destination processor before the message from the next period can be queued. Further, the rate monotonic approach does not lend itself easily to supporting sporadic activities, requiring the need for periodic servers to poll for these activities. For the scheduling of messages this can be very restrictive, since complex behaviour of the priority exchange server algorithm can be very difficult to implement in a distributed system. Deadline monotonic analysis would seem to offer some solutions: for example, deadlines are permitted to be less than or equal to periods, and the approach can easily accommodate sporadic activities [5, 14]. However, in a real system it is often acceptable for the deadline on the arrival time of a message to be longer than the period of a message: certain control and multi-media applications can tolerate long lags (*i.e.*, end-to-end communication delays) so long as the rate at which data arrives is maintained. Hence an approach permitting arbitrary deadlines is required [16]. Such analysis has been provided whereby the worst-case response time of tasks with arbitrary deadlines can be determined [20].

This thesis reviews relevant work, and presents a theoretical analysis of real-time in-vehicle networking protocol. Chapter II presents an introduction to the Controller Area Networks. Chapter III presents a model for analysis of the network. Chapter IV presents the existing processor scheduling analysis. Chapter V presents the analysis of CAN message response times. Chapter VI presents the analysis for different controllers.

CHAPTER II

REAL-TIME COMMUNICATION

2.1 Overview

The *Controller Area Network* (CAN) protocol, developed by Bosch GmbH, offers a comprehensive solution to managing communication between multiple CPUs. The CAN protocol specifies versatile message identifiers that can be mapped to specific control information categories. Communications may occur at a maximum recommended rate of 1 Mbit/sec (on a 40 meter bus length). The protocol has found wide acceptance in automotive in-vehicle applications as well as many non-automotive applications due to its low cost, high performance, and the availability of various CAN protocol implementations.

In-vehicle networking protocols satisfy unique requirements not present in other networking protocols such as those found in telecommunications and data processing. These requirements include a high level of error detection, low latency times and configuration flexibility.

The CAN protocol provides four primary benefits. First, a standard communication protocol simplifies and economizes the task of interfacing subsystems from various vendors into a common network. Second, the communications burden is shifted from the host-CPU to an intelligent peripheral; the host-CPU has more time to run its system tasks. Third, as a multiplexed network, CAN greatly reduces wire harness size and eliminates point-to-point wiring. Lastly, as a standard protocol, CAN has broad market appeal which motivates semiconductor makers to develop competitively priced CAN devices.

An example of an application well-served by the CAN protocol is automotive networking because many modules are inter-dependent. Sub-systems such as the engine, transmission, anti-lock braking, and accident avoidance systems require the exchange of

particular performance and position information within a defined communications latency. The engine transmits engine speed and acceleration parameters to the transmission to allow smoother shifting. Perhaps the transmission requests the engine to reduce fuel injection before a gear change.

CAN is a CSMA/CD-A, or Carrier Sense Multiple Access with Collision Detection and Bit-wise Arbitration. Through a multi-master architecture, prioritized messages of length 8 bytes or less are sent on a serial bus. Error detection mechanisms, such as a 15-bit CRC, provide a high level of data integrity.

The CAN 2.0 protocol was chosen by the SAE Truck & Bus Control and Communications Network Subcommittee of the Truck & Bus Electrical Committee to support its "Recommended Practice for Serial Control and Communications Vehicle Network CLASS C" called the SAE J1939 specification. The SAE CLASS C passenger car subcommittee is currently evaluating CAN, which is a candidate for its high speed networks. Products using CAN Version 2.0 are already in production. The previous CAN specification, Version 1.2, has been successfully implemented in passenger car, train and factory automation applications since 1989. CAN Version 2.0, which features an "extended frame" with a 29-bit message identifier, broadens the application base for this protocol by allowing J1850 message schemes to be mapped into the CAN message format.

The Intel 82526 was the first implementation of the CAN protocol, in production since 1989. The Intel 82527 is a follow-on to the 82526 which implements CAN 2.0, provides greater message handling capability and implements a more flexible interface to CPUs.

2.2. Basic CAN Protocol

The real-time bus we examine in this paper is called *Controller Area Network* (CAN). CAN is a broadcast bus where a number of processors are connected to the bus via an interface (Figure 3).

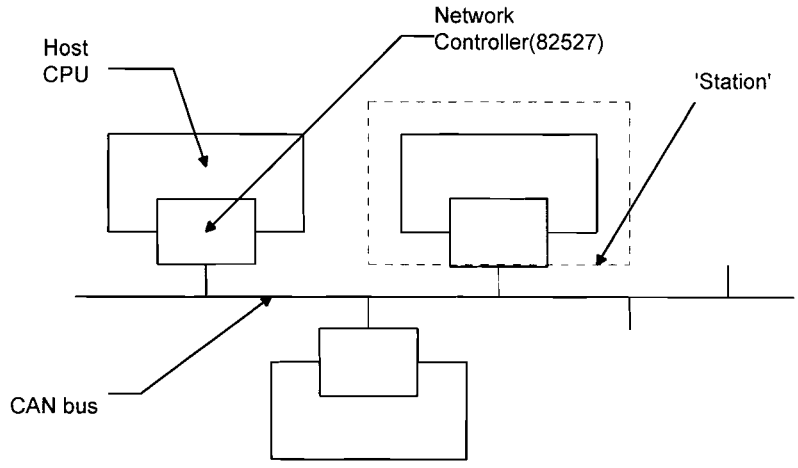


Figure 3: CAN architecture

A data source is transmitted as a *message*, consisting of between 1 and 8 bytes ('octets'). A data source may be transmitted periodically, sporadically, or on-demand. So for example, a data source such as 'road speed' could be encoded as a 1 byte message and broadcast every 100 milliseconds. The data source is assigned a unique *identifier*, represented as an 11 bit number (giving 2032 identifiers - CAN prohibits identifiers with the seven most significant bits equal to "1"). The identifier servers two purposes: filtering messages upon reception, and assigning a priority to the message.

A station on a CAN bus is able to receive a message based on the message identifier: if a particular host CPU needs to obtain the road speed (for example) then it indicates the identifier to the interface processor. Only messages with desired identifiers are received and presented to the host processor. Thus in CAN message has no destination.

The use of the identifier as priority is the most important part of CAN regarding real-time performance. In any bus system there must be a way of resolving contention: with a TDMA bus, each station is assigned a pre-determined time slot in which to transmit.

With Ethernet, each station waits for silence and then starts transmitting. If more than one station tries to transmit together then they all detect this, wait for a randomly determined time period, and try again the next time the bus is idle. Ethernet is an example of a carrier-sense broadcast bus, since each station waits until the bus is idle (*i.e.* no carrier is sensed), and monitors its own traffic for collisions. CAN is also a carrier-sense broadcast bus, but takes much more systematic approach to contention. The identifier field of a CAN message is used to control access to the bus after collisions by taking advantage of certain electrical characteristics.

With CAN, if multiple stations are transmitting concurrently and one station transmits a '0' bit, then all stations monitoring the bus will see a '0'. Conversely, only if all stations transmit a '1' will all processors monitoring the bus see a '1'. In CAN terminology, a '0' bit is termed *dominant*, and a '1' bit is termed *recessive*. In effect, the CAN bus acts like a large AND-gate, with each station able to see the output of the gate. This behavior is used to resolve collisions: each station waits until bus idle (as with Ethernet). When silence is detected each station begins to transmit the highest priority message held in its queue whilst monitoring the bus. The message is coded so that the most significant bit of the identifier field is transmitted first. If a station transmits a recessive bit of the message identifier, but monitors the bus and sees a dominant bus then a collision is detected. The station knows that the message it is transmitting is not the highest priority message in the system, stops transmitting and waits for the bus to become idle. If the station transmits a recessive bit and sees a recessive bit on the bus then it may be transmitting the highest priority message, and proceeds to transmit the next bit of the identifier field. Because CAN requires identifiers to be unique within the system, a station transmitting the last bit of the identifier without detecting a collision must be transmitting the highest priority queued message, and hence can start transmitting the body of the message (if identifiers were not unique then two stations attempting to transmit different messages with the

same identifier would cause a collision after the arbitration process has finished, and an error would occur).

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.

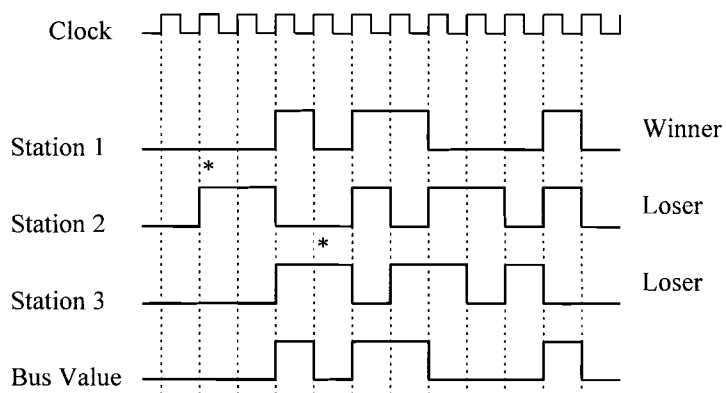


Figure 4. Collision Resolution by Non-Destructive Bitwise Arbitration

There are some general observations to make on this arbitration protocol. Firstly, a message with a smaller identifier value is a higher priority message. Secondly, the highest priority message undergoes the arbitration process without disturbance (since all other stations will backed-off and ceased transmission until the bus is next idle). The whole message is thus transmitted without interruption.

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. 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. 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.

The data frame structure is shown in Figure 5.

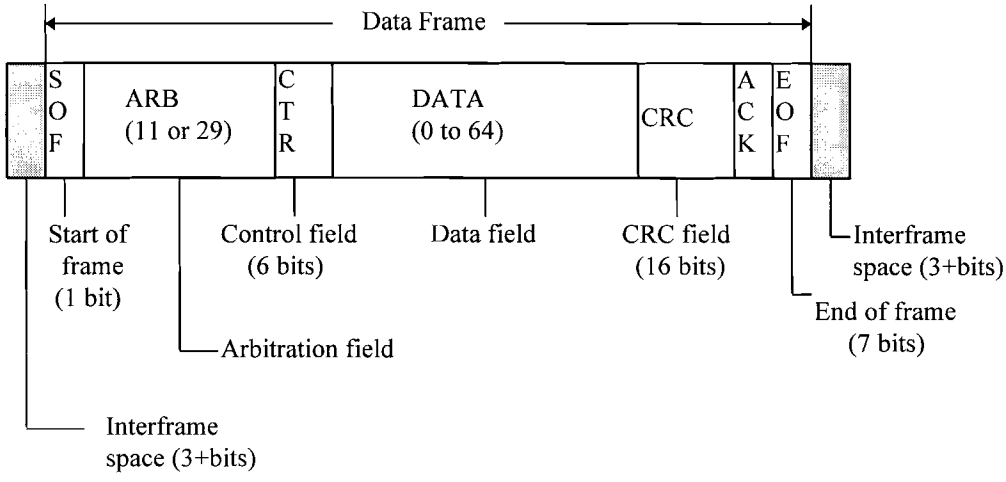


Figure 5. Data Frame Format

Each frame starts with a start of frame bit, signaling the start of a data frame. The arbitration field follows the start bit and contains the message identifier and one additional control bit for other purposes. The control field and data field follow the arbitration field. The control field consists of 6 bits, 2 reserved bits, and 4 data field length bits. The length of the data field is coded in bytes, and the control field identifies the number of bytes of data presented in the data field. The length of the data field

varies from 0 to 8 bytes. The frame with a data field 0 is a special frame called a remote frame, which requests certain message to be sent to the bus. The contents of the arbitration field, control field, and data field of the frame corresponds to the contents of the communication object to be transmitted or received in the DPRAM.

The CRC field contains a 15 bits cyclic redundancy check sum and a 1 bit delimiter. The check sum checks the start bit, arbitration field, control field, data field, and CRC field itself. After a message is completed the CRC field is checked. If any error is detected in the frame, the whole frame will be retransmitted.

The acknowledge field consists of two bits, the ACK_SLOT bit and the ACK_DELIMITER bit. The transmitting node sends the ACK_SLOT bit as 1. Any receiving node which has received a frame correctly will send a 0 to the bus at the same time. This 0 will overwrite the 1 sent by the transmitting node. Because the transmitting node listens to the bus, it will find the change and know that at least one station has received the message completely and correctly. The end of frame field consists of 7 bits, all of them 1. It marks the end of the frame.

Besides data frame and remote frame there are 2 more frames: the error frame, which indicates error conditions, and overload frame, which signifies receiving station not ready condition or wrong interframe space bit condition. Data frames and remote frames are preceded by at least 3 interframe spaces which allow the network interface to get ready to transmit or receive the next frame.

The CAN message format contains 47 bits of protocol control information (the identifier, CRC data, acknowledgment and synchronisation bits, *etc.*). The data transmission uses a bit stuffing protocol which inserts a 'stuff bit' after five consecutive bits of the same value. Because the number of inserted stuff bits depends on the bit pattern of a message, a given message type can very in size, *e.g.* a CAN message with 8 bytes of data (and 47 control bits) is transmitted with between 0 and 19 stuff bits.

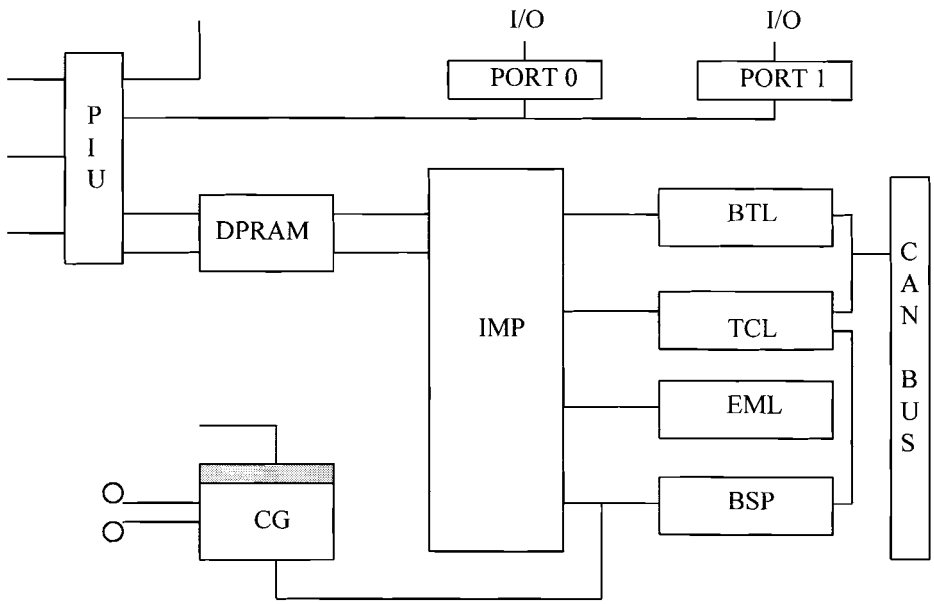
As well as data messages, CAN also permits 'remote transmission request' (RTR) messages. These messages are contentless and have a special meaning; they instruct the station holding a data message of the same identifier to transmit that message. RTR messages are intended for quickly obtaining infrequently used remote data.

2.3. Structure of Communication Controller

The main components of the communication controller include a dual port RAM (DPRAM), an interface management processor (IMP), and a processor interface unit (PIU). Other components include a bus timing logic (BTL), a transceive logic (TCL), and error management logic (EML), a bit stream processor(BSP), and a clock generator (CG). A block diagram representation is as shown in Figure 6.

The DPRAM forms a communication buffer between the station microprocessor and the IMP. Messages are stored as communication objects in the DPRAM. 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 DPRAM. It performs these tasks by means of acceptance and transmission filtering. This is done by scanning the communication objects in the DPRAM 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 DPRAM 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.



PIU: Processor Interface Unit

CG: Clock Generator

IMP: Interface Management Processor

BTL: Bus Time Logic

TCL: Transceive Logic

EML: Error Management Logic

BSP: Bit Stream Processor

DPRAM:Dual Port RAM

Figure 6. CAN Network Controller Structure

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.

2.4. Interface Between Host Processor and CAN Processor

From the observations of the basic CAN protocol mentioned earlier, we can calculate the worst-case time from queuing the highest priority message to the reception of that message (*i.e.*, the worst-case response time of the message): the longest time a station must wait for the bus to become idle is the longest time to transmit a CAN message (we call this delay the *blocking time* of a message). The largest CAN message (8 bytes) takes 130 microseconds to be transmitted (at 1Mbit/sec transmission speed, with a 'bit stuffing' width of 5 bits), and hence the blocking time of a CAN message is 130 microseconds. The worst-case response time of the highest priority CAN message is therefore 130 microseconds plus the time taken to transmit the message. For a lower priority message, the worst-case response time cannot be found so easily, leading to the generally perceived problem that only the highest priority message can be guaranteed on CAN. In this paper, we are going to try to bound the response time of all CAN messages, including the lowest priority message.

CPU software programmers interface CAN via a Dual Port RAM (DPRAM), which is accessed like any other memory. The sum of DPRAMs in all distributed control units forms a common virtual memory containing all COMMUNICATION OBJECTs. An access window is defined for each station by listing up COMMUNICATION OBJECTs to be transmitted and received. Figure 7 depicts a typical interface.

In Figure 7 the host processor is queuing a message into the slot for identifier '1'; the slot for identifier '4' is already occupied with another message. The slots are typically implemented as DPRAM shared between the processors. The interface processor will attempt to transmit message '1' when the bus next becomes idle. There is no queue of messages for a given identifier: in Figure 7, if message '1' is being transmitted when another message with same identifier is queued then the message in the slot is overwritten and destroyed. This is important, since it implies a deadline for a message queued periodically: a given message must be transmitted before the message for the next period can be queued. So, returning to the example of a message containing 'road speed', we can see that the message must be transmitted within 100 milliseconds to avoid being overwritten by the contents of the message corresponding to the next measurement. In effect, we have a deadline on the transmission of any message: the message must be transmitted before the subsequent message can be queued (of course, we may have a deadline on the message that is much shorter than the period).

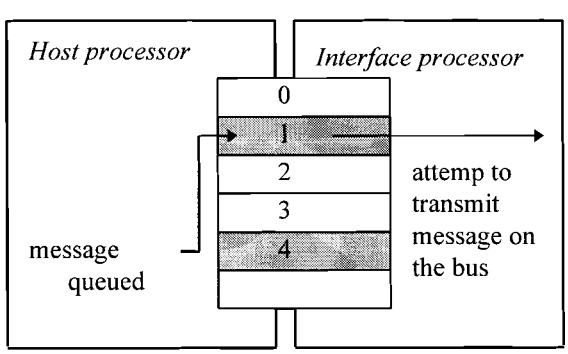


Figure 7. Interface between host processor and CAN processor

CHAPTER III

MODEL FOR COMPUTATION

3.1 Layered Structure of CAN

In CAN, nodes communicate using a single shared channel. Figure 8 shows the layered architecture of the ISO/OSI model. Each layer has a different set of protocols responsible for carrying out the functions required of the layer. For example, the scope of the LLC sublayer is to provide services for data transfer and for remote data request, to decide which messages received by the LLC sublayer are actually to be accepted, and to provide means for recovery management and overload notifications. The scope of the MAC sublayer mainly is the transfer protocol which is responsible for selecting and sending messages over the shared channel of the CAN. In terms of OSI Reference Model, MAC protocols form part of the data link layer.[10]

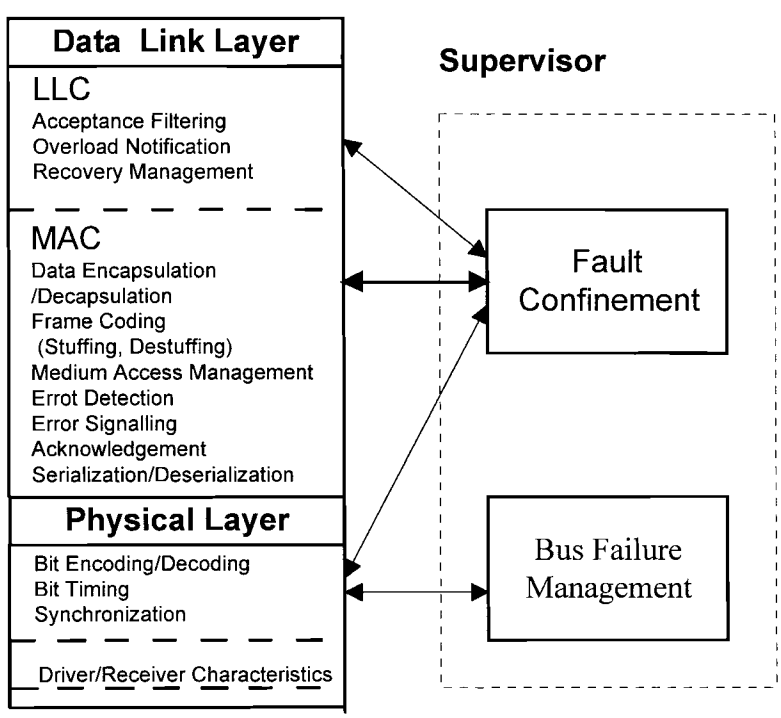


Figure 8. The OSI Reference Model

3.2. Communications Model

We define a message to be either a *data message*, or a *remote transmission request message.* A message has a size (between zero and eight bytes), and an identifier. The set of all messages in the system is denoted messages.

In a typical system, a message is queued by an application task. We assume that each task is invoked repeatedly (a task is said to have arrived when invoked by some action). Each task has a minimum inter-arrival time termed the *period.* Note that the period is a minimum time between subsequent arrivals, rather than a strict fixed interval. If the message queued by a given task is potentially sent each time the task is invoked, then the message inherits a period equal to the period of the task. We denote as P_{msg} the period of a given message *msg*.

A given task *i* has a worst-case response time, denoted RT_i, which is defined as the longest time between the arrival of a task and the time it completes some bounded amount of computation. Existing analysis for single processors is able to determine this worst-case response time.

In general, the queuing of a message can occur with *jitter* (variability in queuing times). Correct analysis requires that jitter be taken into account. Queuing jitter can be defined as the difference between the earliest and latest possible times a given message can be queued.

As with the period, the jitter of a given message *msg* may be inherited from the sender task. For an application task *i* (with worst-case response time RT_i) sending message *msg*, this queuing window is no more than RT_i in duration (*i.e.,* the difference between the earliest and latest queuing times of the message). The jitter of a given message *msg* is denoted QJ_{msg}. In any realistic system all messages will have some queuing jitter.

A given message is assigned a fixed identifier (and hence a fixed priority). We assume that each given hard real-time message must be of bounded size (*i.e.,* contain a bounded number of bytes). Given a bounded size, and a bounded rate at which the

message is sent, we effectively bound the peak load on the bus, and can then apply scheduling analysis to obtain a latency bound for each message.

We assume that there may also be an unbound number of soft real-time messages; these messages have no hard deadline, and may be lost in transmission (for example, the destination processor may be too busy to receive them). They are sent as 'added value' to the system (*i.e.,* if they arrive in reasonable time then some quality aspect of the system is improved).

As mentioned earlier, the queuing of a hard real-time message can occur with *jitter* (variability in queuing times). Figure 9 illustrates this.

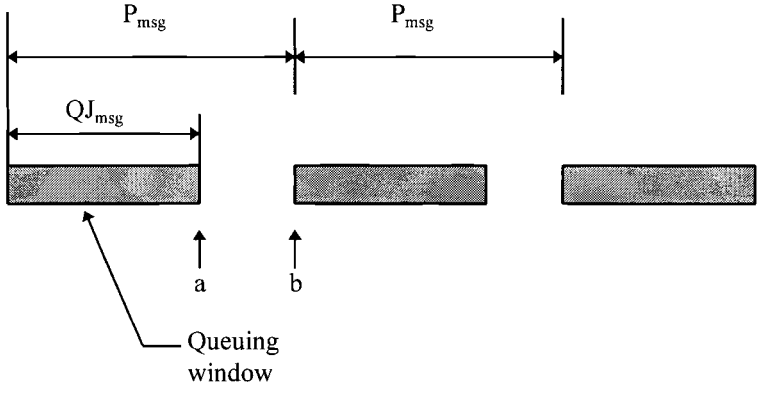


Figure 9. Message queuing jitter

The shaded boxes in the above diagram represent the 'windows' in which a task on the host CPU can queue the message. Queuing jitter can be defined as the difference between the earliest and latest possible times a given message can be queued.

The diagram above also shows how the *period* of a message can be derived from the task sending the message. For example, if the message is sent once per invocation of the task, then the message inherits a period, denoted P_{msg} , equal to the period of the task.

The longest time taken to transmit a given message *msg* we denote as CT_{msg}. for an eight byte message (the largest message permitted with CAN) transmitted on a 1 Mbit/sec network, CT_{msg} is 130μs (64 bits for the data, 47 bits of overhead — CRC and identifier fields, *etc.* — and up to 19 stuff bits).

3.3 Message Classification

Hard real-time messages fall into two categories: periodic messages and sporadic messages. A distributed hard real-time system will typically contain both types of messages.

1. Periodic Messages

A periodic message is one that is generated repetitively in fixed time intervals. A periodic message *msg* can be described by a quadruple (R_{msg}, CT_{msg}, DT_{msg}, P_{msg}), where

- R_{msg} is the release time, *i.e.*, the duration of the time interval between the begining of a period and the earliest time that an transmissiom of message *msg* can be started in each period.
- CT_{msg} is the transmission time.
- DT_{msg} is the deadline, *i.e.*, the duration of the time interval between the begining of a period and the time by which an transmission of message *msg* must be completed in each period.
- P_{msg} is the period, the minimal interval between transmission of message *msg*.

A periodic message *msg* can have an infinite number of periodic message transmissions msg_0, msg_1, msg_2, ..., with one message transmission for each period. For the *i*th message transmission msg_i corresponding to the *i*th period,

msg_i's release time:

$$R_{msg_i} = R_{msg} + P_{msg} \times (i\text{-}1) \qquad i \neq 0,\ i = 1, 2, \ldots$$

msg_i's deadline:

$$DT_{msg_i} = DT_{msg} + P_{msg} \times (i\text{-}1) \quad i \neq 0,\ i = 1, 2, \ldots$$

For an example of a period message, see Figure 10.

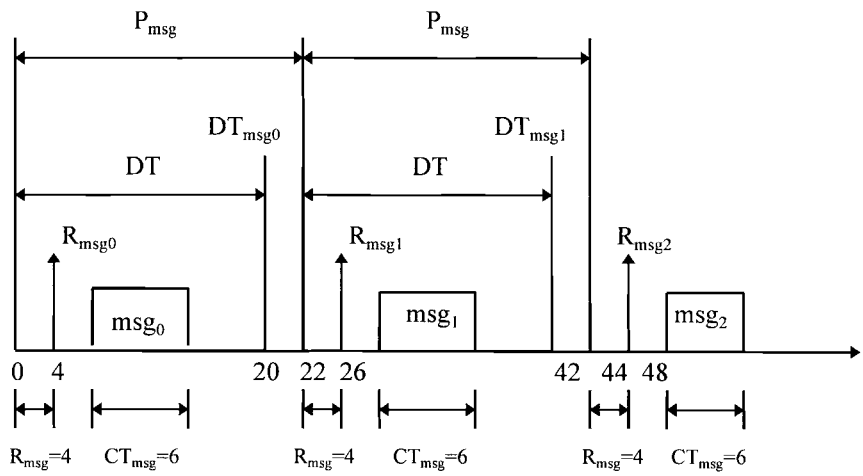


Figure 10. Periodic message.

2. Sporadic Messages

The sporadic message that is asynchronous in nature is one that is generated in response to an internal or external event, and has an arbitrary arrival time and deadline. A sporadic message msg_A can be described by a triple (CT_A, DT_A, min_A) where

- CT_A is the worst case transmission required by message msg_A.

- DT_A is the deadline; *i.e.*, the duration of the time interval between the time when a request is made for message msg_A and time by which an transmissionof msg_A must be completed.

$$DT_{A_i} = R_{A_i} + DT_A$$

For an example of a sporadic message, see Figure 11.

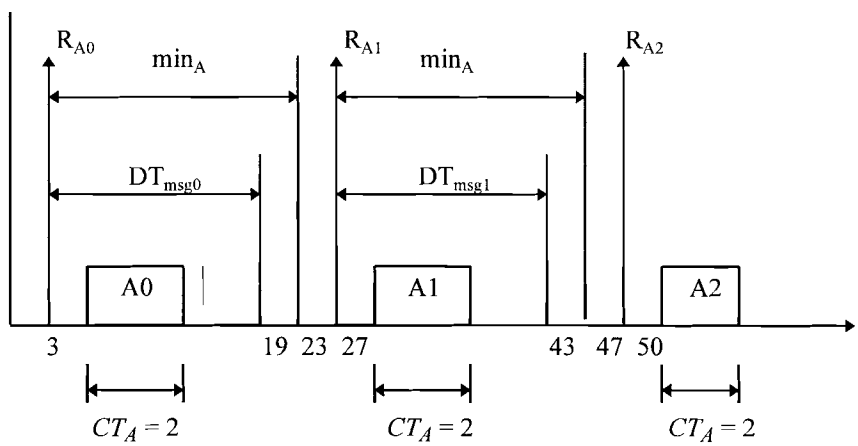
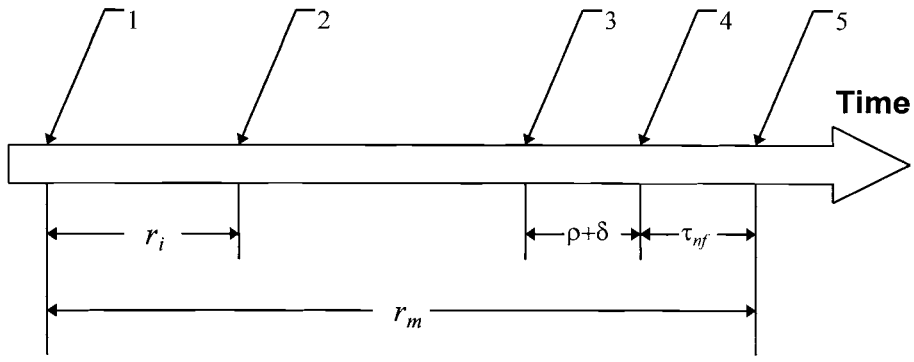


Figure 11. Sporadic message

3.4 Timing Model for CAN

The primary goal of the analysis in this thesis is to bound the time between the arrival of a message at the sending task, and the time at which the last packet of the message reaches the destination task, which we term the worst-case response time of the message. The following diagram illustrates these times:



1		Arrival at the sending task
2		Latest queuing time of message *msg*
3		Last packet of message m removed from packet queue. ρ is the worst-case time taken to transmit a packet, and δ is the electrical propagation delay.
		Transmision of last packet begins
4		Last packet of message reaches communications adapter
5		"Packet arrived" interrupt raised. τnf is the worst-case delay between the packet arriving at the communications adapter and the adapter notifying the processor of the packet arrival
		Arrival at destination task

Figure 12. Worst-case response time of a message

3.5 *Message Delays*

When using CAN to support distributed hard real-time systems, the application-to application delay is crucial in determining whether application deadlines will be satisfied. The application-to-application delay is the time delay experienced by a message that is sent between application tasks.

In CAN, protocols in layers above the MAC protocol are realized in the host processor where applications are executed, while the MAC protocol is implemented by a CAN network controller, such as the Intel 82527 or the Philips 82C200. With this kind of system implementation, the application-to-application delay experienced by a message *msg* can be decomposed as follows (Figure 13).

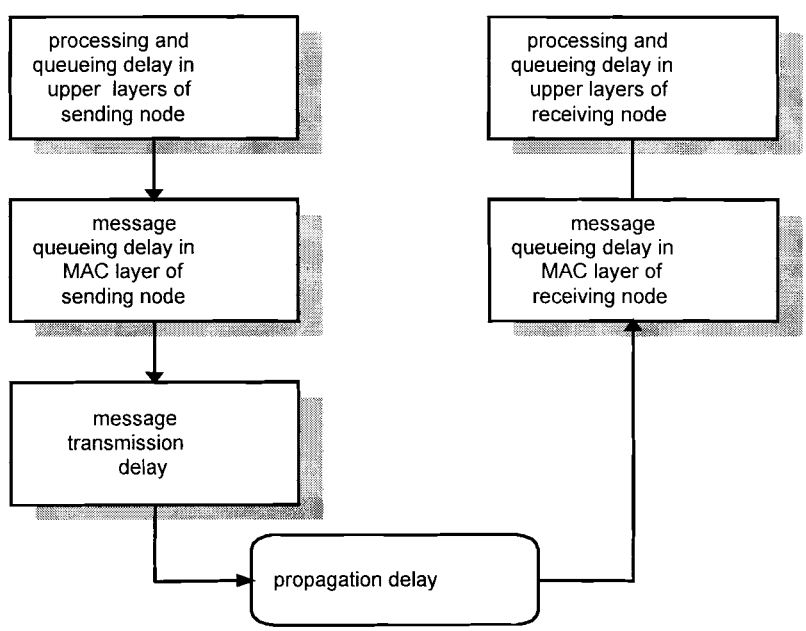


Figure 13. Application-to-Application Delay

Processing and queuing delay in the upper layers of the sending node($d_{upsend}(msg)$). The processing delay includes the time required to create message headers. The queuing delay includes both the time during which a message is waiting to be processed, and the time during which a message is waiting to be passed to lower layer protocols.

Queuing delay in the MAC layer of the sending node ($d_{MACsend}(msg)$). This delay occurs when a message is waiting to be sent by the MAC protocol.

Message transmission delay (C_{msg}). This delay is the time required to physically transmit the message.

Propagation delay (τ). This delay is the time required for a single bit to travel through the channel to the receiving node.

Message queuing delay in the MAC layer of receiving node ($d_{MACreceive}(msg)$). Once a message is received at its destination, it is stored in a queue at the MAC layer of the receiving node until the message can be passed to the upper layer protocols.

Processing and queuing delay in the upper layers of the receiving node($d_{upreceive}(msg)$). After a message is passed to the upper layer protocols, it may again be queued, before the message header is removed and the data passed to the receiving application task.

An upper bound on the application-to-application delay for message *msg*, $d_{TOT}(msg)$, can be written as:

$$d_{TOT}(msg) = d_{upsend}(msg) + d_{MACsend}(msg) + C_{msg} + \tau + d_{MACreceive}(msg) + d_{upreceive}(msg)$$

CHAPTER IV

BASIC PROCESSOR SCHEDULING THEORY

4.1 Objectives of Scheduling Algorithms

The function of a scheduling algorithm is to determine, for a given set of tasks, whether a schedule (the sequence and the time periods) for executing the tasks exists such that the timing, precedence, and resource constraints of the tasks are satisfied, and to calculate such a schedule if one exists. In static systems, a scheduling algorithm determines the schedule for a set of tasks off-line. However, in dynamic systems, because not all the characteristics of tasks are known *a priori*, a scheduling algorithm determines the schedule for tasks on-line *progressively*. A scheduling algoritm is said to *guarantee* a newly arriving task if the algorithm can find a schedule for all the previously guaranteed tasks and the new task such that each task finishes by its deadline. If a scheduling algorithm guarantees a task, it ensures that the task finishes by its deadline. A major performance metric for a dynamic scheduling algorithm is the *guarantee ratio*, which is the total number of tasks guaranteed versus the total number of task that arrive.

A *feasible schedule* is a schedule in which the start time of every task execution is greater than or equal to that task execution's release time or request time, and its completion time is less than or equal to that task execution's deadline. For an example of a feasible schedule, see Figure 14.

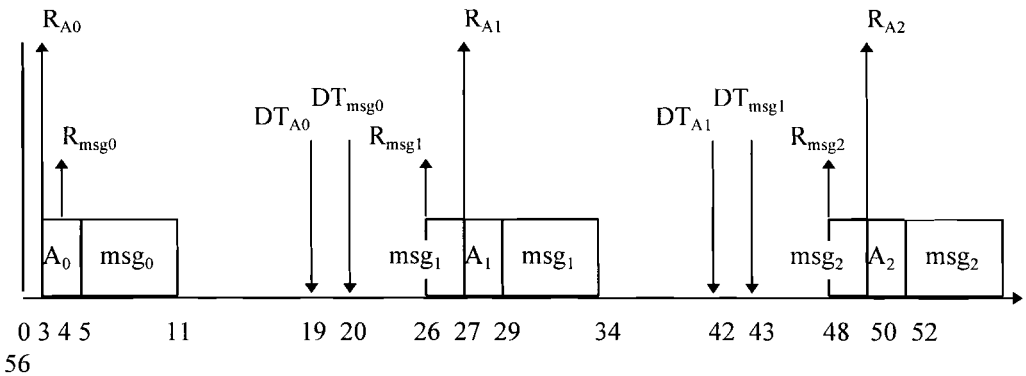


Figure 14. A feasible schedule.

In Figure 14, a feasible schedule for the period task executions msg_0, msg_1, msg_2 and the sporadic task executions A_0, A_1, A_2 in Figures 10 and 11 within the finite time interval [0, 56]. A_1 preempts msg_1 at time 27 and A_2 preempts msg_2 at time 50.

A static scheduling algorithm is said to be *optimal* if, for any set of tasks, it always produces a schedule which satisfies the constraints of the tasks whenever any other algorithm can do so. A dynamic scheduling algorithm is said to be *optimal* if it always produces a feasible schedule whenever a static scheduling algorithm with complete prior knowledge of all the possible tasks can do so. An optimal dynamic algorithm maximiaes the guarantee ratio of tasks that ever arrive in a system.

4.2 Approaches to scheduling

There are two distinct approaches to scheduling tasks in hard real-time systems. One is dynamic (*on-line*) scheduling, the other is static (*off-line*) scheduling.

In dynamic scheduling, the schedule for tasks is computed on-line as tasks arrive, and the scheduler does not assume any knowledge about the major characteristics of tasks that have not yet arrived in the system. Advantages of this approach include the following: it is unnecessary to know the major characteristics of the tasks in advance, and it is flexible and can easily adapt to changes in the environment. Often, the only stated disadvantage is its high run-time cost.

In static schedling, the schedule for tasks is computed off-line; this approach requires that major characteristics of the tasks in the system be known in advance. It is possible to use static scheduling to schedule periodic tasks. This consists of computing off-line a schedule for the entire set of periodic tasks occuring within a time period that is equal to the least common multiple of the periods of the given set of tasks and then executing the periodic tasks at run time in accordance with the previously computed schedule[42].

It is possible to translate an sporadic task into an equivalent periodic one. One technique for achieving this [45], is to translate each sporadic task (CT_A, DT_A, min_A) into a corresponding periodic task (R_{msg}, CT_{msg}, DT_{msg}, P_{msg}) that satisfies the following conditions:

$$CT_{msg} = CT_A,\ DT_A \geq DT_{msg} \geq CT_A$$

$$P_{msg} \leq \min\ (DT_A - DT_{msg} + 1,\ min_A),\ R_{msg} = 0 \text{ (see Figure 15).}$$

Thus it is possible to schedule sporadic tasks using static scheduling.

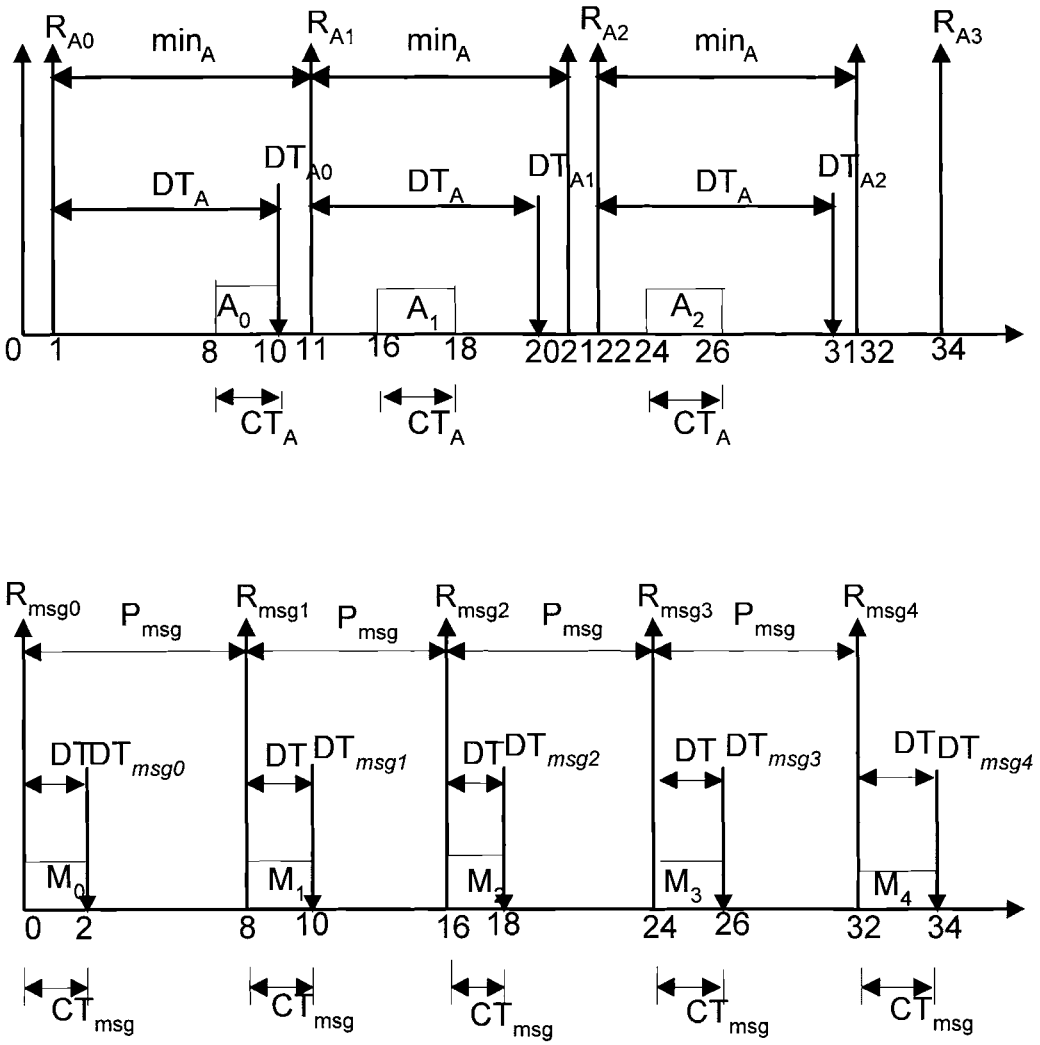


Figure 15 Sporadic task.

In figure 15, Sporadic task (CT_A, DT_A, min_A) where $CT_A = 2$, $DT_A = 9$, $\min_A = 10$.

Periodic task (R_{msg}, CT_{msg}, DT_{msg}, P_{msg}) translated from the sporadic task

$(CT_A, DT_A, min_A) = (2, 9, 10)$ where $R_{msg} = 0$, $CT_{msg} = CT_A = 2$, $DT_{msg} = CT_A = 2$, $P_{msg} = \min(DT_A - DT_{msg} + 1, \min_A) = \min(9 - 2 + 1, 9) = 8$. If periodic task executions msg_0, msg_1, msg_2, msg_3, msg_4, ... are scheduled to start at time 0, 8, 16, 24, 32, ... and if the sporadic request times R_{msg0}, R_{msg1}, R_{msg2}, R_{msg3} are 1, 11, 22, then the start times of the sporadic task executions A_0, A_1, A_2 are 8, 16, 24. A_0 executes in the time slot of msg_1, A_1 executes in the time slot of msg_2, A_2 executes in the time slot of msg_3.

4.3 Feasibility Conditions

Basic Liu and Layland Assumptions

The following restrictions are made regarding process timing and functional characteristics:

(i) all processes are periodic;

(ii) all processes have a deadline equal to their period;

(iii) all processes are independent;

(iv) all processes have a fixed computation time.

The underlying assumption regarding timing constrains is that all processes have $CT_i \leq DT_i = P_i$, and that at some point in time they have a common release time (often the assumption that $QJ_i = 0$ for all processes is made). In (Liu and Layland 1973)[27] assumption (iv) is relaxed to permit processes to have an actual execution time that is not fixed, whilst assuming that the process computation time is bounded.

Feasibility Testing of Static Priority Process Sets

The fundamental result regarding the feasibility of fixed priority process sets (assuming that all processes have $CT_i \leq DT_i = P_i$ and $QJ_i = 0$) is that only the first deadline of each process (at $DT_i = P_i$) need be checked. Time 0 (or any point in time

where all processes have a simultaneous release) represents the point in time at which the work-load on the processor is at a maximum: the demand of higher priority processes τ_1 $\ldots\tau_{i-1}$ in $[0, DT_i)$ is at a maximum, creating the hardest situation for τi to meet its deadline. The time when all processes are invoked simultaneously is termed a *critical instant*. Thus, if the deadline of a process is met for a release commencing at a critical instant, all subsequent deadlines will be met.

Based on the concept of a critical instant, Liu and Layland identified and proved the sufficiency of a utilization-based feasibility test appropriate for processes assigned priorities according to the rate-monotonic priority assignment policy:

$$\sum_{i=1}^{n} \frac{CT_i}{P_i} \leq n(2^{\frac{1}{n}} - 1) \qquad (1)$$

This implies that, if for a set of two processes, the combined utilization of those processes is no greater than 82.84%, the process set is feasible. As the cardinality of the process set approaches infinity, the permissible utilization approaches 69.31% (i.e., $ln(2)$). This test is sufficient but not necessary, as process sets with utilization greater than the level given by the above equation may still be feasible.

Development of Feasibility Analysis

One of the properties of the early utilization-based feasibility analysis is its simplicity, both in concept and computational complexity. However, the analysis also suffers from three major drawbacks:

(i) it is sufficient and not necessary;

(ii) it imposes unrealistic constrains upon the timing characteristics of processes; i.e., all processes must have $DT_i = P_i$;

(iii) process priorities have to be assigned according to the rate-monotonic policy (if priorities are not assigned in this way then the test is insufficient).

Following on from the early utilization-based analysis, several sufficient and necessary feasibility tests were developed. In general, such tests are known to have non-polynomial complexity (assuming that NP ≠ P). In 1980, Leung and Merrill [25] formulated the well-known approach of simulating the schedule over an interval equivalent to the least common multiple (LCM) of process periods. Unfortunately, this approach is inefficient as, even in the case of small process sets, the LCM can be very large.

Response time analysis was initiated by Harter in 1984 [15], who used a temporal logic proof system to derive the *Time Dilation Algorithm*. The algorithm can be restated, in the terminology adopted within this paper, as the following equation:

$$RT = CT_i + \sum_{j=1}^{i-1} \left\lceil \frac{RT}{P_j} \right\rceil CT_j \qquad (2)$$

If for any value of $RT \in [0, DT_i]$ the above condition holds, then process τ_i is feasible. Further, the smallest such value of RT equates to the worst-case response time of process τ_i. Equations of this form do not lend themselves easily to efficient analytical solutions. However, as observed by Harter, *et. al*, only a subset of time points in the interval need to be examined for feasibility. For example, if some value of $RT \leq DT_i$ is chosen and the above condition does not hold then the next value that may form a solution is given by the right-hand side of the equation. By noting that the summation term increases monotonically in RT, solutions can be found using a recurrence relation.

We note that the family of response time tests described above do not make any assumptions regarding the priority assignment policy used (i.e. rate-monotonic priority

assignment is not a requirement for the tests to be sufficient and necessary). Further, these tests are also applicable to processes with deadlines less than their periods.

Step-by-Step Example

The following steps illustrate how to apply the equation (2) disscussed above to task 3. The characteristics of tasks 1, 2, and 3 are defined in the following table:

Task	Execution Time *CT*	Arrival Period *P*	Priority	Deadline *DT*
task 1	40	100	high	100
task 2	40	150	medium	150
task 3	100	350	low	350

Step 1 Compute the first approximation.

$$RT^0 = \sum_{j=1}^{i} CT_j$$

$$RT^0 = 100 + 40 + 40 = 180$$

The first approximation is simply the sum of the execution times of all higher priority tasks and task 3. This first approximation of the response time does not take into consideration the preemption from task 1 at 100 nor the preemption from task2 at 150.

Step 2 Calculate the next approximation.

$$RT^{n+1} = CT_i + \sum_{j=1}^{i-1} \left\lceil \frac{RT^n}{P_j} \right\rceil CT_j$$

$$RT^1 = 100 + \left\lceil \frac{180}{100} \right\rceil 40 + \left\lceil \frac{180}{150} \right\rceil 40 = 260$$

This step accounts for the preemption at 100 and 150 for tasks 1 and 2 respectively, resulting in the next approximation. However, this latest approximation does not account for the preemption from task 1 at 200. Notice that the term, $\left\lceil \frac{180}{100} \right\rceil 40$, represents the total amount of execution time requested by task 1 from 0 to 180. In general, term, $\left\lceil \frac{RT_n}{P_i} \right\rceil CT_i$, represents the total amount of execution time requested by task i from time 0 to time RT^n.

Step 3 Determine if the approximation is the answer.

$$RT^2 = 100 + \left\lceil \frac{260}{100} \right\rceil 40 + \left\lceil \frac{260}{150} \right\rceil 40 = 300$$

$$RT^3 = 100 + \left\lceil \frac{300}{100} \right\rceil 40 + \left\lceil \frac{300}{150} \right\rceil 40 = 300$$

Since (RT^3 is less than 350) and (RT^3 is equal to RT^2), the technique terminates. Worst-case response time is 300. Since 300 is less than the deadline, 350, task 3 is schedulable. This last iteration accounts for the preemption at 200 plus all prior preemption, and there is no additional preemption to account for. Thus 300 is the worst-case completion time for task 3. Figure 16 shows the timeline for task 1, 2 and 3.

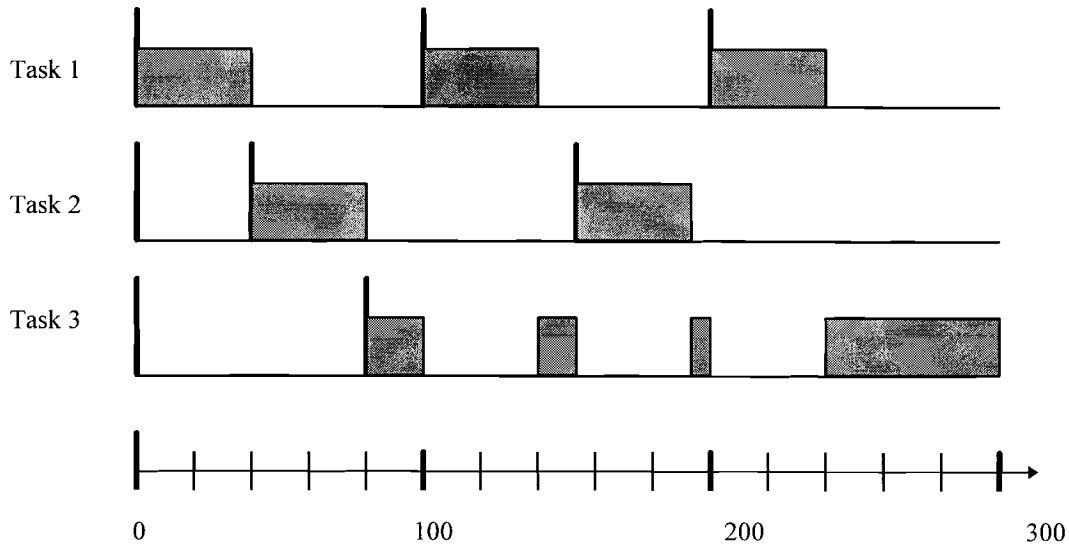


Figure 16. Timeline for tasks 1, 2 and 3

4.4 Priority Assignment

Early work in fixed priority pre-emptive scheduling produced the (optimal) rate-monotonic priority assignment policy (assuming Liu and Layland's constraints upon the timing characteristics of processes). One side effect of relaxing the constraint that process deadlines must be equal to their respective periods is that rate-monotonic priority assignment is no longer optimal (Leung and Whitehead 1982) [26].

In 1982, the *deadline monotonic* policy was proposed for processes having $CT_i \leq DT_i = P_i$ and $QJ_i = 0$. With this policy, priorities are assigned in a similar manner to rate-monotonic: the shortest deadline process is assigned the highest priority; processes with successively longer deadlines are assigned successively lower priorities. We note that deadline-monotonic priority assignment is equivalent to rate-monotonic priority assignment when, for all processes $DT_i = P_i$. Deadline-monotonic priority assignment is optimal in a similar manner to rate-monotonic: if there exists a feasible priority ordering

over a set of processes, a deadline-monotonic priority ordering over those processes will also be feasible.

Rate-monotonic and deadline-monotonic priority assignments assume that all processes share a critical instant (i.e. common release time). If processes are permitted to have arbitrary offsets, then this condition may not hold. Under these circumstances neither priority assignment policy is optimal. Indeed, whilst rate-monotonic and deadline-monotonic priority assignments can be achieved in polynomial time, Leung, *et. al* questioned whether the same applied to priority assignments for processes with no common release time. Later, Audsley showed that optimal priority assignment can be achieved by examining a polynomial number of priority orderings over the process set (i.e. not $n!$) assuming an exact (pseudo-polynomial) feasibility test (Audsley 1993)[4].

4.5 Usage of Theory on Hard Real-Time Scheduling for CAN

Scheduling messages on a CAN bus is analogous to scheduling tasks with fixed priorities. It is possible to take the existing analysis and apply it to CAN messages.

Audsley, *et. al* [6] and Burns, *et. al* [9] show how the analysis of Joseph and Pandys [24] can be updated to include blocking factors introduced by periods of non-preemption, release jitter, and accurately take account of a task being non-preemptive for an interval before termination. The following equations represent this analysis:

$$RT_i = QJ_i + w_i + CT_i \qquad (3)$$

where w_i is the least fixed-point of the following recursive relation.

$$w_i^{n+1} = BT_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{w_i^n + QJ_j + \tau_{res}}{P_j} \right\rceil CT_j \qquad (4)$$

Where *hp(i)* is the set of tasks of higher priority than task *i*, CT_i is the worst-case computation time required by a given task *i*, and P_j is the period of a given task *j*. BT_i is the blocking factor of task *i* (a bound on the time that a lower priority task can execute and prevent the execution of task *i*); the priority ceiling protocol [37] controls this 'priority inversion' and defines how BT_i can be computed. Variable τ_{res} is the resolution with which we measure time. On a CAN bus we deal with time units as multiples of the bit-time, which we denote as τ_{bit}; with a 1 Mbit/sec bus, this is equal to 1 μs.

Variable *QJi* is the release jitter of task *i*, analogous to the queuing jitter of a message.

The feasibility of a given task can be trivially assessed by comparing the worst-case response time of the task againt its deadline. Note that the deadline of a given task *i*, denoted DT_i, is assumed to be less than or equal to P_i. Another assumption is that a task cannot voluntarily suspend itself (and hence the processor cannot be idle when tasks have work to do).

Equation above describes a recurrence relation, where the (*n*+1)th approximation to the value of w_i is found in terms of the *n*th approximation, with the first approximation set to zero. A solution is reached when the (*n*+1)th approximation equals the *n*th.

Having introduced this analysis we can apply it to CAN bus scheduling. We do this by first deriving a simple CAN model analysis, and then discussing how the analysis is affected by the behaviour of implemented hardware.

CHAPTER V

CALCULATING CAN MESSAGE RESPONSE TIMES

5.1 Analysis of a Simple CAN Model

In this chapter we develop a simple analysis for the CAN model. In reality, CAN is more complex than described, and later sections will extend the analysis to cover these complexities. The analysis of the previous chapter can be applied to simple CAN by the analogy between task scheduling and message scheduling: a task is released at some time (i.e. is placed in a priority ordered queue of runnable tasks), and contends with other tasks (both lower and higher priority tasks) until it becomes the highest priority task in the runnable task.

Because of the operation of the priority ceiling protocol, a task need only contend with at most one lower priority task. In addition, it contends with all higher priority tasks until these have all completed and the processor is freed. With the model of Burns, *et. al* [6], the task is then dispatched and runs until completion. Upon completion it is returned to the waiting queue until next made runnable.

The same behavior holds for CAN messages: a message is queued at some time, and contends with other messages until it becomes the highest priority message. It commences transmission, and is transmitted without interruption until completion. Note that this assumes that the bus cannot become idle between the transmission of messages if there are pending messages (this is analogous to the assumption that a task must not voluntarily suspend itself).

The worst-case response time of a given message *msg* is the longest time between the queuing of a message and the time the message arrives at destination stations. A message is said to be *schedulable* if and only if:

$$RT_{msg} \leq DT_{msg}$$

We have a restriction on the worst-case response time: a queued message must be sent before the next queuing of the message (we want to prevent the overwriting of a message). Thus we must also have:

$$RT_{msg} \leq P_{msg} - QJ_{msg}$$

From this we can see that the message queuing window(i.e. the message queuing jitter) must be less than the periodicity of the message. We now develop analysis to determine the worst-case response time of a given message *msg*.

The worst-case response time is composed of two delays: the *queuing delay* and the *transmission delay*. The queuing delay is the longest time that a message can be queued in a station and be delayed because other higher and lower priority messages are being sent on the bus. We denote this time as QT_{msg}. The transmission delay is the time taken to actually send the message on the bus. This time is denoted CT_{msg}. The worst-case response time is thus defined as:

$$RT_{msg} = QT_{msg} + CT_{msg}$$

The queuing delay QT_{msg} is itself composed of two times: the longest time that any lower priority message can occupy the bus, and the longest time that all higher priority messages can be queued and occupy the bus before the message *msg* is finally transmitted. These times are called the *blocking time* (denoted it as BT_{msg}) or the *interference*. From earlier scheduling theory [5], the interference from higher priority messages over an interval of duration t is:

$$\sum_{\forall j \in hp(m)} \left\lceil \frac{t + QJ_j + \tau_{bit}}{P_j} \right\rceil CT_j$$

τ_{bit} is the time taken to transmit a bit on CAN and *hp(msg)* is the set of messages in the system of higher priority than messge *msg*. Note that the set *h(msg)* defines a priority ordering. From other work we know that the optimal priority ordering is deadline monotonic [26]. In fact, in the presence of queuing jitter, the optimal ordering is to select priorities on the basis of:

$$DL_{msg} - QJ_{msg}$$

That is, the smaller the value of $DL - QJ$ the higher the message priority [5]. Recall that, QJ_{msg} is the queuing jitter of message *msg*, inherited from the worst-case response time $R_{sender(msg)}$ (where *sender(msg)* denotes the task queuing message *msg*).

From the above description we can see that the queuing delay is given by:

$$QT_{msg} = BT_{msg} + \sum_{\forall j \in hp(msg)} \left\lceil \frac{QT_{msg} + QJ_j + \tau_{bit}}{P_j} \right\rceil CT_j \qquad (6)$$

Where the term BT_{msg} is the worst-case blocking time of message *msg*, and is analogous to the blocking factor defined by the analysis of the priority ceiling protocol. BT_{msg} is equal to the longest time taken to transmit a lower priority message, and given by:

$$BT_{msg} = \max_{\forall k \in lp(msg)} (CT_k)$$

lp(msg) is the set of messages in the system of lower priority than message *msg*. If *msg* is the lowest priority message then BT_{msg} is zero (just as the lowest priority task has a blocking factor of zero with the priority ceiling protocol).

CT_{msg} is the longest time taken to transmit message *msg*. As mentioned earlier, CAN has a 47 bit overhead per message, and a stuff width of 5 bits. Only 34 of the 47 bits of overhead are subject to stuffing, so CT_{msg} can be defined by:

$$CT_{msg} = \left(\left\lfloor \frac{34 + 8s_{msg}}{5} \right\rfloor + 47 + 8s_{msg} \right) \tau_{bit}$$

where s_{msg} is the number of data bytes in the message. Equation 6 above can be solved in the same way as Equation 4.

We desire the smallest value satisfying the above equation. Unfortunately, the above equation cannot be re-arranged to give a solution for QT_{msg}. However, a recurrence relation can be formed:

$$QT_{msg}^{n+1} = BT_{msg} + \sum_{\forall j \in h(msg)} \left\lceil \frac{QT_{msg}^{n} + QJ_j + \tau_{bit}}{P_j} \right\rceil CT_j$$

Because the recurrence relation is monotonically increasing in QT_{msg}, i.e., $QT_{msg}^{n+1} \geq QT_{msg}^{n}$, we need to start the iteration with a value of $QT_{msg}^{0} = 0$, and the iteration terminates when $QT_{msg}^{n+1} = QT_{msg}^{n}$.

5.2 Extending The Model: Error Handling and 'RTR' Messages

In the previous sections we described briefly the CAN architecture and protocol. However, we made two simplifications: we ignored error handling, and we did not address a special type of message called a Remote Transmission Request message. In this section, we describe a model for error handling, discuss remote transmission request (RTR) messages, and extend the analysis to handle the full CAN Model.

Error Frame

The error frame is a means by which any node in the system may indicate to all others the detection of an error condition. The error flag consists of six consecutive dominant bits and is recognized by all other nodes as an error condition due to violation of the bit-stuffing rules. Due to different error flags being superimposed, the flag may consist of up to a maximum of 12 dominant bits. On detection or transmission of an error flag, all nodes will monitor the bus for a recessive bit and will then transmit six recessive bits before continuing as shown in Figure 17.

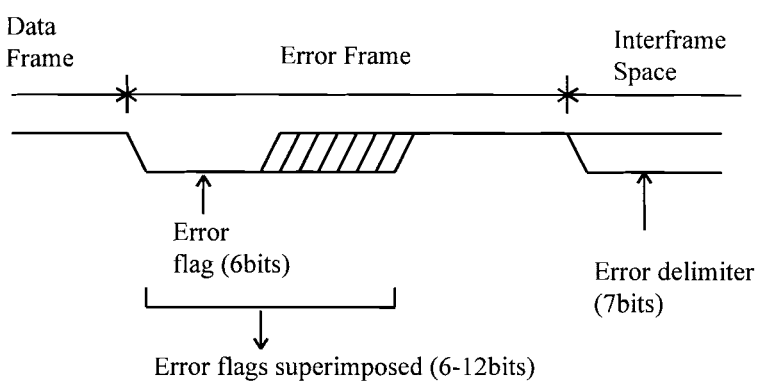


Figure 17. Error frame

Error Handling

CAN has an effective error detection mechanism: an error detected by either the sender of a message, or receiver of the message, is signalled to the sender. The sender then re-transmits the message. In the worst-case, upon detection of an error, the recovery process requires the transmission of up to 19 bits (plus the retransmission of the message). To include the costs of error handling, in the previous analysis, we define the function $E(t)$; the most probable bound on the overheads due to errors in an interval of

duration t. We include in this function the costs of retransmission. This function can be defined using statistical analysis based on observed error characteristics of a given configuration of CAN in a given environment. Each detected error implies the retransmission of a message. We assume that as soon as the sending station detects an error in the transmission of a message it immediately requeues the message for transmission. The assumption is an important one for the following reason: if the message is not immediately requeued then the bus may become idle and a lower priority message may attain access to the bus (and then begin transmission). This means that the message being retransmitted may be again delayed by a lower priority message. In general, therefore, a given message *msg* would be delayed by lower priority messages for up to time (n+1)B, where n is the number of re-transmissions of message *msg*. This would needlessly add to the worst-case response time of the message.

A probable bound on the error recovery overheads before a message *msg* arrives at the destination is:

$$E(RT_{msg})$$

Now that we have defined the overheads due to error handling for the transmission of a given message *msg*, we can include these overheads in the analysis developed in the previous section. We update Equation 4 to:

$$QT_{msg} = E(QT_{msg} + CT_{msg}) + BT + \sum_{\forall j \in h(m)} \left\lceil \frac{QT_{msg} + QJ_j + \tau_{bit}}{P_j} \right\rceil CT_j \quad (7)$$

Note that we have rewritten RT_{msg} as QT_{msg}+ CT_{msg}

RTR message

We now describe CAN Remote Transmission Request (RTR) messages (Figure 18). This message is a special CAN message with a zero length data field. It is interpreted by

stations to mean "please transmit the message with the same identifier as this message". Because identifiers are unique within the system, there can only be one station that responds to this message (if no stations respond by transmitting the requested message then an error is flagged). We make the assumption that a station responding to an RTR message will immediately queue the requested message for transmission such that no lower priority message can be transmitted first (for the same reasons described earlier for the assumption that retransmissions occur immediately). Of course, if a higher priority message has been queued since the transmission of RTR message,

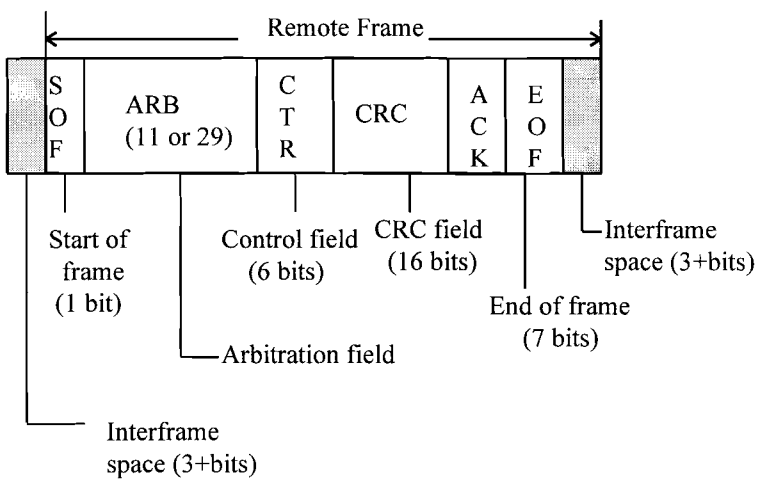


Figure 18. Remote Frame Format

then the higher priority message will be transmitted after the RTR message has been sent and before the requested message is sent.

A number of stations may transmit RTR messages, and one station may transmit the corresponding requested message; all of these messages have the same identifier, and hence priority. This complicates the analysis slightly; previously, messages were assumed to have unique identifiers and the set *hp*(*msg*) for a given message msg indicated all the

messages that could win the arbitration process and delay the transmission of msg. However, with the introduction of RTR messages this is no longer true. This problem is addressed by CAN in two ways. First, although a number of stations can simultaneously attempt to transmit RTR messages with the same identifier, no collision results. This is because RTR messages with the same identifier have identical bit patterns (recall that RTR messages are zero byte messages): no station will see other than the data transmitted. Second, the CAN message arbitration gives priority to requested messages over RTR messages with the same identifier.

One way to address the problem of interference between RTR and requested data messages all with the same identifier is to change the set *h(msg)* to include all messages of higher or the same priority. However, this is pessimistic, since it is possible for CAN to prevent the interference if we are careful with how we define the semantics of an RTR message. We say that the worst-case response time of an RTR message is defined as the longest time between queuing the RTR message and the requested message arriving at destination stations. This definition means that if an RTR message is queued at some time, but that before the RTR message is transmitted the requested data message is received (in response to an earlier RTR message, say), the response time is the time between the still-untransmitted RTR message and the reception of the requested data message. Thus, the RTR message could be satisfied before it has been transmitted.

Because of this definition of RTR response time we do not have to consider the interference between data messages and RTR messages of the same priority. We now continue, and define new notation: the time $C_{RTR(msg)}$ is the value of C_{msg} for the requested message as well as the time to transmit RTR messages, where *msg* is a given message. If *msg* is not an RTR message then we define $C_{RTR(msg)}$ to be zero.

Because the worst-case response time of an RTR message includes the time taken to transmit the requested message, we must re-define the equation for the worst-case response time of a given message *msg*:

$$RT_{msg} = QT_{msg} + C_{RTR(msg)} \quad \text{if } msg \in RTR$$
$$RT_{msg} = QT_{msg} + CT_{msg} \qquad \text{otherwise}$$

Where RTR is the set of RTR messages. The term QT_{msg} represents the queuing delay for message *msg* (as before), but this queuing delay must also include the time taken to transmit the RTR message. The interference from higher priority RTR messages must include the transmission of the corresponding data message. Equation 7 is therefore updated to:

$$QT_{msg} = CT_{msg} + E(QT_{msg} + C_{RTR(msg)}) + BT + \sum_{\forall j \in h(msg)} \left\lceil \frac{QT_{msg} + QJ_j + \tau b_{it}}{P_j} \right\rceil (CT_j + C_{RTR(j)}) \qquad (8)$$

CHAPTER VI

ANALYSIS FOR DIFFERENT CONTROLLERS

6.1 Intel 82527

General Features

The 82527 serial communications controller is a highly integrated device that performs serial communication according to the CAN protocol. The CAN protocol uses a multi-master (contention based) bus configuration for the transfer of "communication objects" between nodes of the network. This multi-master bus is also referred to as CSMA/CD. The 82527 performs all serial communication functions such as transmission and reception of messages, message filtering, transmit search, and interrupt search with minimal interaction from the host microcontroller, or CPU.

The 82527 is Intel's first device to support the standard and extended message frames in CAN Specification 2.0 part B. It has the capability to transmit, receive, and perform message filtering on extended message frames with a 29-bit message identifier.

The 82527 features a powerful CPU interface that offers flexibility to directly interface to many different CPUs. It can be configured to interface with CPUs using an 8-bit multiplexed, 16-bit multiplexed, or 8-bit non-multiplexed address/data bus for Intel and non-Intel architectures. A flexible serial interface is also available when a parallel CPU interface is not required.

The 82527 provides storage for 15 message objects of 8-byte data length. Each message object can be configured as either transmit or receive except for the last message object. The last message object is a receive only buffer with a special acceptance mask designed to allow select groups of different message identifiers to be received.

The 82527 also implements a global acceptance masking feature for message filtering. This feature allows the user to globally mask any identifier bits of the incoming message. The programmable global mask can be used for both standard and extended messages.

The 82527 provides an improved set of network management and diagnostic functions including fault confinement and a built-in development tool. The built-in development tool alerts the CPU when a global status change occurs. Global status changes include message transmission and reception, error frames, or sleep mode wake-up. In addition, each message object offers full flexibility in detecting when a data or remote frame has been sent or received.

Functional Overview

The 82527 CAN controller consists of six functional blocks. The CPU Interface logic manages the interface between the CPU and the 82527 using an address/data bus. The CAN controller interfaces to the CAN bus and implements the protocol rules of the CAN protocol for the transmission and reception of messages. The RAM is the interface layer between the CPU and the CAN bus. The two port blocks provide 8-bit low speed I/O capability. The clockout block allows the 82527 to drive other chips, such as the host-CPU.

The 82527 RAM provides storage for 15 message objects of 8-byte data length. Each message object has a unique identifier and can be configured to either transmit or receive except for the last message object. The last message object is a receive only buffer with a special mask design to allow select groups of different message identifiers to be received.

Each message object contains control and status bits. A message object with the direction set as receive will send a remote frame by requesting a message transmission. A message object with the direction set as transmit will be configured to automatically send a data frame whenever a remote frame with a matching identifier is received over the

CAN bus. All message objects have separate transmit and receive interrupts and status bits,

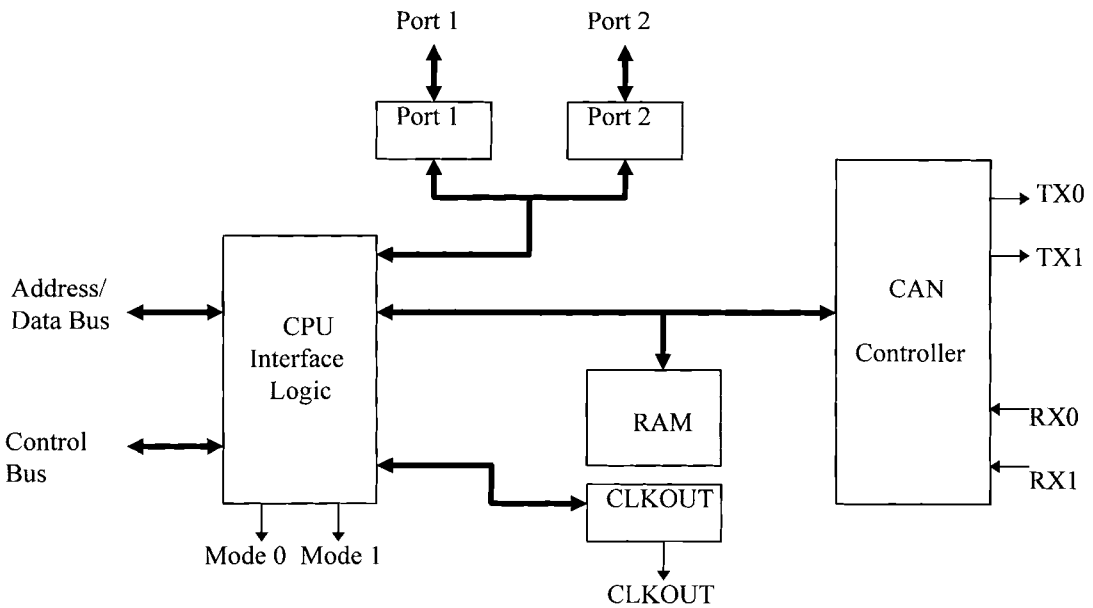


Figure 19. A block diagram of the 82527

allowing the CPU full flexibility in detecting when a remote or data frame has been sent or received.

The 82527 also implements a global masking feature for acceptance filtering. This feature allows the user to globally mask, or "don't care", any identifier bits of the incoming message. This mask is programmable to allow the user to design an application-specific message identification strategy. There are separate global masks for standard and extended frames.

The incoming message first passes through the global mask and is matched to the identifiers in message objects 1-14. If there is no identifier match then the message passes through the local mask in message object 15. The local mask allows a large number of infrequent messages to be received by the 82527. Message object 15 is also buffered to allow the CPU time to service a message received.

82527 Message Objects

The message object is the means of communication between the host microcontroller and the CAN controller in the 82527. Message objects are configured to transmit or receive messages.

There are 15 message objects located at fixed addresses in the 82527. Each message object starts at a base address that is a multiple of 16 bytes and uses 15 consecutive bytes. For example, message object 1 starts at address 10H and ends at address 1EH. The remaining byte in the 16 byte field is used for other 82527 functions. In the above example the byte at address 1FH is used for the clockout register.

Message object 15 is a receive-only message object that uses a local mask called the message 15 mask register. This mask allows a large number of infrequent messages to be received by the 82527. In addition, message object 15 is buffered to allow the CPU more time to receive messages.

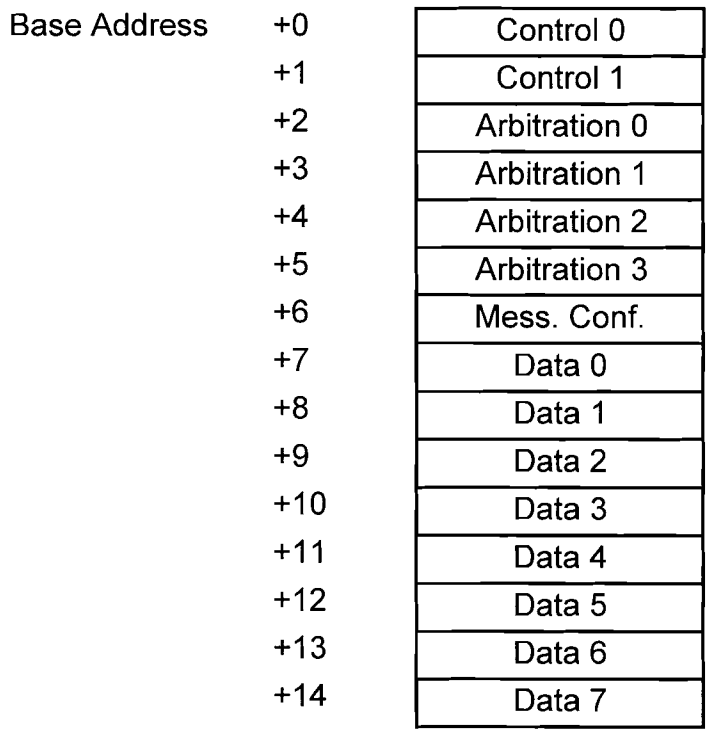


Figure 20. Message Object Structure

Message Object Priority

If multiple message objects are waiting to transmit, the 82527 will first transmit the message from the lowest numbered message object, regardless of message identifier priority.

If two message objects are capable of receiving the same message (possible due to message filtering strategies). the message will be received by the lowest numbered message object. For example, if all acceptance mask bits were set as "don't care", message object 1 will receive all messages.

Message Acceptance Filtering

The mask registers provide a method for developing an acceptance filtering strategy for a specific system. Software can program the mask registers to require an exact match on specific identifier bits while masking ("don't care") the remaining bits. Without a masking strategy, a message object could accept only those messages with an identical message identifier. With a masking strategy in place, a message object can accept messages whose identifiers are not identical.

The CAN controller filters messages by comparing an incoming message's identifier with that of an enabled internal message object. The standard global mask register applies to messages with standard (11-bit) identifiers, while the extended global mask register applies to those with extended (29-bit) identifiers. The CAN controller applies the appropriate global mask to each incoming message identifier and checks for an acceptance match in message objects 1-14. If no match exists, it then applies the message 15 mask and checks for a match on message object 15. The message 15 mask is ANDed with the global mask, so any bit that masked by the global mask is automatically masked for message 15.

Transmit message object ID	11000000000
Mask (0=don't care; 1=must match)	00000000011
Received remote message object ID	00111111100
Resulting message object ID	00111111100

Table 1 Effect of Masking on Message Identifiers

The CAN controller accepts an incoming data message if the message's identifier matches that of any enabled receive message object. It accepts an incoming remote message (request for data transmission) if the message's identifier matches that of any enable transmit message object. The remote message's identifier is stored in the transmit message object, overwriting any masked bits. Table 1 shows an example.

Real-time Behaviour of the Intel 82527

The queuing of messages in the Intel 82527 is undertaken in the controller and interfaced to the host processor via dual-ported RAM. The intention is to map permanently message identifiers to memory locations (termed *slots*), so that both outgoing and desired incoming messages are assigned unique slots.

A slot is tagged with a message identifier, and marked as an incoming or outgoing slot. If a message is received with the same identifier as a slot marked as incoming then the message contents are stored in that slot (the slot also contains an interrupt enable flag so that an interrupt can be raised when the message arrives). If the host processor wishes to initiate the transmission of the message then it is able to mark the message as ready for transmission.

Because of hardware limitations, only 15 slots are available for outgoing and incoming messages (instead of the ideal 2032 — the full range of CAN identifiers). However, these 15 slots can be programmed to map to any CAN identifier. The controller will transmit messages in order of slot number, rather than the message identifier, and therefore it is important that the messages are allocated to the slots in identifier order. It should be noted that in most envisaged automotive systems, 15 messages per station is sufficient[Ana 8].

There is also a dedicated double-buffered receive buffer: when a message has been received in the controller without errors, a "message received" interrupt may be raised on the host processor. If the identifier of the message does not match the identifier in one of

the slots in the controller then the interrupt handler must copy the contents of the message from the buffer and store it in main memory. The handler then issues a "removed message" signal to the controller, indicating that the receive buffer is free. This is needed because the receive buffer is double buffered: while the host processor is removing data from one buffer, the controller may be placing data in the other buffer. The controller needs to synchronize with the host processor in order to place data in a free buffer.

There is an implicit deadline on handling the "message received" interrupt: if the host processor fails to remove the data and signal "removed message" before the controller has received the subsequent message then any further incoming messages may be lost (the smallest time between two successive messages is 47 τ_{bit}).

In many ways the dual-ported memory approach is an elegant way of implementing a controller, but one drawback is that there is an implicit restriction on message deadlines: a message cannot be queued if the previous queuing of the same message has not yet been transmitted. Therefore, we must have $DT_{msg} \leq P_{msg}$ (an assumption also made by the scheduling analysis in this paper).

Apart from the limitations discussed, the Intel 82527 controller behaves as an ideal CAN controller with respect to the analysis derived in this paper.

6.2 Philips 82C200

General Description

The 82C200 is a highly integrated stand-alone controller for the CAN used with automotive and general industrial environments. The 82C200 contains all necessary features required to implement a high performance communication protocol. The 82C200 with a simple bus line connection performs all the functions of the physical and data-link layers. The application layer of an Electronic Control Unit (ECU) is provided by a microcontroller, to which the 82C200 provides a versatile interface. Figure 21 is the block diagram of 82C200.

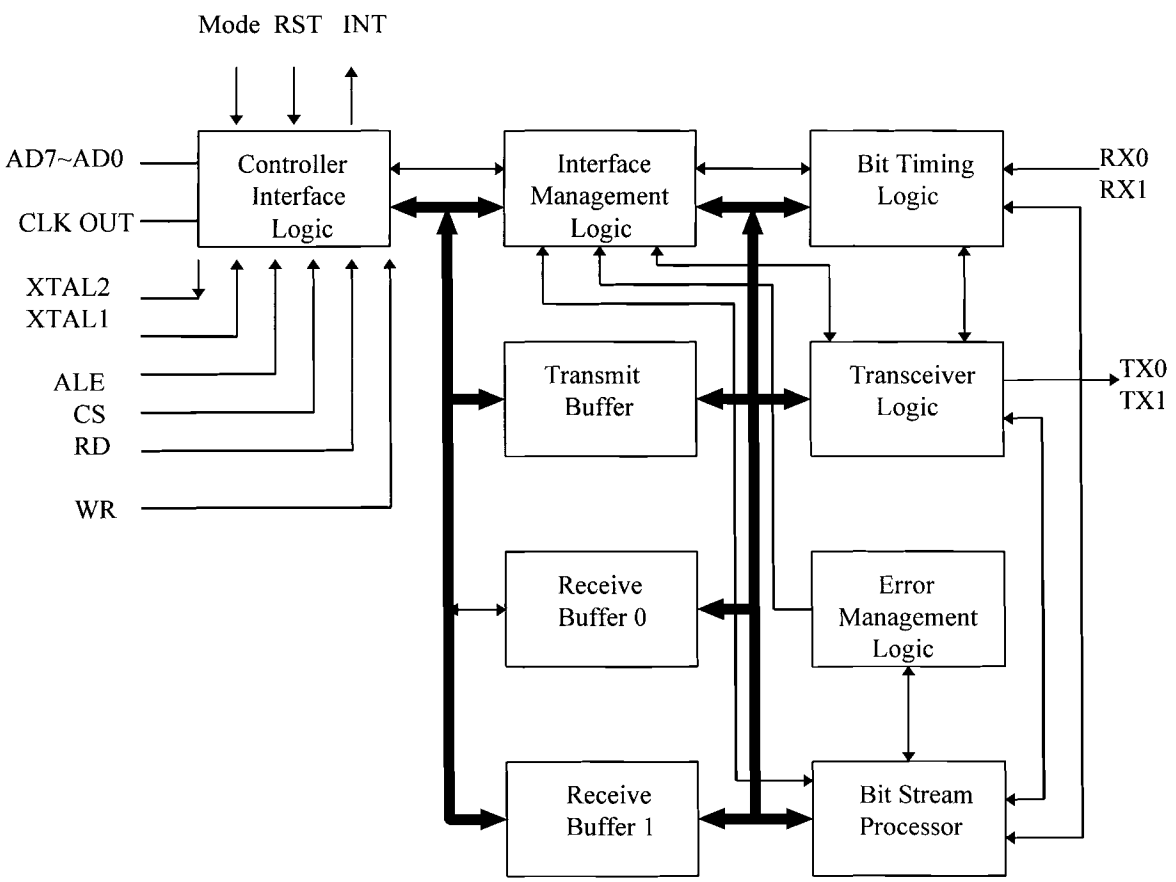


Figure 21. The Block Diagram of 82C200

Functional Description

The 82C200 contains all necessary hardware for a high performance serial network communication. The 82C200 controls the communication flow through the area network using the CAN-protocol. The 82C200 meets the following automotive requirements:

- short message length
- guaranteed latency time for urgent messages
- bus access priority, determined by the message identifier
- powerful error handling capability
- configuration flexibility to allow area network expansion.

The latency time defines the period between the initiation (Transmission Request) and the start of the transmission on the bus. Latency time is dependent on a variety of bus related conditions. In the case of a message being transmitted on the bus and one distortion the latency time can be up to 149 bit times (worst case).

Interface Management Logic (IML) interprets commands from the microcontroller, allocates the message buffers (TBF, RBF0 and RBF1) and provides interrupts and status information to the microcontroller.

Transmit Buffer (TBF) is a 10 byte memory into which the microcontroller writes messages which are to be transmitted over the CAN network.

Receive Buffers (RBF0 and RBF1) are each 10 byte memories which are alternatively used to store messages received from the CAN network. The CPU can process one message while another is being received.

Bit Stream Processor (BSP) is a sequencer, controlling the data stream between the Transmit Buffer, the Receive Buffer (parallel data) and the CAN-bus (serial data).

Bit Timing Logic (BTL) synchronizes the 82C200 to the bitstream on the CAN-bus.

Transceiver Control Logic (TCL) controls the output driver.

Error Management Logic (EML) performs the error confinement according to the CAN-protocol.

Controller Interface Logic (CIL) is the interface to the external microcontroller. The 82C200 can directly interface with a variety of microcontrollers.

Latency Time Requirements

1. Maximum allowed bit-time calculation

The maximum allowd bit-time (t_{BIT}) due to latency time requirements can be calculated as:

$$t_{BIT} \leq \frac{t_{MAX\,TRANSFER\,TIME}}{(n_{BIT,\,MAX\,LATENCY} + n_{BIT,\,MESSAGE})} \qquad \text{(a)}$$

Where:

- $t_{MAX\ TRANSFER\ TIME}$: the maximum allowed transfer delay time (application specific).
- $n_{BIT,\ MAX\ LATENCY}$: the maximum latency time (in terms of number of bits), which depends on the actual state of the CAN network (e.g. another message already on the network).
- $n_{BIT,\ MESSAGE}$: the number of bits of a message; it varies with the number of transferred data bytes $n_{DATA\ BYTES}$ (0...8) and Stuffbits like:

$$44 + 8.n_{DATA\ BYTES} \leq n_{BIT,\ MESSAGE} \leq 52 + 10.n_{DATABYTES} \qquad \text{(b)}$$

Examples:

For the calculation of $n_{BIT,\ MAX\ LATENCY}$ the following is assumed (the term 'our message' refers to that one the latency time is calculated for):

- since at maximum one-bit-time ago another CAN-controller is transmitting.
- a single error occurs during the transmission of that message preceding ours, leading to the additional transfer of one Error Frame.

- 'our message' has the highest priority,

giving:

$$n_{BIT,\ MAX\ LATENCY} \geq 44 + 8.n_{DATA\ BYTES,\ WORST\ CASE} + 18 \qquad (c)$$

$$n_{BIT,\ MAX\ LATENCY} \leq 52 + 10.n_{DATA\ BYTES,\ WORST\ CASE} + 18 \qquad (d)$$

Where:

- The additional 18 bits are due to the Error Frame and the Intermission Field preceding 'our message'.
- $n_{DATA\ BYTES,\ WORST\ CASE}$ denotes the number of data bytes contained by the longest message being used in a given CAN network.

2. Calculating the maximum bit-time

Table 2 illustrates calculation of the maximum bit-time

STATEMENT	COMMENTS
$t_{MAX\ TRANSFER\ TIME} = 10$ ms	assumption
$n_{DATA\ BYTES,\ WORST\ CASE} = 6$	longest message in that network; asumption
$n_{DATA\ BYTES} = 4$	'our message'; assumption
$n_{BIT\ MAX\ LATENCY} \leq 130$	using Equation (c) and (d)
$n_{MESSAGE} \leq 92$	using Equation (b)
$t_{BIT} \leq 10ms/(130+92) = 45\ \mu s$	using Equation (a)

Table 2. Example for calculating the maximum bit-time

Real-time Behavior of the Philips 82C200

In this section we discuss the behavior of the Philips 82C200 CAN controller, and show its worst-case real-time properties are poor (space limitations preclude the development of analysis for this controller).

The Philips controller is a simple controller, with two message buffers on-chip: a single 10 byte transmission buffer, and a 10 byte double-buffered receive buffer. The controller is typically interfaced to the processor as a memory mapped I/O device, and can raise two interrupts: "message received", and "message sent". The controller accepts three signals from the host processor: "send message", "abort message", and "removed message". The controller requires messages to be held on the host processor, and software drivers to copy the messages from the processor to the controller when appropriate.

To send a message, the host processor fills the transmit buffer with up to eight bytes of data, the identifier of the message, and some control bits, and then sends a "transmit message" signal to the controller. We denote the longest time to do this as τ_{copy}. The controller attempts to transmit the message according to the CAN protocol; when the message has been sent, a "message sent " interrupt is raised on the host processor.

The reception of messages is very similar to the Intel 82527 controller: when a message has been received in the controller without errors a "message received" interrupt is raised on the host processor and the interrupt handler must copy the 10 bytes of message data from the controller and store it in main memory.

The signal "abort message" is to aid in the writing of software device drivers for pre-emptive queuing. Without the signal, the real-time performance of the controller would be very poor indeed. Consider the situation where there is a low priority message in the transmit buffer of the controller, and a high priority message has just been queued by the host processor software. If the host processor were unable to remove the low priority message, then the high priority message would blocked until the low priority message is

sent. The low priority message will only be sent when all other higher priority traffic on the bus has finished: this could be very long.

Instead of succumbing to this problem, the device driver should abort the transmission of the low priority message, and copy the high priority message to the transmit buffer. The controller will only abort the message if it has not yet begun transmission. This is a sensible approach, since if the low priority message has begun transmission then there will be only a short delay (equal to the transmission time of the message) before the transmission buffer is freed.

There remains a major problem with the management of the transmission buffer: the time between "message sent" and the host processor copying the next message to the transmit buffer is non-zero (although short if the host processor is fast). In this short interval the bus could be claimed by a low priority message from another station and defer the transmission of the newly copied message. This problem also occurs when a message is pre-empted: the short interval between a lower priority message being aborted, and the higher priority message being copied into the buffer, releases the bus to low priority traffic.

For every pre-emption (i.e. when a message is aborted, and replaced by a higher priority message) in an interval, the bus may potentially be claimed twice by lower priority traffic at other stations: once when the higher priority message preempts, and once when the message has been transmitted.

To illustrate this, consider the following scenario: a message *M* is to be sent from a given station. Also sent from this station are a high priority message *H* and a low priority message *L1*. Other stations also have low priority traffic to send(messages *L2*, *L3*, *L4*). In this scenario, message *M* can be delayed four times by lower priority messages whilst being pre-empted just once. this is solely a result of the buffer management mechanism.

The first delay occurs when message *M* is queued: as mentioned earlier, the 82C200 controller is not able to abort a message if the message has begun transmission. Therefore

message *M* can be delayed by *L1*. After the message has been sent, the host processor copies message *M* to the transmission buffer(taking at most τ_{copy}). In this time the bus is released and may become idle, or may be claimed by lower priority messages from other stations. When message *M* has been copied to the buffer, and is ready for transmission, it may be delayed by a lower priority message that has just started transmission from another station (*L2*). Just before message *M* starts transmitting, a higher priority message *H* can preempt *M*: the 82C200 controller aborts message *M*, and copies the higher priority message to the transmission buffer. Again, the bus is released, and again lower priority message can be transmitted (*L3*), delaying both message *H* and message *M*. When message *H* has been transmitted the host processor copies message *M* back to the transmission buffer. Again, the bus is released, and again message *M* can be delayed (by *L4*).

It is straightforward to bound the delays due to this priority inversion, and the delays due to copying messages. This priority inversion can be very large, and lead to very poor worst-case performance of the controller. The following table details a set of messages based on the above scenario. They conform to the 'rate monotonic' model of deadlines equal to periods.

Message	***P***	***DT***	***CT***	***util***
H	605	605	47	7.8%
M	610	610	47	7.8%
L1	100000	100000	130	0.13%
L2	100000	100000	130	0.13%
L3	100000	100000	130	0.13%
L4	100000	100000	130	0.13%

In above table, all times are in microseconds. Messages are assumed to be queued with zero jitter.

A small value for τ_{copy} is assumed: large enough to release the bus to lower priority messages when copying a message to the transmission buffer, but not large enough to form a significant part of the response time of a message(in practice, such a small value would be unattainable).

The example message set is unschedulable: in the scenario described, we find that the response time of message *M* is 614 μs. The bus utilization in this example is just under 16%. By comparison, the worst-case response time of *M* with the Intel controller is 224μs. It is possible to find unschedulable scenarios with bus utilizations as low as 11%. Clearly using the Philips controller could lead to very poor resource utilization.

Note that in the situation where there is a large amount of low priority 'soft' real-time traffic on the bus, the impact of higher priority traffic on lower priority traffic sent from the same station is at least trebled when compared to the ideal CAN behavior (and when compared to the behavior of the Intel 82527), and that worst-case response times will therefore be very much larger.

REFERENCES

[1] Agrawal, G., B. Chen, and W. Zhao. 1993. Local synchronous capacity allocation schemes for guaranteeing message deadlines with the timed token protocol. In *Proceedings of INFOCOM '93*, pp. 186-193

[2] Agrawal, G., B. Chen, W. Zhao, and S. Davari. 1992. Guaranteeing synchronous message deadlines in high speed token ring networks with timed token protocol. In *Proceedings of the 12th IEEE International Conference on Distributed Computing Systems*, pp. 468-475

[3] Agrawal, G., Chen, B., Zhao, W., and Davari, S., "Architecture Impact of FDDI Network on Scheduling Hard Real Time Traffic," Workshop on Architectural Aspects of Real Time Systems (December 1991).

[4] Audsley, N. C. 1993. "Flexible Scheduling in Hard Real-Time Systems", Department of Computer Science, University of York, UK. YCST 9307

[5] Audsley, N., Burns, A., Richardson, M., Tindell, K., and Wellings, A., "Applying New Scheduling theory to Static Priority Pre-emptive Scheduling," Software Engineering Journal 8(5) pp. 284-292 (September 1993)

[6] Audsley, N., Burns, A., *et al,* Applying New Scheduling Theory to Static Priority Pre-emptive Scheduling, *Software Engineering Journal* 8(5), pp. 285-292(sept. 1993).

[7] ANSI Standard X3T9.5/88-139, Rev 4.0. 1990. *FDDI Media Access Control (MAC)*

[8] Burns, A., Nicholson, M., Tindell, K. and Zhang, N. "Allocating and Scheduling Hard Real-Time Tasks on a Point-to-Point distributed System", Proc. Workshop on Parallel and Dist. Real-Time Syst., pp. 11-20 (Apr. 1993)

[9] Burns, A., Nicholson, M. *et al,* Allocating and Scheduling Hard Real-Time Tasks in a Point-to-Point distributed System, Proc. *Workshop on Parallel and Dist. Real-Time Syst.*, pp. 11-20(Apr. 1993)

[10] CAN Specification Version 2.0, *Robert Bosch GmbH*, September, 1991

[11] Cheng, S. C., and Stankovic, J. A., Scheduling Algorithms for Hard Real-Time Systems, *Tutorial - Hard Real-Time Systems*, IEEE Computer Society Press, pp. 150 - 173

[12] Coviello, G. J. 1979. Comparative discussion of circuit vs. packed switched voice. *IEEE Trans. Commun.* COM-27, 8(Aug), 1153-1160

[13] Damm, A., Reisinger, W., Schwabl. W., "The Real-Time Operating System of MARS", *ACM Operating Systems Review*, 23(3), pp. [4] - [5] (1989)

[14] Grow, R.M. 1982. A timed token protocol for local area networks. In *Proceedings of Electro/82, Token Access Protocols,* paper 17/3

[15] Harter, P. K. 1984. "Response Times in Level Structured Systems". Department of Computer Science, University of Colorado, USA. CU-US-269-94

[16] Hong, J., X. Tan, and D. Towsley. 1989. A performance analysis of minimum laxity and earliest deadline scheduling in a real-time system. *IEEE Transactions on Computers*, 38(12):1736-1744

[17] IEEE Standard 802.5-1989. 1989. *Token Ring Access Method and Physical Layer Specifications*

[18] Intel, 82527 Serial Communications Controller Architectural Overview, Feb., 1995

[19] Intel, 8XC196K*x*, 8XC196*Jx*, 87C196CA Microcontroller Family User's Manual, June, 1995

[20] Jeffay, K., Stanat, D. F., and Martel, C. V., On Non-Preemptive Scheduling of Periodic and Sporadic Tasks, *Proceedings of the 12th IEEE Real Time Systems Symposium*, December 1991, pp. 129 - 139

[21] Joseph, M., and Pandya, P., Finding Response Times in a Real-Time System, *Computer Journal*. 29(5), pp.390-395 (Oct. 1986)

[22] Kurose, J. F., M. Schwartz, and Y. Yemini. 1984. Multiple-access protocols and time constrained communication. *Computer Surveys*, 16(1):43-70.

[23] Lehoczky, J., Sha, L., and Ding, Y., "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour," Proceedings of the 11th Real-Time Systems Symposium (December 1989)

[24] Lehoczky, J. P., "Fixed Priority Scheduling of Periodic Task Sets With Arbitrary Deadlines," Proceedings 11th IEEE Real-Time Systems Symposium pp. 201 - 209 (December 1990).

[25] Leung, J. Y. T. and Merrill, M. L. 1980. "A Note on Preemptive Scheduling of Periodic, Real-Time Tasks". Information Processing Letters. 11(3): 115-118

[26] Leung, J. Y. T. and Whitehead, J. 1982. "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks". Performance Evaluation. 2(4): 237-250

[27] Liu, C.L., and J.W.Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46-61

[28] Liu, M.T. 1978. Distributed loop computer networks. *Advances in Computers*, 17:163-221

[29] Malcolm, N., W.Zhao, and C. Barter. 1990. Guarantee protocols for communication in distributed real-time systems. In *Proceedings of IEEE infocom'90*, pp. 1078-1086

[30] MIT DSN 1982 Workshop on Distributed Sensor Networks (Lexington, Mass., Jan.). MIT Lincoln Laboratories, Lexington, Mass.

[31] Panwar, S. S., D. Towsley, and J. K. Wolf. 1988. Optimal scheduling policies for a class of queues with customer deadlines to the beginning of service. *Journal of the ACM.* 35(4):832-844

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

[33] Philips Semiconductors, 80C51 - Based 8 - Bit Microcontrollers DATA HANDBOOK IC20, 1995

[34] Philips Semiconductors, Application Notes and Development Tools for 80C51 Microcontrollers DATA HANDBOOK, 1995

[35] Pleinevaux, P., "An improved hard real-time scheduling for the IEEE 802.5", Real-Time Systems 4(2) pp. 99 - 112. *Real-Time Systems* (June 1992).

[36] Ramamritham, K. 1987. Channel characteristics in local-area hard real-time systems. *Computer Networks and ISDN Systems*, 13(1):3-13

[37] Sha, I., Lehoczky, J.P., Rajkumar, R., Priority Inheritance Protocols; An Approach to Real-Time Synchronization, *IEEE Trans on Computers,* 39(9), pp.1175-1185(Sept. 1990)

[38] Shin, K. G., and C. J. Hou. 1990. Analysis of three contention protocols in distributed real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium,* pp. 136-145

[39] Strosnider, J. K., Marchok, T., and Lehoczky, J., "Advanced Real-Time Scheduling Using the IEEE 802.5 Token Ring," Proceedings of the 9th IEEE Real-Time Systems Symposium, pp. 42 - 52 (December 1988).

[40] Strosnider, J.K., and T.E. Marchok. 1989. Responsive, deterministic IEEE 802.5 token ring scheduling. *Journal of Real-Time Systems*, 1(2):133-158

[41] Tindell, K., Burns, A., Wellings, A., "An Extendible Approach for Analysing Fixed Priority Hard Real-Time Tasks," Real-Time Systems 6(2), pp. 133-151 (March 1994).

[42] 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

[43] Zhao, W., J. A. Stankovic, and K. Ramamritham. 1990. A window protocol for transmission of time constrained messages. *IEEE Transactions on Computers*, 39(9):1186-1203.

[44] Zhao, W., and K. Ramamritham. 1987. Virtual time CSMA protocols for hard real-time communications. *IEEE Transactions on Software Engineering*, SE-13(8):938-952.

[45] ---------, The design of real-time programming systems based on process models, in *Proc. IEEE Real-Time Systems Symp.*, Dec. 1984, pp.5-17

VITA

Weiqing Li

Candidate for the Degree of

Master of Science

Thesis: HARD REAL-TIME COMMUNICATIONS IN CONTROLLER AREA NETWORK

Major Field: Computer Science

Biographical:

Personal Data: Born in Jinan, Shandong province, China, October 3, 1959, the son of Nailin Li and Yanmei Yu.

Education: Graduated from Jinan No. 2 High School, Jinan, Shandong province, China, in May 1976; received Bachelor of Engineering Degree in Electrical Engineering from Shandong Engineering Institute in July, 1982; received Master of Engineering Degree in Electrical & Computer Engineering from Shandong Polytecnic University in July, 1988; Completed requirements for the Master of Science Degree at Oklahoma State University in December, 1995.

Professional Experience: Teaching Assistant, Department of Computer Science, Oklahoma State University, August, 1992 to December, 1995. Lecturer, Department of Electrical & Mechanical Engineering, Shandong Light Industrial University, Jinan, China, September 1982 to July 1991.