Understanding and using the Controller Area Network

Marco Di Natale

October 30, 2008

Contents

1 Introduction **7**

2 The CAN 2.0b Standard **9**
2.1 Physical layer 10
2.1.1 Bit timing 10
2.1.2 The physical layer in ISO and SAE standards 13
2.1.3 Network topology and Bus length 15
2.1.4 Physical encoding of dominant and recessive states 18
2.2 Message frame formats 19
2.2.1 Data frame 20
2.2.2 Remote frame 22
2.2.3 Error frame 22
2.2.4 Overload frame 22
2.3 Bus arbitration 22
2.4 Message reception and filtering 23
2.5 Error management 24
2.5.1 CRC checks 24
2.5.2 Acknowledgement 25
2.5.3 Error types 25
2.5.4 Error signalling 25
2.5.5 Fault confinement 26

3 Time analysis of CAN messages **29**
3.1 Ideal behavior and worst case response time analysis 30
3.1.1 Notation 30
3.1.2 Message buffering inside the peripheral 34
3.1.3 An ideal implementation 35
3.2 Stochastic analysis 43
3.3 Probabilistic analysis 43

Chapter 1

Introduction

This book is the result of several years of study and practical experience in the design and analysis of communication systems based on the Controller Area Network (CAN) standard. CAN is a multicast-based communication protocol characterized by the deterministic resolution of the contention, low cost and simple implementation. The Controller Area Network (CAN) [4] was developed in the mid 1980s by Bosch GmbH, to provide a cost-effective communications bus for automotive applications, but is today widely used also in factory and plant controls, in robotics, medical devices, and also in some avionics systems.

CAN is a broadcast digital bus designed to operate at speeds from 20kb/s to 1Mb/s, standardized as ISO/DIS 11898 [1] for high speed applications (500 kbit/s) and ISO 11519-2 [2] for lower speed applications (125Kbit/s). The transmission rate depends on the bus length and transceiver speed. CAN is an attractive solution for embedded control systems because of its low cost, light protocol management, the deterministic resolution of the contention, and the built-in features for error detection and retransmission. Controllers supporting the CAN communication standard are today widely available as well as sensors and actuators that are manufactured for communicating data over CAN. CAN networks are today successfully replacing point-to-point connections in many application domains, including automotive, avionics, plant and factory control, elevator controls, medical devices and possibly more.

Commercial and open source implementation of CAN drivers and middleware software are today available from several sources, and support for CAN is included in automotive standards, including OSEKCom and AUTOSAR. The standard has been developed with the objective of time determinism and support for reliable communication. With respect to these properties, it has been widely studied by academia and industry and methods and tools have been developed for predicting the time and reliability characteristics of messages.

This book attempts at providing an encompassing view on the study and use of the CAN bus, with references to theory and analysis methods, but also a description of the issues in the practical implementation of the communication stack for CAN and the implications of design choices at all levels, from the selection of the controller, to the SW developer and the architecture designer. We believe such an approach may be of

advantage to those interested in the use of CAN, from students of embedded system courses, to researchers, architecture designers, system developers and all practitioners that are interested in the deployment and use of a CAN network and its nodes.

As such, the book attempts at covering all aspects of the design and analysis of a CAN communication system. The second chapter contains a short summary of the standard, with emphasis on the bus access protocol and on the protocol features that are related or affect the reliability of the communication. The third chapter focuses on the time analysis of the message response times or latencies. The fourth chapter addresses reliability issues. The fifth chapter deals with the analysis of message traces. The sixth chapter contains a summary of the main transport level and application-level protocols that are based on CAN.

Chapter 2

The CAN 2.0b Standard

This chapter introduces the version 2.0b of the CAN Standard, as described in the official Bosch specification document [4] with the main protocol features. Although the presentation provides enough details, the reader may want to check the freely available official specification document from the web for a complete description, together with the other standard sources referenced throughout this chapter.

The CAN network protocol has been defined to provide deterministic communication in complex distributed systems with the following features/capabilities:

- Possibility of assigning priority to messages and guaranteed maximum latency times.
- Multicast communication with bit-oriented synchronization.
- System wide data consistency.
- Multimaster access to the bus.
- Error detection and signalling with automatic retransmission of corrupted messages.
- Detection of possible permanent failures of nodes and automatic switching off of defective nodes.

If seen in the context of the ISO/OSI reference model, the CAN specification, originally developed by Robert Bosch Gmbh covers only the *Physical* and *Data link* layers. Later, ISO provided its own specification of the CAN protocol, with additional details on the implementation of the physical layer.

The purpose of the Physical Layer is in general to define how bits are encoded into (electrical or electromagnetic) signals with defined physical characteristics, to be transmitted over wired or wireless links from one node to another. In the Bosch CAN standard, however, the description is limited to the definition of the bit timing, bit encoding, and synchronization, which leaves out the specification of the physical transmission medium, the acceptable (current/voltage) signal levels, the connectors and other characteristics that are necessary for the definition of the driver/receiver stages and the

physical wiring. Other reference documents and implementations have filled this gap, providing solutions for the practical implementation of the protocol.

The Data-link layer consists of the Logical Link Control (LLC) and Medium Access Control (MAC) sublayers. The LLC sublayer provides all the services for the transmission of a stream of bits from a source to a destination. In particular, it defines

- services for data transfer and for remote data request,
- conditions upon which received messages should be accepted, including message filtering.
- mechanisms for recovery management and flow management (overload notification).

The MAC sublayer is probably the kernel of the CAN protocol specification. The MAC sublayer is responsible for message framing, arbitration of the communication medium, acknowledgment management, error detection and signalling. For the purpose of fault containment and additional reliability, in CAN, the MAC operations are supervised by a controller entity monitoring the error status and limiting the operations of a node if a possible permanent failure is detected.

The following sections provide more detail into each sublayer, including requirements and operations.

2.1 Physical layer

As stated in the introduction, the Bosch CAN standard defines bit encoding, timing and synchronization, which go under the Physical Signaling (PS) portion of the ISO-OSI physical layer. The standard does not cover other issues related to the physical layer, including the types of cables and connectors that can be used for communication over a CAN network, and the ranges of voltages and currents that are considered as acceptable as output and input. In OSI terminology, The Physical Medium Attachment (PMA) and Medium Dependent Interface (MDI) are the two parts of the physical layer which are not defined by the original standard.

2.1.1 Bit timing

The signal type is digital with Non Return to Zero (NRZ) bit encoding. The use of NRZ encoding ensures a minimum number of transitions and high resilience to external disturbance. The two bits are encoded in medium states defined as "‘recessive"’ and "‘dominant"’. (0 is typically assumed as associated to the "‘dominant"’ state). the protocol allows multimaster access to the bus with deterministic collision resolution. at the very lowest level, this means that if multiple masters try to drive the bus state, the "‘dominant"’ configuration also prevails upon the "‘recessive"’.

Nodes are requested to be synchronized on the bit edges so that every node agrees on the value of the bit currently transmitted on the bus. To do so, each node implements a synchronization protocol that keeps the receiver bit rate aligned with the actual rate

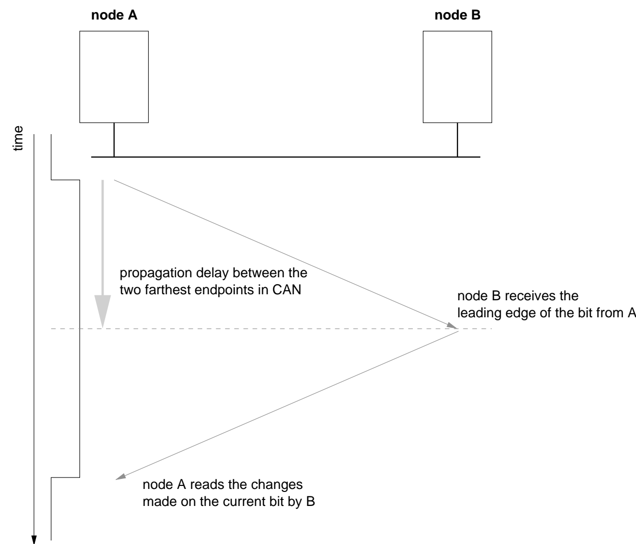

Figure 2.1: Bit propagation delay.

of the transmitted bits. The synchronization protocol uses transition edges to resynchronize nodes. Hence, long sequences without bit transitions should be avoided to avoid drifts in the node bit clocks. This is the reason why the protocol employs the so-called "‘bit stuffing"’ or "‘bit padding"’ technique, which forces a complemented bit in the stream after 5 bits of the same type have been transmitted. Stuffing bits are automatically inserted by the transmission node and removed at the receiving side before processing the frame contents.

Synchronous bit transmission enables the CAN arbitration protocol and simplifies data-flow management, but also requires a sophisticated synchronization protocol. Bit synchronization is performed first upon the reception of the start bit available with each asynchronous transmission. Later, to enable the receiver(s) to correctly read the message content, continuous resynchronization is required. Other features of the protocol influence the definition of the bit timing. For the purpose of bus arbitration, message acknowledgement and error signalling, the protocol requires that nodes can change the status of a transmitted bit from recessive to dominant, with all the other nodes in the network being informed of the change in the bit status before the bit transmission ends. This means that the bit time must be at least large enough to accomodate the signal propagation from any sender to any receiver and back to the sender.

The bit time includes a propagation delay segment that takes into account the signal propagation on the bus as well as signal delays caused by transmitting and receiving nodes. In practice, this means that the signal propagation is determined by the two nodes within the system that are farthest apart from each other (Figure 2.1).

The leading bit edge from the transmitting node (node A in the figure) reaches nodes B after the signal propagates all the way from the two nodes. At this point, B can change its value from recessive to dominant, but the new value will not reach A

until the signal propagates all the way back. Only then can the first node decide whether its own signal level (recessive in this case) is the actual level on the bus or whether it has been replaced by the dominant level by another node.

Considering the synchronization protocol and the need that all nodes agree on the bit value, the nominal bit time (reciprocal of the bit rate or bus speed) can be defined as composed of four segments (Figure 2.2)

- **Synchronization segment (SYNC_SEG)** This is a reference interval, used for synchronization purposes. The leading edge of a bit is expected to lie within this segment

- **Propagation segment (PROP_SEG)** This part of the bit time is used to compensate for the (physical) propagation delays within the network. It is twice the sum of the signals propagation time on the bus line, the input comparator delay, and the output driver delay.

- **Phase segments (PHASE_SEG1 and PHASE_SEG2)** These phase segments are time buffers used to compensate for phase errors in the position of the bit edge. These segments can be lengthened or shortened to resynchronize the position of SYNCH_SEG with respect to the following bit edge.

- **Sample point (SAMPLE_POINT)** The sample point is the point of time at which the bus level is read and interpreted as the value of that respective bit. The quantity INFORMATION PROCESSING TIME is defined as the time required to convert the electrical state of the bus, as read at the SAMPLE_POINT in the corresponding bit value.

All bit segments are multiple of the TIME QUANTUM, a time unit derived from the local oscillator. This is typically obtained by a prescaler applied to a clock with rate MINIMUM TIME QUANTUM as

TIME QUANTUM = m * MINIMUM TIME QUANTUM

with m the value of the prescaler. The TIME QUANTUM is the minimum resolution in the definition of the bit time and the maximum error assumed for the bit-oriented synchronization protocol. The segments are defined as, respectively, SYNC_SEG equal to 1 TIME QUANTUM. PROP_SEG and PHASE SEG are between 1 and 8 TIME QUANTUM. PHASE_SEG2 is the maximum between PHASE_SEG1 and the INFORMATION PROCESSING TIME, which must always be less than or equal to 2 TIME QUANTA.

This is how the synchronization protocol works. Two types of synchronization are defined: hard synchronization, and resynchronization.

- **Hard synchronization** takes place at the beginning of the frame, when the start of frame bit (see frame definition in the following section) changes the state of the bus from recessive to dominant. Upon detection of the corresponding edge, the bit time is restarted at the end of the sync segment. Therefore the edge of the start bit lies within the sync segment of the restarted bit time.

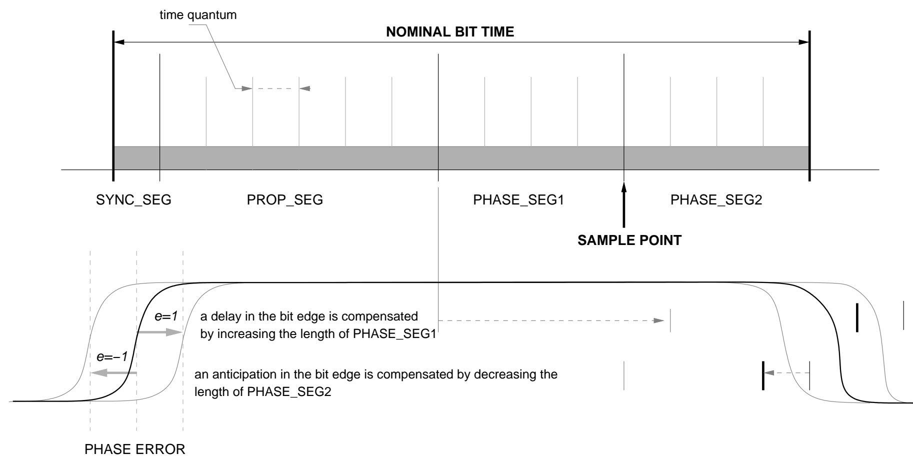

Figure 2.2: Definition of the bit time and synchronization.

- **Synchronization** takes place during transmission. The phase segments are shortened or lengthened so that the following bit starts within the SYNCH_SEG portion of the following bit time. In detail, PHASE_SEG1 may be lengthened or PHASE_SEG2 may be shortened. Damping is applied to the synchronization protocol. The amount of lengthening or shortening of the PHASE BUFFER SEGMENTs has an upper bound given by a programmable parameter RESYNCHRONIZATION JUMP WIDTH (between 1 and min(4, PHASE_SEG1) TIME QUANTA).

Synchronization information may be only be derived from transitions from one bit value to the other. Therefore, the possibility of resynchronizing a bus unit to the bit stream during a frame depends on the property that a maximum interval of time exists between any two bit transitions (enforced by the bit stuffing protocol).

The device designer may program the bit-timing parameters in the CAN controller by means of the appropriate registers. Please note that depending on the size of the propagation delay segment the maximum possible bus length at a specific data rate (or the maximum possible data rate at a specific bus length) can be determined.

2.1.2 The physical layer in ISO and SAE standards

The requirement that all nodes synchronize at the bit level whenever a transmission takes place, the arbitration mechanism, and the need that all nodes agree on the logical value encoded in the electrical status of the bus (including the ability of representing "‘dominant"’ and "‘recessive"’ bits) results in implementation constraints for the physical layer. In principle, the system designer can choose any driver/receiver and transport medium as long as the PS requirements are met, including electrical and optical media, but also powerline and wireless transmission. In practice, the physical layer has been specified for specific class of users or applications by standardization bodies

or (industrial) user groups.

ISO included CAN in its specifications as ISO 11898, with three parts: ISO 11898-1, ISO 11898-2 (high speed CAN) and ISO 11898-3 (low-speed or fault-tolerant CAN). ISO standards include the PMA and MDA parts of the physical layer. The most common type of physical signalling is the one defined by the CAN ISO 11898-2 standard, a two-wire balanced signaling scheme. ISO 11898-3 defines another two-wire balanced signaling scheme for lower bus speeds. It is fault tolerant, so the signaling can continue even if one bus wire is cut or shorted. In addition, SAE J2411 (SAE is the Society of Automotive Engineers) defines a single-wire (plus ground, of course) physical layer.

ISO 11898-2

ISO 11898-2 is the most used physical layer standard for CAN networks. The data rate is defined up to 1 Mbit/s with a required bus length of 40 m at 1 Mbit/s. The high-speed standard specifies a two-wire differential bus whereby the number of nodes is limited by the electrical busload. The two wires are identified as CAN_H and CAN_L. The characteristic line impedance is 120 Ω, the common mode voltage ranges from -2 V on CAN_L to +7 V on CAN_H. The nominal propagation delay of the two-wire bus line is specified at 5 ns/m. For automotive applications the SAE published the SAE J2284 specification. For industrial and other non-automotive applications the system designer may use the CiA 102 recommendation. This specification defines the bit-timing for rates of 10 kbit/s to 1 Mbit/s. It also provides recommendations for bus lines and for connectors and pin assignment.

ISO 11898-3

This standard is mainly used for body electronics in the automotive industry. Since for this specification a short network was assumed, the problem of signal reflection is not as important as for long bus lines. This makes the use of an open bus line possible. This means low bus drivers can be used for networks with very low power consumption and the bus topology is no longer limited to a linear structure. It is possible to transmit data asymmetrically over just one bus line in case of an electrical failure of one of the bus lines. ISO 11898-3 defines data rates up to 125 kbit/s with the maximum bus length depending on the data rate used and the busload. Up to 32 nodes per network are specified. The common mode voltage ranges between -2 V and +7 V. The power supply is defined at 5 V. The fault-tolerant transceivers support the complete error management including the detection of bus errors and automatic switching to asymmetrical signal transmission.

SAE J2411 single wire

The single-wire standard SAE J2411 is also for CAN network applications with low requirements regarding bit rate and bus length. The communication takes place via just one bus line with a nominal data rate of 33,3 kbit/s (83,3 kbit/s in high-speed mode for diagnostics). The standard defines up to 32 nodes per network. The main application area of this standard is in comfort electronics networks in motor vehicles.

An unshielded single wire is defined as the bus medium. A linear bus topology structure is not necessary. The standard includes selective node sleep capability, which allows regular communication to take place among several nodes while others are left in a sleep state.

ISO 11992 point-to-point

An additional approach to using CAN low-speed networks with fault-tolerant functionality is specified in the ISO 11992 standard. It defines a point-to-point connection for use in e.g. towing vehicles and their trailers, possibly extended to daisy-chain connections. The nominal data rate is 125 kbit/s with a maximum bus line length of 40 m. The standard defines the bus error management and the supply voltage (12 V or 24 V). An unshielded twisted pair of wires is defined as the bus medium.

Others

Not standardized are fiber-optical transmissions of CAN signals. With optical media the recessive level is represented by ”dark” and the dominant level by ”light”. Due to the directed coupling into the optical media, the transmitting and receiving lines must be provided separately. Also, each receiving line must be externally coupled with each transmitting line in order to ensure bit monitoring. A star coupler can implement this. The use of a passive star coupler is possible with a small number of nodes, thus this kind of network is limited in size. The extension of a CAN network with optical media is limited by the light power, the power attenuation along the line and the star coupler rather than the signal propagation as in electrical lines. Advantages of optical media are emission- and immission-free transmission and complete galvanic decoupling. The electrically neutral behavior is important for applications in explosive or electromagnetically disturbed environments.

2.1.3 Network topology and Bus length

The interface between a CAN controller chip and a two-wire differential bus typically consists of a transmitting and a receiving amplifier (transceiver = transmit and receive). The transceiver must convert the electrical representation of a bit from the one in use by the controller to the one defined for the bus. In addition, it must provide sufficient output current to drive the bus electrical state, and protect the controller chip against overloading. As a receiver, the CAN transceiver provides the recessive signal level and protects the controller chip input comparator against excessive voltages on the bus lines. Furthermore, it detects bus errors such as line breakage, short circuits, shorts to ground, etc. A further function of the transceiver can also be the galvanic isolation between a CAN node and the bus line.

Because of the definition of the bit time, it clearly exists a dependency between the bit time and the signal propagation delay, that is, between the maximum achievable bit rate (or transmission speed) and the length of the bus. The signal propagation delay to be considered for the computation of the maximum allowed bus length includes several stages, with variable delays, depending on the quality of the selected components:

CAN controller (50 ns to 62 ns), optocoupler (40 ns to 140 ns), transceiver (120 ns to 250 ns), and cable (about 5 ns/m).

For short, high speed networks, the biggest limitation to bus length is the transceivers propagation delay. The parameters of the electrical medium become important when the bus length is increased. Signal propagation, the line resistance and wire cross sections are factors when dimensioning a network. In order to achieve the highest possible bit rate at a given length, a high signal speed is required. Figure 2.3 plots the correspondence between bus length and bit rate when the delays of the controller, the optocoupler and the transceiver add to 250 ns, the cable delay is 5ns/m and the bit time is divided in, respectively, 21, 17 or 13 TIME QUANTA, of which 8 (the maximum allowed) represent the propagation delay.

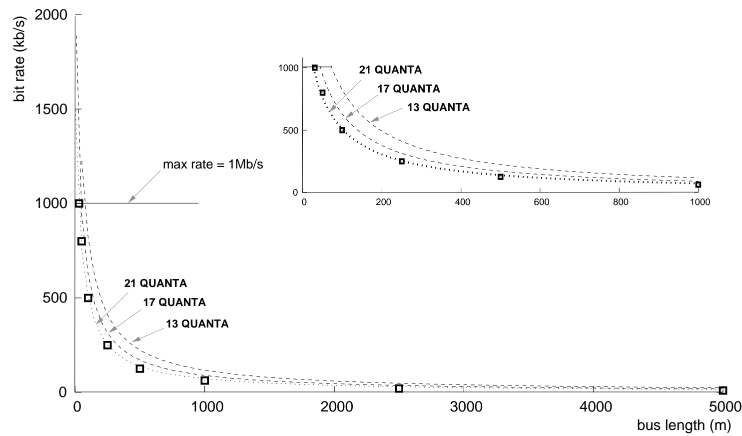

Figure 2.3: Relationship between bus length and bit rate for some possible configurations.

Some reference (not mandatory) value pairs for bus length and transmission speed are shown in Table 2.1 ([**?**]) and represented in figure 2.3 as plots. The CANopen consortium has a similar table of correspondencies (see Chapter **??**).

Bit rate	**bit time**	**bus length**
1 Mb/s	1 μs	25m
800 kb/s	1.25 μs	50m
500 kb/s	2 μs	100m
250 kb/s	4 μs	250m
125 kb/s	8 μs	500m
62.5 kb/s	16 μs	1000m
20 kb/s	50 μs	2500m
10 kb/s	100 μs	5000m

Table 2.1: Typical transmission speeds and corresponding bus lengths (CANopen)

In addition, for long bus lines the voltage drops over the length of the bus line. The wire cross section necessary is calculated by the permissible voltage drop of the signal level between the two nodes farthest apart in the system and the overall input resistance of all connected receivers. The permissible voltage drop must be such that the signal level can be reliably interpreted at any receiving node. The consideration of electromagnetic compatibility and choice of cables and connectors belongs also to the tasks of a system integrator. Table 2.2 shows possible cable sections and types for selected network configurations.

Bus speed	Cable type	Cable resistance/m	Terminator	Bus Length
50 kb/s at 1000 m	0.75 ...0.8 mm^2 (AWG18)	70 mΩ	150 ... 300 Ω	600 .. 1000 m
100 kb/s at 500 m	0.5 ... 0.6 mm^2 (AWG20)	$<$ 60 mΩ	150 ... 300 Ω	300 ... 600 m
500 kb/s at 100 m	0.34 ...0.6 mm^2 (AWG22, AWG20)	$<$ 40 mΩ	127 Ω	40 ... 300 m
1000 kb/s at 40 m	0.25 ...0.34 mm^2 (AWG23, AWG22)	$<$ 26 mΩ	124 Ω	0 ... 40 m

Table 2.2: Bus cable characteristics

Bus termination

Electrical signals on the bus are reflected at the ends of the electrical line unless measures are taken. For the node to read the bus level correctly it is important that signal reflections are avoided. This is done by terminating the bus line with a termination resistor at both ends of the bus and by avoiding unnecessarily long stubs lines of the bus. The highest possible product of transmission rate and bus length line is achieved by keeping as close as possible to a single line structure and by terminating both ends of the line. Specific recommendations for this can be found in the according standards (i.e. ISO 11898-2 and -3). The method of terminating your CAN hardware varies depending on the physical layer of your hardware: High-Speed, Low-Speed, Single-Wire, or Software-Selectable. For High-Speed CAN, both ends of the pair of signal wires (CAN_H and CAN_L) must be terminated. The termination resistors on a cable should match the nominal impedance of the cable. ISO 11898 requires a cable with a nominal impedance of 120 ohms, and therefore 120 ohm resistors should be used for termination. If multiple devices are placed along the cable, only the devices on the ends of the cable need termination resistors. Figure 2.4 gives an example of how to terminate a high-speed network.

For Low-Speed CAN, each device on the network needs a termination resistor for each data line: R(RTH) for CAN_H and R(RTL) for CAN_L. Unlike the High-Speed CAN, Low-Speed CAN requires termination on the transceiver rather than on the cable. Figure 3 indicates where the termination resistors should be placed on a single-wire, low-speed CAN network (The National Instruments Single-Wire CAN hardware includes a built-in 9.09 kohm load resistor.)

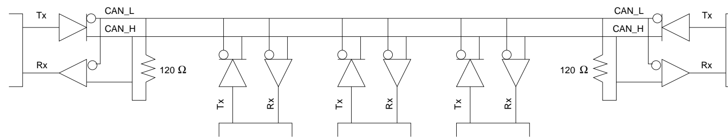

Figure 2.4: Terminating a High-Speed Network.

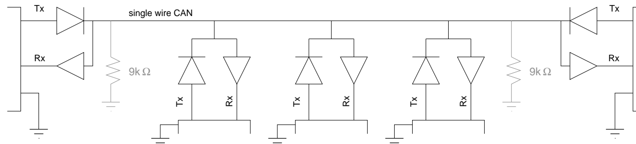

Figure 2.5: Placement of Termination Resistors on a Low-Speed Network.

It is possible to overcome the limitations of the basic line topology by using repeaters, bridges or gateways. A repeater transfers an electrical signal from one physical bus segment to another segment. The signal is only refreshed and the repeater can be regarded as a passive component comparable to a cable. The repeater divides a bus into two physically independent segments. This causes an additional signal propagation time. However, it is logically just one bus system. A bridge connects two logically separated networks on the data link layer (OSI layer 2). This is so that the CAN identifiers are unique in each of the two bus systems. Bridges implement a storage function and can forward messages or parts thereof in an independent time-delayed transmission. Bridges differ from repeaters since they forward messages, which are not local, while repeaters forward all electrical signals including the CAN identifier. A gateway provides the connection of networks with different higher-layer protocols. It therefore performs the translation of protocol data between two communication systems. This translation takes place on the application layer (OSI layer 7).

2.1.4 Physical encoding of dominant and recessive states

CAN specifies two logical states: recessive and dominant. According to ISO-11898-2, a differential voltage is used to represent recessive and dominant states (or bits), as shown in Figure 2.6.

In the recessive state (usually logic 1), the differential voltage on CAN_H and CAN_L is less than the minimum threshold (<0.5V receiver input or <1.5V transmitter output). In the dominant state (logic 0), the differential voltage on CAN_H and CAN_L is greater than the minimum threshold. If at least a node outputs a dominant bit, the status of the bus changes to "‘dominant"’ regardless of other recessive bit outputs. This is the foundation of the nondestructive bitwise arbitration of CAN.

The ISO11898-2 specification requires that a compliant or compatible transceiver must meet a number of electrical specifications. Some of these specifications are intended to ensure the transceiver can survive harsh electrical conditions, thereby protecting the communications of the CAN node. The transceiver must survive short circuits

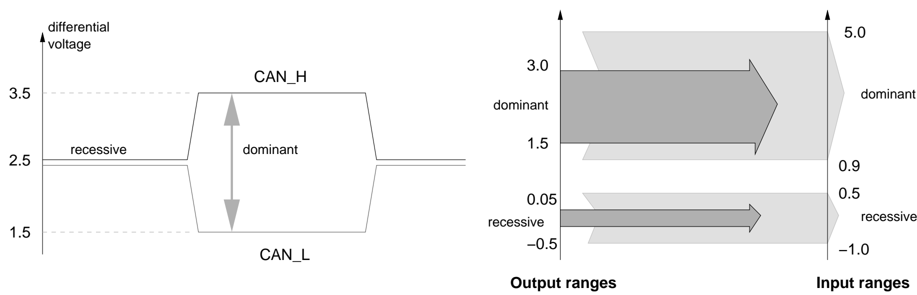

Figure 2.6: Encoding of dominant and recessive bits.

on the CAN bus inputs from -3V to +32V and transient voltages from -150V to +100V. Table 2.3 shows the major ISO11898-2 electrical requirements,

Parameter	min	max	unit
DC Voltage on CANH and CANL	-3	+32	V
Transient voltage on CANH and CANL	-150	+100	V
Common Mode Bus Voltage	-2.0	+7.0	V
Recessive Output Bus Voltage	+2.0	+3.0	V
Recessive Differential Output Voltage	-500	+50	mV
Differential Internal Resistance	10	100	kΩ
Common Mode Input Resistance	5.0	50	kΩ
Differential Dominant Output Voltage	+1.5	+3.0	V
Dominant Output Voltage (CANH)	+2.75	+4.50	V
Dominant Output Voltage (CANL)	+0.50	+2.25	V
Output Current		100	mA

Table 2.3: Acceptable voltage ranges for CAN transmitters and receivers

2.2 Message frame formats

In CAN, there are 4 different types of frames, according to their content and function.

- **DATA FRAME**s contain data information from a source to possibly multiple receivers.
- **REMOTE FRAME**s are used to request transmission of a corresponding (with the same identifier) DATA FRAME.
- **ERROR FRAME**s are transmitted whenever a node on the network detects an error.
- **OVERLOAD FRAME**s are used fro flow control, to request an additional time delay before the transmission of a DATA or REMOTE frame.

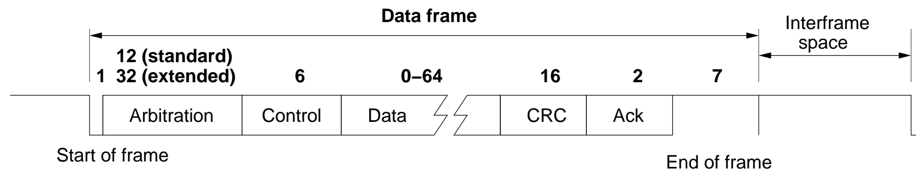

Figure 2.7: The CAN data frame format.

2.2.1 Data frame

DATA frames are used to transmit information between a source node and one of more receivers. CAN frames do not use explicit addressing for identifying the message receivers, but each node defines the messages that will be received based on their information content, which is encoded in the IDENTIFIER field of the frame. There are two different formats of CAN messages, according to the type of message identifier that is used by the protocol. Standard FRAMES are frames defined with an IDENTIFIER FIELD of 11 bits. Extended frames have been later defined (from version 2.0 of the protocol) as frames with an IDENTIFIER of 29 bit. Standard and extended frames can be transmitted on the same bus by different nodes or by the same node. The arbitration part of the protocol has means to arbitrate between frames of different identifier type.

The CAN data frame has the format of Figure 2.7, where the size of the fields is expressed in bits. Each frame starts with a single dominant bit, interrupting the recessive state of the idle bus. Following, the identifier field defines both the priority of the message for arbitration (Section yy) and the data content (identification) of the message stream. The other fields are: the control field containing information on the type of message; the data field containing the actual data to be transmitted, up to a maximum of 8 bytes; the checksum used to check the correctness of the message bits; the acknowledge (ACK) used to acknowledge the reception; the ending delimiter (ED) used to signal the end of the message and the idle space (IS) or interframe bits (IF) used to separate one frame from the following.

Identifier field

The CAN protocol requires that all contending messages have a unique identifier. The identifier field consists of 11 (+1) bits in standard format and 29 (+3) bits in extended format, following the scheme of Figure 2.8. In both cases, the field starts with the 11 bits (the most significant bits, in the extended format) of the identifier, followed by the RTR (Remote Transmission Request) bit in the standard format and by the SRR (Substitute Remote Request) in the extended format. The RTR bit distinguishes data frames from remote request frames. It is dominant for data frames, recessive for remote frames. The SRR is only a placeholder (always recessive) for guaranteeing the deterministic resolution of the arbitration between standard and extended frames.

The extended frame continues with a single IDE bit (IDentifier Extension, always recessive), then the remaining 18 least significant identifier bits and finally, the RTR bit. The IDE bit is part of the control field in standard frames (where it is always dominant).

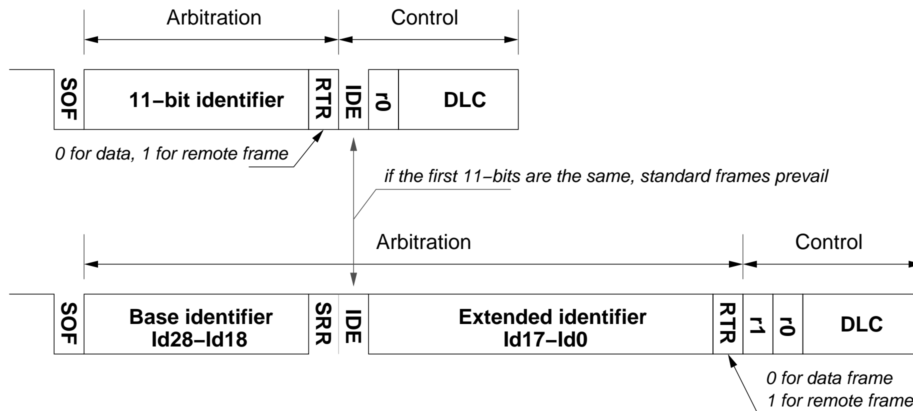

Figure 2.8: The CAN identifier format.

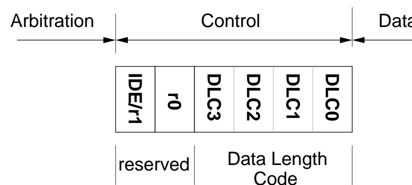

Figure 2.9: The control field format.

Control field

The control field contains 6 bits. the first two bits are reserved or predefined in content. In the standard message format the first bit is the IDE (Identifier Extension Bit), followed by a reserved bit. The IDE bit is dominant in standard formats and recessive in extended formats and ensures the deterministic resolution of the contention (in favor of standard frames) when the first eleven identifier bits of two messages (one standard, one extended) are the same. In the extended format there are two reserved bits. For these reserved bits, the standard specifies that they are to be sent as recessive, but receivers will accept any value (dominant or recessive). The following four bits define the length of the data content (Data Length Content, or DLC) in bytes. If the dominant bit is interpreted as 1 (contrary to the common notation in which it is read as 0) and the recessive as 0, the four DLC bits are the unsigned binary coding of the length.

CRC e acknowledgement fields

The ACK FIELD consists of two bits. One has the function of recording acknowledgements from receivers (ACK SLOT). The other is simply a delimiter (one bit of bus recessive state). The acknowledgement is recorded in the ACK SLOT by simply letting the receivers othat have validated the received message verwrite the recessive bit sent by the transmitter by a dominant bit.

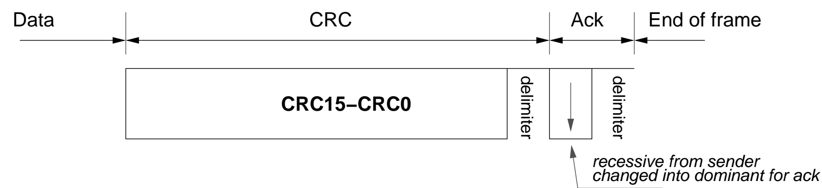

Figure 2.10: The CRC and Acknowledgement formats.

Interframe space

Data frames and remote frames are separated by an INTERFRAME space (7 recessive bits) on the bus.

The frame segments START OF FRAME, ARBITRATION FIELD, CONTROL FIELD, DATA FIELD and CRC SEQUENCE are subject to bit stuffing. The remaining bit fields of the DATA FRAME or REMOTE FRAME (CRC DELIMITER, ACK FIELD, and END OF FRAME) are of fixed form and not stuffed.

2.2.2 Remote frame

A remote frame is used to request the transmission of a message with a given identifier from a remote node. A remote frame has the same format of a data frame with the following characteristics:

- the *identifier* field is used to indicate the identifier of the requested message
- the data field is always empty (0 bytes)
- the DLC field indicates the data length of the requested message (not the transmitted one)
- the RTR bit in the arbitration field is always set to recessive

2.2.3 Error frame

The error frame is not a true frame, but rather the result of an error signalling and recovery. The details of such an event are described in the following sections.

2.2.4 Overload frame

The ERROR FRAME and the OVERLOAD FRAME are of fixed form as well and not coded by the method of bit stuffing.

2.3 Bus arbitration

The CAN arbitration protocol is both priority-based and *non-preemptive*, as a message that is being transmitted can not be preempted by higher priority messages that were

queued after the transmission has started. The CAN[?] bus is essentially a wired AND channel connecting all nodes. The media access protocol works by alternating contention and transmission phases. The time axis is divided into slots, which must be larger than or equal to the time it takes for the signal to travel back and forth along the channel. The contention and transmission phases take place during the digital transmission of the frame bits.

If a node wishing to transmit finds the shared medium in a idle state, it waits for the next slot and starts an arbitration phase by issuing a start- of-frame bit. At this point, each node with a message to be transmitted (e.g., the message may be placed in a peripheral register called *TXObject*) can start racing to grant access of the shared medium, by serially transmitting the identifier (priority) bits of the message in the arbitration slots, one bit for each slot starting from the most significant. Collisions among identifier bits are resolved by the logical AND semantics, and if a node reads its priority bits on the medium without any change, it realizes it is the winner of the contention and it is granted access for transmitting the rest of the message while the other nodes switch to a listening mode. In fact, if one of the bits is changed when reading it back from the medium, this means there is a higher priority (dominant bit) contending the medium and thus the message withdraws.

2.4 Message reception and filtering

Controllers have one or more registers (commonly defined as RxObjects) for the reception of CAN messages. Nodes can define one or more message *Filters* (typically one associated to each RxObject) and one or more reception *Masks* to declare the messages they are interested in receiving.

Masks can be individually associated to RxObjects, but most often to group of them (or all of them). A reception mask tells on which bits of the incoming message identifier the filters should operate to detect a possible match (Figure xx). A bit at 1 in the mask register usually enables comparison between the bits of the filter and the received id in the corresponding positions. A bit at 0 means don't care or don't match with the filter data. In the example in the figure, the id transmitted on the bus is `01110101010` (`0x3AA`). Given the mask configuration, only the first, third, sixth, seventh and eight bit are going to be considered for comparison with the reception filters.

After comparison with the filters, the filter of RxObject1 is found to be a match for the required bits, and the incoming message is then stored in the corresponding RxObject.

The point of time at which a message is taken to be valid, is different for the transmitter and the receivers of the message. The transmitter checks all bits until the end of END OF FRAME field. Receivers consider a message valid if there is no error until the last but one bit of END OF FRAME. The value of the last bit of END OF FRAME is treated as dont care, (a dominant value does not lead to a FORM ERROR). This difference in the interpretation of a valid transmission/reception of a message can give rise to inconsistent message omissions/duplicates. This problem will be explored in detail in chapter xx.

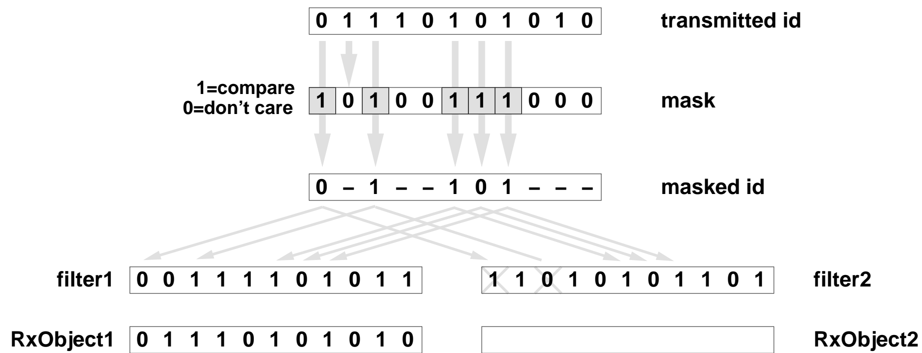

Figure 2.11: Masks and filters for message reception.

2.5 Error management

The CAN protocol is designed for safe data transfers and provides mechanisms for error detection, signalling and self-diagnostics, including measures for fault confinement, which prevent faulty nodes from affecting the status of the network.

The general measures for error detection are based on the capability of each node of monitoring broadcast transmissions over the bus, whether it is a transmitter or a receiver, and to signal error conditions resulting from several sources. Corrupted messages are flagged by any node detecting an error. Such messages are aborted and will be retransmitted automatically.

2.5.1 CRC checks

The CRC portion of the frame is obtained by selecting the input polynomial (to be divided) as the stream of bits from the START OF FRAME bit (included) to the DATA FIELD (if present) followed by 15 zeros. This polynomial is divided by the generator

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

The remainder of the division is the CRC SEQUENCE portion of the frame. Details on the computation of the CRC SEQUENCE field (including the generating algorithm) can be found in the Bosch specification. The CAN CRC has the following properties: it detects all errors with 5 or fewer modified bits, and all burst errors up to 15 bits long and all errors affecting an odd number of bits. That specification states that multi-bit errors outside this specification (6 or more disturbed bits or bursts longer than 15 bits) are undetected with a probability of $3x10^{-5}$. Unfortunately, this evaluation does not take into account the effect of bit stuffing. It is shown in [**?**] that in reality, the protocol is much more vulnerable to bit errors, and even 2-bit errors can happen undetected. More details will be provided in the Reliability chapter.

2.5.2 Acknowledgement

The protocol requires that receivers acknowledge reception of the message by changing the content of the ACK FIELD. The ACK FIELD is two bits long and contains the ACK SLOT and the ACK DELIMITER. In the ACK FIELD the transmitting station sends two recessive bits. All receiver nodes that detect a correct message (after CRC checks), inform the sender by changing the recessive bit of the ACK SLOT into a dominant bit.

2.5.3 Error types

There are 5 different error types:

- **BIT ERROR** A unit that is sending a bit on the bus also monitors the bus. A BIT ERROR has to be detected at that bit time, when the bit value that is monitored is different from the bit value that is sent. Exceptions are the recessive bits sent as part of the arbitration process or the ACK SLOT.
- **STUFF ERROR** a STUFF ERROR is detected at the 6th consecutive occurrence of the same bit in a message field that is subject to stuffing.
- **CRC ERROR** If the CRC computed by the receiver differs from the one stored in the message frame.
- **FORM ERROR** when a fixed-form bit field contains one or more illegal bits. (Note, that for a Receiver a dominant bit during the last bit of END OR FRAME is not treated as FORM ERROR, this is the possible cause of inconsistent message omission and duplicates, as further explained in the chapter on reliability).
- **ACKNOWLEDGMENT ERROR** detected by a transmitter if a recessive bit is found on the ACK SLOT.

2.5.4 Error signalling

The ERROR FRAME is not a true frame, but it is actually the result of an error signalling sequence, consisting of the superposition of ERROR FLAGs, transmitted from different nodes, possibly at different times, followed by an ERROR DELIMITER field. Whenever a BIT ERROR, a STUFF ERROR, a FORM ERROR or an ACKNOWLEDGMENT ERROR is detected by any station, transmission of an ERROR FLAG is started at the respective station at the next bit. Whenever a CRC ERROR is detected, transmission of an ERROR FLAG starts at the bit following the ACK DELIMITER, unless an ERROR FLAG for another condition has already been started. A station detecting an error condition signals this by transmitting an ERROR FLAG. For an error active node it is an ACTIVE ERROR FLAG, consisting of 6 dominant bits, for an error passive node it is a PASSIVE ERROR FLAG, consisting of six consecutive recessive bits.

The ERROR FLAGs form violates the bit stuffing rule or destroys the fixed form ACK FIELD or END OF FRAME fields. As a consequence, all other stations detect an error condition and on their part start transmission of an ERROR FLAG. So the

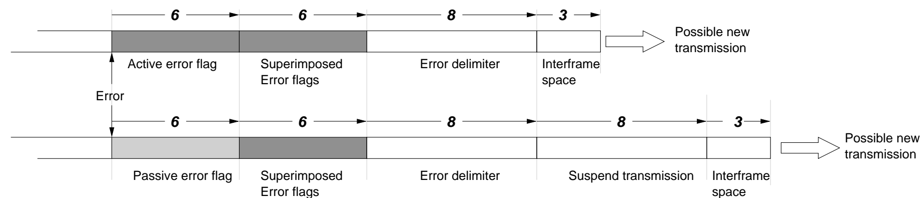

Figure 2.12: Bit sequence after detection of an error.

sequence of dominant bits which actually can be monitored on the bus results from a superposition of different ERROR FLAGs transmitted by individual stations. The total length of this sequence varies between a minimum of six and a maximum of twelve bits. The PASSIVE ERROR FLAG sent by error passive nodes has no effect on the bus. However, the signalling node will still have to wait for six consecutive bits of equal polarity, beginning at the start of the PASSIVE ERROR FLAG before continuing.

The recovery time from detecting an error until the start of the next message is at most 31 bit times, if there is no further error.

2.5.5 Fault confinement

The CAN protocol has a fault confinement protocol that detects faulty units and places them in passive or off states, so that they cannot affect the bus state with their outputs. The protocol assigns each node to one of three states:

- **Error active** units in this state are assumed to function properly. Units can transmit on the bus and signal errors with an ACTIVE ERROR FLAG.
- **Error passive** units are suspected of faulty behavior (in transmission or reception). They can still transmit on the bus, but their error signalling capability is restricted to the transmission of a PASSIVE ERROR FLAG.
- **Bus off** units are very likely corrupted and cannot have any influence on the bus. (e.g. their output drivers are switched off.)

Units change their state according to the value of two integer counters: TRANSMIT ERROR COUNT and RECEIVE ERROR COUNT, which give a measure of the likelyhood of a faulty behavior on transmission and reception, respectively. The state transitions are represented in Figure 2.13.

The transition from "'Bus off"' to "'Active error"' state is subject to the additional condition that 128 occurrence of 11 consecutive recessive bits have been monitored on the bus.

The counters are updated according to the rules of Table 2.5.5:

A special condition may happen during start-up or wake-up. If during start-up only 1 node is online, and if this node transmits some message, it will get no acknowledgment, detect an error and repeat the message. It can become error passive but not bus off due to this reason.

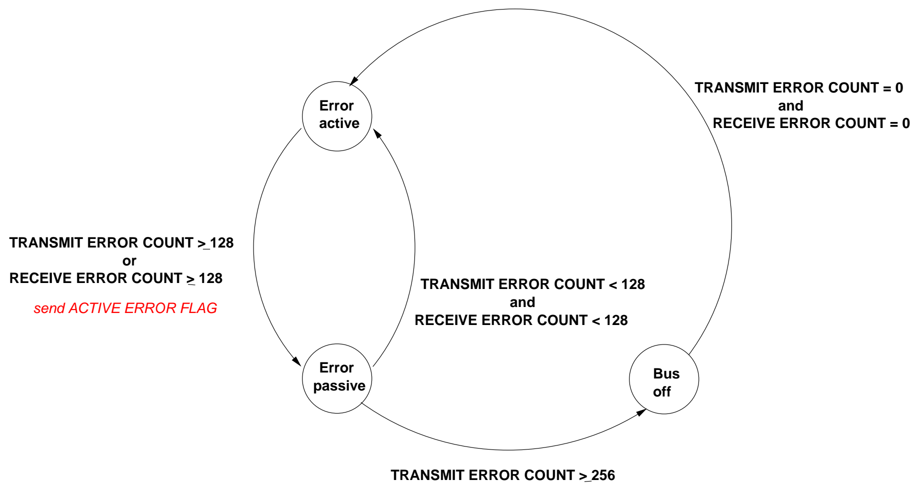

Figure 2.13: Node error states.

Node as RECEIVER	change	when node
RECEIVE ERROR COUNT	+ 1	detects an error unless a BIT ERROR during an ACTIVE ERROR FLAG or an OVERLOAD FLAG
	+ 8	detects a dominant bit as the first bit after sending an ERROR FLAG
	+ 8	detects a BIT ERROR while sending an ACTIVE ERROR FLAG or an OVERLOAD FLAG
	- 1	successful reception of a message if RECEIVE ERROR COUNT $\leq$ 127,
	any n 119 $\leq$ n $\leq$ 127	successful reception of a message if RECEIVE ERROR COUNT ¿ 127
	+ 8	node detects more than 7 consecutive dominant bits after sending an ACTIVE ERROR FLAG, PASSIVE ERROR FLAG or OVERLOAD FLAG (+8 for each additional 8 dominant bits)
Node as TRANSMITTER	change	when node
TRANSMIT ERROR COUNT	+ 8	sends an ERROR FLAG (with some exceptions, see [**?**])
	+ 8	detects a BIT ERROR while sending an ACTIVE ERROR FLAG or an OVERLOAD FLAG
	- 1	successful transmission of a message
	+ 8	detects more than 7 consecutive dominant bits after sending an ACTIVE ERROR FLAG, PASSIVE ERROR FLAG or OVERLOAD FLAG (+8 for each additional 8 dominant bits)

Table 2.4:

Chapter 3

Time analysis of CAN messages

The CAN protocol adopts a collision detection and resolution scheme, where the message to be transmitted is chosen according to its identifier. When multiple nodes need to transmit over the bus, the lowest identifier message is selected for transmission. This MAC arbitration protocol allows encoding the message priority into the identifier field and implementing priority-based real-time scheduling of periodic and aperiodic messages. Predictable scheduling of real-time messages on the CAN bus is then made possible by adapting existing real-time scheduling algorithms to the MAC arbitration protocol or by superimposing a higher-level purposely designed scheduler.

Starting from the early 90's solutions have been derived for the worst case latency evaluation. The analysis method, commonly known in the automotive world as Tindell's analysis (from [14]) is very popular in the academic community and had a substantial impact on the development of industrial tools and automotive communication architectures.

The original paper has been cited more than 200 times, its results influenced the design of on-chip CAN controllers like the Motorola msCAN and have been included in the development of the early versions of Volcano's Network Architect tool. Volvo used these tools and the timing analysis results from Tindell's theory to evaluate communication latency in several car models, including the Volvo S80, XC90, S60, V50 and S40 [5].

The analysis was later found to be flawed, although under quite high network load conditions, that are very unlikely to occur in practical systems. Davis and Bril provided evidence of the problem as well as a set of formulas for the exact or approximate evaluation of the worst case message response times (or latencies) in [6].

The real problem with the existing analysis methods, however, are a number of assumptions on the architecture of the CAN communication stack that are seldom true in practical automotive systems. These assumptions include the existence of a perfect priority-based queue at each node for the outgoing messages, the availability of one output register for each message (or preemptability of the transmit registers) and immediate (zero-time) or very fast copy of the highest priority message from the software queue used at the driver or middleware level, to the transmit register(s) or TxObjects at the source node as soon as they are made available by the transmission of a message.

When these assumptions do not hold, as unfortunately is the case for many systems, the latency of the messages can be significantly larger then what is predicted by the analysis. A relatively limited number of studies have attempted the analysis of the effect of additional priority inversion and blocking caused by limited TxObject availability at the CAN adapter. The problem has been discussed first in [16] where two peripheral chips (Intel 82527 and Philips 82C200) are compared with respect to the availability of TxObjects. The possibility of unbounded priority inversion is shown for the single TxObject implementation with preemption and message copy times larger than the time interval that is required for the transmission of the interframe bits. The effect of multiple TxObject availability is discussed in [11] where it is shown that even when the hardware transmits the messages in the TxObjects according to the priorities (identifiers) of the messages, the availability of a second TxObject is not sufficient to bound priority inversion. It is only by adding a third TxObject, under these assumptions, that priority inversion can be avoided.

However, sources of priority inversion go well beyond the limited availability of TxObjects at the bus adapter. In this chapter we review the time analysis of the ideal system configuration and later, the possible causes of priority inversion, as related to implementation choices at all levels in the communication stack. We will detail and analyze all possible causes of priority inversion and describe their expected impact on message latencies. Examples derived from actual message traces will be provided to show typical occurrences of the presented cases.

3.1 Ideal behavior and worst case response time analysis

This section starts from the description of the original analysis method by Tindell. Although formally not correct in the general case (later, the flaw is discussed, together with the fix proposed in [6]), it is still valid when the worst-case response time (or latency) of a message is not caused by a busy period that extends beyond the message period (or interarrival time), and provides a very good introduction to the general analysis method.

3.1.1 Notation

We assume the system is composed by a set of *periodic* or *sporadic* messages with *queuing jitter*, that is, messages are enqueued at periodic time instants or two consecutive instances of the same message are enqueued with a *minimum interarrival time*. In addition, the actual queuiing instants can be delayed with respect to the reference periodic event stream by a variable amount of time, upper bounded by the *queuing jitter*.

For the purpose of schedulability analysis, a periodic or sporadic message stream M_i is characterized by the t-uple

$$M_i = \{m_i, id_i, N_i, T_i, J_i, D_i\}$$

where m_i is the length of the message in bits. CAN frames can only contain a data content that is multiple of 8 bits. Hence, m_i is the smallest multiple of 8 bits that is larger than the actual payload. id_i is the CAN identifier of the message, N_i the index of its sending node, T_i its period or minimum interarrival time, J_i (in case the message is periodic, $J_i = 0$ if the message is sporadic) its arrival jitter and D_i its deadline (relative to the release time of the message). In almost all cases of practical interest it must be $D_i < T_i$ because a new instance of a message will overwrite the data content of the old one if it has not been transmitted yet. Messages are indexed according to their identifier priority, that is, $i < j \leftrightarrow id_i < id_j$. B_r is the bit rate of the bus, and p is the number of protocol bits in a frame. In the case of a standard frame format (11-bit identifier), then $p = 46$, if an extended format (29-bit identifier) is used, then $p = 65$. In the following, we assume a standard frame format, but, the formulas can be easily adapted to the other case. The worst case message length needs to account for bit stuffing (only 34 of the 46 protocol bits are subject to stuffing) and e_i is the time required to transmit the message provided that it wins the contention on the bus

$$e_i = \frac{m_i + p + \left\lfloor \frac{m_i+34}{4} \right\rfloor}{B_r}$$

Finally, w_i is the *queuing delay*, that is, the worst case interval, from the time message M_i is enqueued (at the middleware level) to the time it starts transmission on the bus.

Please note that, in reality, messages are enqueued by tasks. In most cases, it is actually a single task, conceptually at the middleware level, that is called TxTask. If this is the case, the queuing jitter of each message can be assumed to be the worst case response time of the queuing TxTask.

In modeling the scheduling problem for CAN, one assumption is commonly used. Each time a new bus arbitration starts, the CAN controller at each node enters the highest priority message that is available. In other words, it is assumed that the peripheral TxObjects always contain the highest priority messages in the middleware queue, and the controller selects the highest priority (lowest id) among them.
If this is the case, the worst case response time of M_i is given by

$$R_i = J_i + w_i + e_i \tag{3.1}$$

The queueing delay term w_i consists of two terms:

- blocking B_i due to a lower priority message that is transmitted when M_i is enqueued
- interference I_i due to higher priority messages that are transmitted before M_i on the bus

The queuing delay w_i is part of the *busy period* of level i, defined as follows: a contiguous interval of time that starts at the time t_s when a message of priority higher than i is queued for transmission and there are no other higher priority messages waiting to be transmitted that were queued before t_s. During the busy period, only messages with priority higher than i are transmitted. It ends at the earliest time t_e when the bus becomes idle or a message with priority lower than i is transmitted.

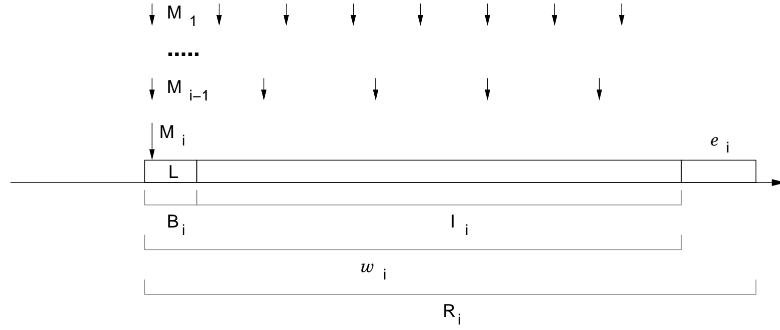

Figure 3.1: Worst case response time, busy period and time critical instant for M_i.

The worst-case queuing delay for message M_i occurs for some instance of M_i queued within a level-i busy period that starts immediately after the longest lower priority frame is transmitted on the bus. The busy period that corresponds to the worst case response time must occur at the critical instant for M_i [10], that is, when M_i is queued at the same time together with all the other higher priority messages in the network. Subsequently, these messages are enqueued with their highest possible rate (if sporadic). The situation is represented in figure 3.1.

Formally, this requires the evaluation of the following fixed-point formula for the worst-case queuing delay:

$$w_i = B_i + \sum_{k \in hp(i)} \left\lceil \frac{w_i + J_k + \tau_{bit}}{T_k} \right\rceil e_k \tag{3.2}$$

where hp(i) is the set of messages with priorities higher than i and τ_{bit} is the time for the transmission of one bit on the network. The solution to the fixed-point formula can be computed considering that the right hand side is a monotonic non-decreasing function of w_i. The solution can be found iteratively using the following recurrence.

$$w_i^{(m+1)} = B_i + \sum_{k \in hp(i)} \left\lceil \frac{w_i^{(m)} + J_k + \tau_{bit}}{T_k} \right\rceil e_k \tag{3.3}$$

A suitable starting value is $w_i^{(0)} = B_i$, or $w_i^{(0)} = \epsilon$, with $\epsilon \ll C_j \forall j$ for the lowest priority message. The recurrence relation iterates until, either $w_i^{(m+1)} > D_i$, in which case the message is not schedulable, or $w_i^{(m+1)} = w_i^{(m)}$ in which case the worst-case response time is simply given by $w_i^{(m)}$.

The flaw in the above analysis is that, given the constraint $D_i \leq T_i$, it implicitly assumes that if M_i is schedulable, then the priority level-i busy period will end at or before T_i. Please note that this assumption fails only when the network is loaded to the point that the response time of M_i is very close to its period. Given the typical load of CAN networks (seldom beyond 65%) this is almost always not the case.

When there is such a possibility, the analysis needs to be corrected as follows. To explain the problem and the fix, we refer to the same example in [6] with three

messages (A, B and C) with very high utilization. All messages have a transmission time of 1 unit and periods of, respectively, 2.5, 3.5 and 3.5 units. The critical instant is represented in Figure 3.2. Because of the impossibility of preempting the transmission of the first instance of C, the second instance of message A is delayed, and, as a result, it pushes back the execution of the second instance of C. The result is that the worst case response time for message C does not occur for the first instance (3 time units), but for the second one, with an actual response time of 3.5 time units. The fix can be found observing that the worst case response time is always inside the busy period for message C. Hence, to find the correct worst case, the formula to be applied is a small modification of (3.4) that checks all the q instances of message M_i that fall inside the busy period starting from the critical instant. Analytically, the worst-case queuing delay for the q-th instance in the busy period is:

$$w_i(q) = B_i + qe_i + \sum_{k \in hp(i)} \left\lceil \frac{w_i + J_k + \tau_{bit}}{T_k} \right\rceil e_i \tag{3.4}$$

and its response time is

$$R_i(q) = J_i + w_i(q) - qT_i + e_i \tag{3.5}$$

where q ranges from 0 to the last instance of M_i inside the busy period, that is, until $R_i(q) < T_i$. After the end of the busy period, the worst case is simply found as

$$R_i = max_q\{R_i(q)\}$$

Alternatively, an upper bound to the worst case response time may simply be found by substituting the worst case frame transmission time in place of B_i in formula (3.4).

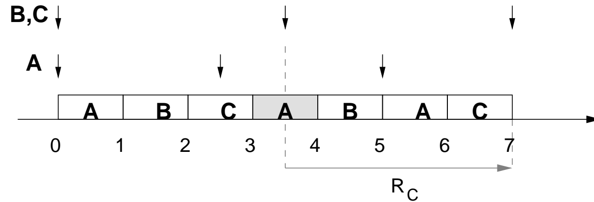

Figure 3.2: An example showing the need for a fix in the evaluation of the worst-case response time.

Please note that the previous analysis methods are based on the assumption that the highest priority active message at each node is considered for bus arbitration. This simple statement has many implications. Before getting into details, however, the definition of *priority inversion*, as opposed to *blocking*, must be provided. *Blocking* is defined as the amount of time a message M_i must wait because of the ongoing transmission of a lower priority message on the network (at the time M_i is queued). Blocking cannot be avoided and derives from the impossibility of preempting an ongoing transmission. *Priority inversion* is defined as the amount of time M_i must wait for the transmission of lower priority messages because of other causes, including queuing policies, active waits or other events. Priority inversion may occur after the message is queued.

Model	Type	Buffer Type	Priority and Abort
Microchip MCP2515	Standalone controller	2 RX - 3 TX	lowest message ID, abort signal
ATMEL AT89C51CC03 AT90CAN32/64	8 bit MCU w. CAN controller	15 TX/RX msg. objects	lowest message ID, abort signal
FUJITSU MB90385/90387 90V495	16 bit MCU w. CAN controller	8 TX/RX msg. objects	lowest TxObject num. abort signal
FUJITSU 90390	16 bit micro w. CAN controller	16 TX/RX msg. objects	lowest TxObject num. abort signal
Intel 87C196 (82527)	16 bit MCU w. CAN controller	14 TX/RX + 1 RX msg. objects	lowest TxObject num. abort possible (?)
INFINEON XC161CJ/167 (82C900)	16 bit MCU w. CAN controller	32 TX/RX msg. objects (2 buses)	lowest TxObject num., abort possible (?)
PHILIPS 8xC592 (SJA1000)	8 bit MCU w. CAN controller	one TxObject	abort signal

Table 3.1: Summary of properties for some existing CAN controllers.

3.1.2 Message buffering inside the peripheral

The configuration and management of the peripheral transmit and receive objects is of utmost importance in the evaluation of the priority inversion at the adapter and of the worst case blocking times for real-time messages. A set of CAN controllers are considered in this paper with respect to the availability of message objects, priority ordering when multiple messages are ready for transmission inside TxObjects, and possibility of aborting a transmission request and changing the content of a TxObject. The last option can be used when a higher priority message becomes ready and all the TxObjects are used by lower priority data.

There is a large number of CAN controllers available on the market. In this paper, seven controller types, from major chip manufacturers, are analyzed. The results are summarized in Table 3.1. The chips listed in the table are MCUs with integrated controllers or simple bus controllers. In case both options are available, controller product codes are shown between parenthesis. All controllers allow both polling-based and interrupt-based management of both transmission and reception of messages.

Some of those chips have a fixed number of TxObjects and RxObjects. Others give the programmer freedom in allocating the number of Objects available to the role of transmission or reception registers. When multiple messages are available in the TxObjects at the adapter, most chips select for transmission the object with the lowest identifier, *not necessarily the message with the lowest Id*. Finally, in most chips, a message that is currently inside a TxObject can be evicted from it (the object preempted or the transmission aborted), unless the transmission is actually taking place. In the following sections we will see how these decision can affect the timing predictability and become a possible source of priority inversion.

3.1.3 An ideal implementation

The requirement that the highest available message at each node is selected for the next arbitration round on the bus can be satisfied in several ways. The simplest solution is when the CAN controller has enough TxObjects to accomodate all the outgoing message streams. This situation is represented in figure 3.3. Even if the controller transmits messages in order of TxObject identifier, it is sufficient to map messages to objects in such a way that the (CAN identifier) priority order is preserved by the mapping.

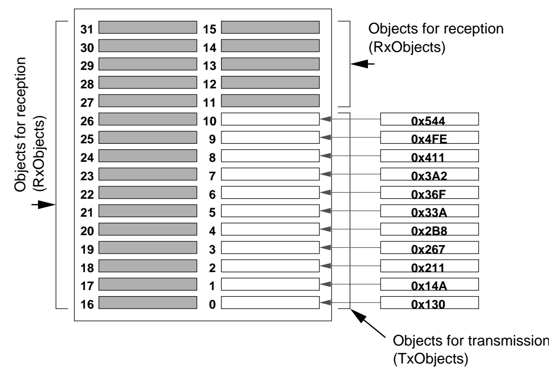

Figure 3.3: Static allocation of TxObjects.

This solution is possible in some cases. In [16] it is argued that, since the total number of available objects for transmission and reception of messages can be as high as 14 (for the Intel 82527) or 32 (for most TouCAN-based devices), the device driver can assign a buffer to each outgoing stream and preserve the ID order in the assignment of buffer numbers. Unfortunately, this is not always possible in current automotive architectures where message input is typically polling-based rather than interrupt-based and a relatively large number of buffers must be reserved to input streams in order to avoid message loss by overwriting. Furthermore, for some ECUs, the number of outgoing streams can be very large, such as, for example, gateway ECUs. Of course, in the development of automotive embedded solutions, the selection of the controller chip is not always an option and designers should be ready to deal with all possible HW configurations.

The other possible solution, when the number of available TxObjects is not sufficient, is to put all outgoing messages, or a subset of them, in a software queue (figure 3.4). When a TxObject is available, a message is extracted from the queue and copied into it. This is the solution used by the Vector CAN drivers. Preserving the priority oder would require that:

- The queue is sorted by message priority (message CAN identifier)
- When a TxObject becomes free, the highest priority message in the queue is immediately extracted and copied into the TxObject (interrupt-driven management of message transmissions).

- If, at any time, a new message is placed in the queue, and its priority is higher than the priority of any message in the TxObjects, then the lowest priority message holding a TxObjects needs to be evicted, placed back in the queue and the newly enqeueud message copied in its place.

- Finally, messages in the TxObjects must be sent in order of their identifiers (priorities). If not, the position of the messages in the Txobjects should be dynamically rearranged.

When any of these conditions does not hold, priority inversion occurs and **the worst case timing analysis fails**, meaning that the actual worst-case can be larger than what is predicted by Eq. (3.4). Each of these causes of priority inversion will now be analyzed in detail. However, before delving into these details, it is necessary to recall another, more subtle cause of priority inversion that may happen even when all the previous conditions are met. This problem arises because because of the necessary finite copy time between the queue and the TxObjects.

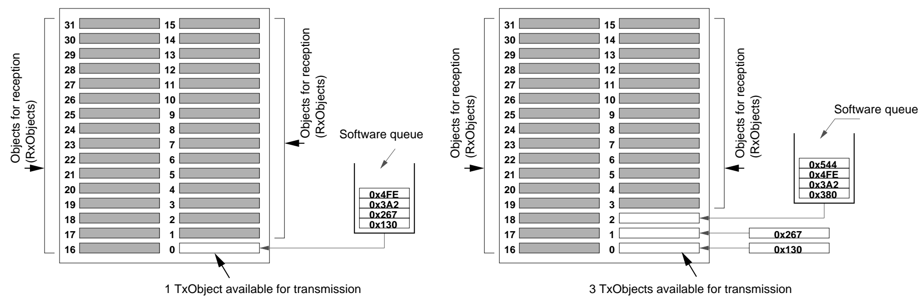

Figure 3.4: Temporary queuing of outgoing messages.

Priority inversions when less than 3 TxObjects are available

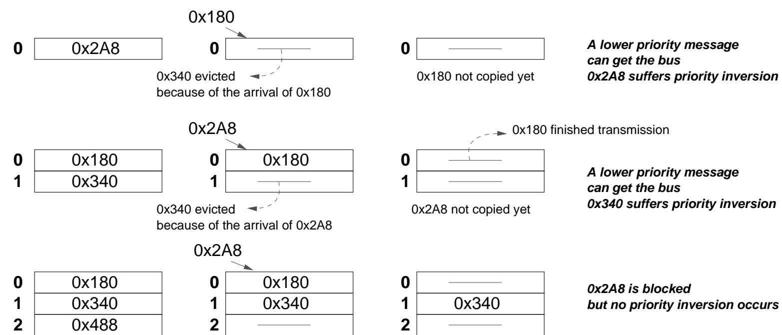

Figure 3.5: Priority inversion for the two buffer case.

Single buffer with preemption. This case was discussed first in [16]. Figure 3.5 shows the possible cause of priority inversion. Suppose message $0x2A8$ is in the only available TxObject when higher priority message `0x180` arrives. Transmission of `0x2A8` is aborted and the message is evicted from the TxObject. However, after eviction, and before `0x180` is copied, a new contention can start on the bus and, possibly, a low priority message can win, resulting in a priority inversion for `0x2A8`.

The priority inversion illustrated in figure 3.5 can happen multiple times during the time a medium priority message (like `0x2A8` in our example), is enqueued and waits for transmisison. the combination of these multiple priority inversions can result in a quite nasty worst case scenario. Figure 3.6 shows the sequence of events that result in the worst case delay. The top of the figure shows the mapping of messages to nodes, the bottom part, the timeline of the message transmission on the bus and the state of the buffer. The message experiencing the priority inversion is labeled as $\mathbf{M}$, indicating a "medium priority" message.

Suppose message $\mathbf{M}$ arrives at the queue right after message $\mathbf{L}_1$ started being transmitted on the network (from its node). In this case, $\mathbf{M}$ needs to wait for $\mathbf{L}_1$ to complete its transmission. This is unavoidable and considered as part of the blocking term B_i. Right after $\mathbf{L}_1$ ends its transmission, $\mathbf{M}$ starts being copied into the TxObject. If the message copy time is larger than the interframe bits, a new transmission of a lower priority message $\mathbf{L}_2$ from another node can start while $\mathbf{M}$ is being copied. Before $\mathbf{L}_2$ ends its transmission, a new higher priority message $\mathbf{H}_n$ arrives on the same node as $\mathbf{M}$ and aborts its transmission. Unfortunately, while $\mathbf{H}_n$ is being copied into the buffer, another lower priority message from another node, $\mathbf{L}_3$ can be transmitted. This priority inversion can happen multiple times, considering that transmission of $\mathbf{H}_n$ can be aborted by another message $\mathbf{H}_{n-1}$, and so on, until the highest priority message from the node, $\mathbf{H}_1$ is written into the buffer and eventually transmitted.

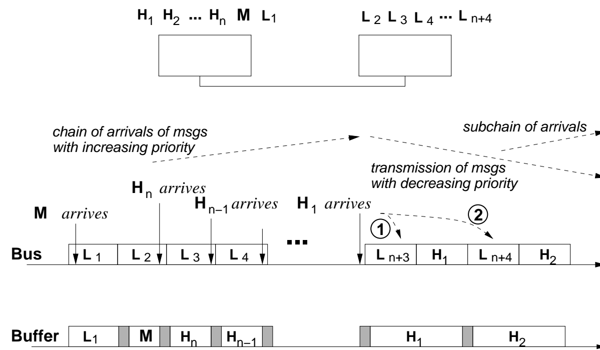

Figure 3.6: Priority inversion for the single buffer case.

The priority inversion may continue even after the transmission of $\mathbf{H}_1$. While $\mathbf{H}_2$ is being copied into the buffer, another lower priority message from another node can be transmitted on the bus and so on. In the end, each message with a priority higher than $\mathbf{M}$ allows for the transmission of two messages with priority lower than $\mathbf{M}$, one while it is preempting the buffer and the other after transmission. Consider that during

the time in which the queue of higher priority messages is emptied (in the right part of the scheduling timeline of Figure 3.6), a new chain of message preemptions may be activated because of new arrivals of high priority messages. All these factors lead to a very pessimistic evaluation of the worst case transmission times.

Please note, if message copy times are smaller than the transmission time of the interframe bits, then it is impossible for a lower priority message to start its transmission *when a message is copied right after the transmission of another message* and the second set of priority inversions cannot possibly happen. In this case, the possibility of additional priority inversion is limited to the event of a high priority message performing preemption during the time interval in which interframe bits are transmitted. To avoid this event, it is sufficient to disable preemption from the end of the message transmission to the start of a new contention phase. Since the message transmission is signalled by an interrupt, the implementation of such a policy should not be difficult. In this case, even availability of a single buffer does not prevent implementation of priority based scheduling and the use of the feasibility formula in [15], with the only change that, in the computation of the blocking factor B_i, the transmission time of the interframe bits must be added to the transmission time of local messages with lower priority. In the case of a single message buffer and copy times longer than the transmission time of the interframe bits, avoiding buffer preemption can improve the situation by breaking the chain of priority inversions resulting from message preemption in the first stage. However, additional priority inversion must be added considering that lower priority messages from other nodes can be transmitted in between any two higher priority message from the same node as M_i.

Dual buffer with preemption In [11] the discussion of the case of single buffer management with preemption was extended to the case of two buffers. Figure 3.5 shows a priority inversion event and Figure 3.7 shows a combination of multiple priority inversions in a worst-case scenario that can lead to large delays. The case of Figure 3.5 defines a priority inversion when `0x340` is evicted from the TxObject while the message in the other TxObject (`0x180`) is finishing its transmission on the network. At this time, before the newly arrived message is copied into the TxObject, both buffers are empty and a lower priority message from a remote node can win the arbitration and be transmitted. `0x340` experiences in this case a priority inversion. As for the description of the effect of multiple instances of this event, in the top half of figure 3.7, the scheduling on the bus is represented (allocation of messages to nodes is the same as in Figure 3.6). In the bottom half, the state of the two buffers is shown. We assume the peripheral hardware selects the higher priority message for transmission from any of the two buffers.

The initial condition is the same as in the previous case. Message $\mathbf{M}$ arrives at the middleware queue right after message $\mathbf{L}_1$ started being transmitted and it is copied in the second available buffer. Before $\mathbf{L}_1$ completes its transmission, message $\mathbf{H}_n$ arrives. The transmitting buffer cannot be preempted and the only possible option is preempting the buffer of message $\mathbf{M}$. Unfortunately, during the time it takes to copy $\mathbf{H}_n$ in this buffer, the bus becomes available and a lower priority message from another node ($\mathbf{L}_2$) can perform priority inversion. This priority inversion scenario can repeat multiple times considering that a new higher priority message $\mathbf{H}_{n-1}$ can preempt $\mathbf{M}$ right before $\mathbf{H}_n$ ends its transmission, therefore allowing transmission of another lower

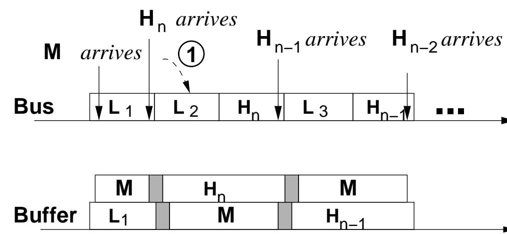

Figure 3.7: Priority inversion for the two buffer case.

priority message during its copy time, and so on.

The conclusion of [11] is that the only way to avoid having no buffer available at the time a new contention starts, which is ultimately the cause of priority inversion from lower priority messages, is to have at least three buffers available at the peripheral and sorted for transmission priority according to the priority of the messages contained in them.

Message queue not sorted by priority

Using FIFO queuing for messages inside the CAN driver/middleware layers may seem an attractive solution because of its simplicity and the illusion that a faster queue management improves the performance of the system. When the message queue is FIFO, preemption of the TxObjects makes very little sense. In this case, a high priority message that is enqueued after lower priority messages will wait for the transmission of all the messages in front of it. Any message in the queue will add to its latency, the transmission latency of the messages enqueued before it, with a possible substantial priority inversion. However, there is a case where priority inversion will not occur. This may happen when messages are enqueued by a single TxTask, the TxTask puts them in the FIFO queue in priority order, and the queue is always emptied before the next activation of the TxTask. The latter condition may be unlikely, considering that the activation period of TxTask is typically the greatest common divider of the message periods.

Messages sent by index of the corresponding TxObject

For the purpose of worst case timing analysis, controllers from Microchip [12] and ATMEL [3] exhibit the most desirable behavior. These chips provide at least three transmission buffers and the peripheral hardware selects the buffer containing the message with the lowest ID (the highest priority message) for attempting a transmission whenever multiple buffers are available. Other chips, from Fujitsu [7] , Intel [9] and Infineon [8], provide multiple message buffers, but the chip selects the lowest buffer number for transmission (not necessarily the lowest message ID) whenever multiple buffers are available. Finally, the Philips SJA1000 chip [13], an evolution of the 82C200 discussed in [16], still retains the limitations of its predecessor, that is, a single output buffer. When TxObjects are not preempted, the CAN controller may, at least temporar-

ily, subvert the priority order of CAN messages. If, at some point, a higher priority message is copied in a higher id TxObject, it will have to wait for the transmission of the lower priority messages in the lower id TxObjects, thereby inheriting their latency. This type of priority inversion is however unlikely, and restricted to the case in which there is dynamic allocation of TxObjects to messages, as when the message extracted from the message queue can be copied in a set of TxObjects. In almost all cases of practical interest, there is either a 1-to-1 mapping of messages to Txobjects, or a message queue associated with a single TxObject.

Impossibility of preempting the TxObjects

A special case consists of a single queue sorted by priority, where the messages extracted from the queue use a single TxObject. In addition, the TxObject cannot be preempted, that is, when a message is copied into it, the other messages in the queue need to wait for its transmission. In this case, the behavior is the same as that of a single-position FIFO queue (the TxObject). All the messages in the priority queue may be blocked by a possible lower priority message waiting for transmission in the TxObject. A typical scenario that considers enqueuing of messages by a TxTask is shown in Figure 3.8.

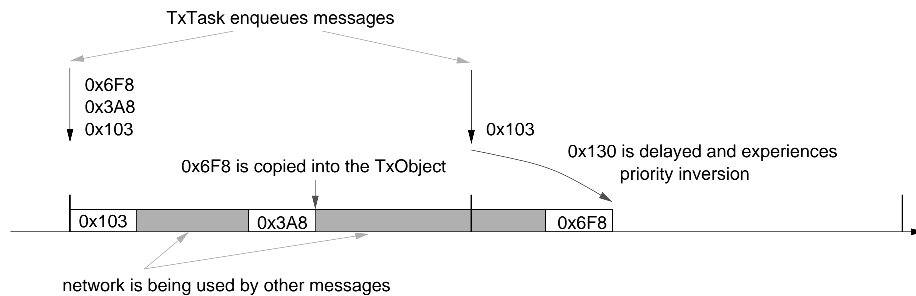

Figure 3.8: Priority inversion when the TxObject cannot be revoked.

In this case, there is a possible priority inversion when a lower priority message (`0x6F8` in the figure), enqueued by a previous instance of the TxTask, is still waiting for transmission in the TxObject when the next instance of TxTask arrives and enqueues higher priority messages (like `0x6F8` in the figure). This type of priority inversion clearly violates the rules on which Eq. (3.4) is derived. Determination od the worst case response time of messages becomes extremely difficult because of the need of understanding what is the critical instant configuration for this case. A way of separating concerns and alleviating the problem could be to separate the outgoing messages in two queues. A high priority queue, linked to the highest priority (lowest id) Txobject, and a lower priority queue, linked to a lower priority object.

Besides the possibility that the same type of priority inversion described in figure 3.8 still occurs for one or both queues, there is also the possibility that a slight priority inversion between the messages in the two queues occurs because of finite copy times between the queues and the TxObjects. variation of the single-available TxOb-

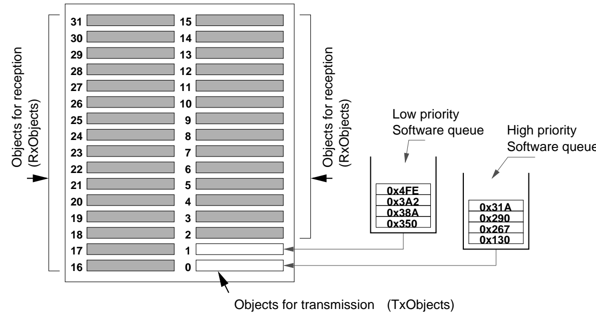

Figure 3.9: double software queue.

ject case occurs. Figure 3.10 shows a CAN trace where a message from the lower priority queue manages to get in between two messages from the higher priority queue taking advantage of the copy time from the end of the transmission of `0x138` to the copy of the following message `0x157` from the high priority queue to the TxObject. In this case (not uncommon), the copy time is larger than the transmission time of the interframe bits on the bus.

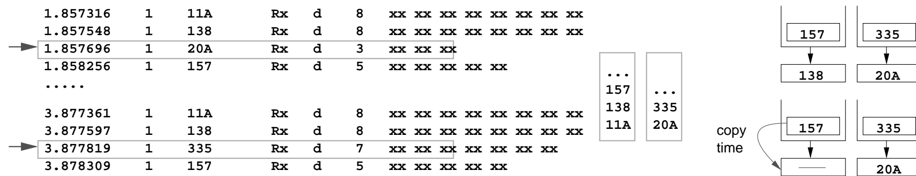

Figure 3.10: Trace of a priority inversion for a double software queue.

Polling based output

From the point of view of schedulability analysis, the case of polling based output can be studied along the same lines of the previous examples and it appears to be definitely more serious than all the cases examined until now. Figure 3.11 shows the basic functioning of a polling-based output and its impact on message latency. The polling task is executed with a period T_p. when it executes, it checks availability of the TxObject. If the TxObject is free, then it extracts the (highest priority) message from the queue and copies in into the TxObject. Then it goes back to sleep. The result of this output management policy is that transmissions from a node are always separated by at least the $T - p$ period. If a message is enqueued with n other message in front of it in the queue, then it will have to wait for (at least) $(n-1)T_p$ before being copied in the TxObject and considered for arbitration on the bus.

Figure 3.12 shows an example of a quite unfortunate, but possibly not even worst case scenario. Following the framework of the previous sections, a message M_i must wait up to t_i^{buf} time units for the worst case transmission of lower priority messages

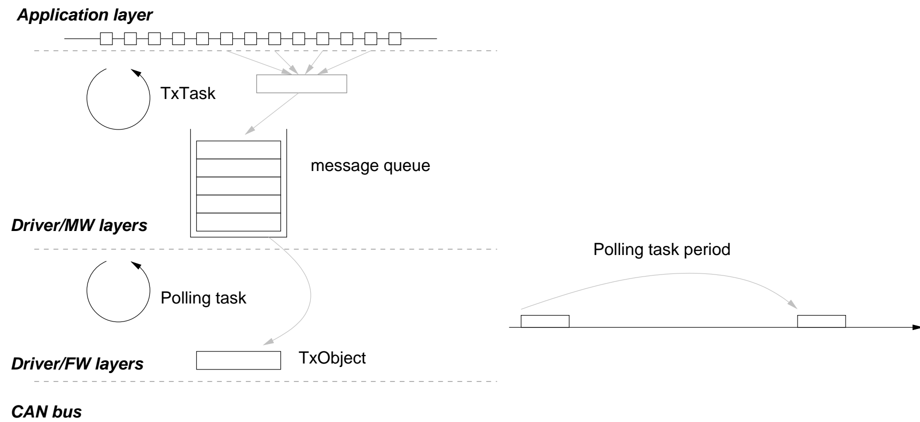

Figure 3.11: The Tx polling task introduces delays between transmission.

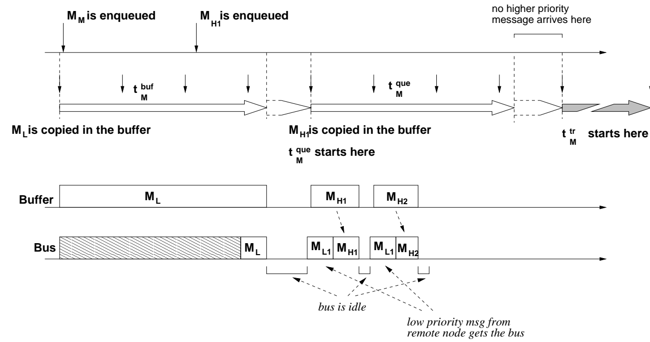

Figure 3.12: Priority inversion for polling-based output.

from the same node.

The time interval of length t_i^{buf} started with the copy of the corresponding message on the peripheral buffer. Assuming perfectly periodic execution of the polling task τ_p, with negligible execution time, the start time of t_i^{buf} is also the reference time for the periodic activation of task τ_p. After t_i^{buf}, the transmission of the lower priority message is completed, but no other message from the same node can be copied into the peripheral buffer until the following execution of the polling task at $\lceil t_i^{buf}/P_p \rceil$. During this time interval, the bus can be idle or used by lower priority messages. For example, right before a new message is copied from the queue into the peripheral buffer, a lower priority message from another node can get the bus. This priority inversion can happen again for each higher priority message sent in the time interval t_i^{que}. Furthermore, the bus scheduling is now an idling algorithm (the bus can be idle even if there are pending requests in the queues of the nodes). Finding the worst case scenario for an idling scheme is not an easy task and very likely to be an NP-hard problem.

The typical result of a node with polling-based message output is shown in Fig-

ure 3.13. The left side of the figure shows (Y-axis) the cumulative fraction of messages from a given medium priority stream that are transmitted with the latency values shown on the X-axis. Despite the medium priority, there are instances that are as late as more than 15 ms (the X-scale is in μs). This is quite unusual, compared to the typical shape of a latency distribution for a message with a similar priority. These messages should not be delayed more than a few ms (Tindell's analysis for this case predicted a worst case latency of approx. 8 ms). However, the problem is that the left hand side node transmits by polling, and the period of the polling task is 2.5 ms (as proven by the steps in the latency distribution.)

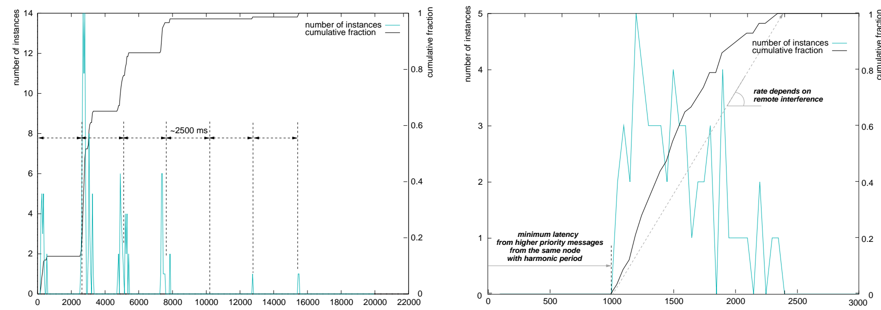

Figure 3.13: Latency distribution for a message with polling-based output compared to interrupt-driven output.

Indeed, the message is enqueued with other messages, in a node with polling-based output. the number of messages in front of it in the queue varies from 1 to 7, which is the cause of the corresponding large worst case latency, and also of substantial jitter.

3.2 Stochastic analysis

3.3 Probabilistic analysis

Bibliography

[1] Road vehicles - interchange of digital information - controller area network (can) for high-speed communication. *ISO 11898*, 1993.

[2] Low-speed serial data communication part 2: Low-speed controller area network (can). *ISO 11519-2*, 1994.

[3] ATMEL. *AT89C51CC03 Datasheet, Enhanced 8-bit MCU with CAN Controller and Flash Memory*. web page: http://www.atmel.com/dyn/resources/prod_documents/doc4182.pdf.

[4] R. Bosch. Can specification, version 2.0. Stuttgart, 1991.

[5] L. Casparsson, Antal Rajnak, Ken Tindell, and P. Malmberg. Volcanoa revolution in on-board communications. *Volvo Technology Report*, 1998.

[6] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real Time Systems Journal*, 35(3):239–272, 2007.

[7] Fujitsu. *MB90385 Series Datasheet*. web page: http://edevice.fujitsu.com/fj/DATASHEET/e-ds/e713717.pdf.

[8] Infineon. *Infineon XC161CJ-16F, Peripheral Units User's Manual*. available from: http://www.keil.com/dd/docs/datashts/infineon/xc161_periph_um.pdf.

[9] Intel. *8XC196Kx, 8XC196Jx, 87C196CA Microcontroller Family User's Manual*. web page: http://www.intel.com/design/mcs96/manuals/27225802.pdf.

[10] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[11] Antonio Meschi, Marco DiNatale, and Marco Spuri. Priority inversion at the network adapter when scheduling messages with earliest deadline techniques. In *Proceedings of Euromicro Conference on Real-Time Systems*, June 12-14 1996.

[12] Microchip. *MCP2515 Datasheet, Stand Alone CAN Controller with SPI Interface*. web page: http://ww1.microchip.com/downloads/en/devicedoc/21801d.pdf.

[13] Philips. *P8xC592; 8-bit microcontroller with on-chip CAN Datasheet*. web page: http://www.semiconductors.philips.com/acrobat_download/datasheets/P8XC592_3.pdf.

[14] K. Tindell, A. Burns, and A. Wellings. An extendible approach for analysing fixed priority hard real-time tasks. *Journal of Real Time Systems*, 6(2):133–151, Mar 1994.

[15] Ken Tindell and Alan Burns. Guaranteeing message latencies on control area network (can). *Proceedings of the 1st International CAN Conference*, 1994.

[16] Ken Tindell, Hans Hansson, and Andy J. Wellings. Analysing real-time communications: Controller area network (can). *Proceedings of the 15th IEEE Real-Time Systems Symposium (RTSS'94)*, 3(8):259–263, December 1994.

List of Figures

2.1 Bit propagation delay. 11
2.2 Definition of the bit time and synchronization. 13
2.3 Relationship between bus length and bit rate for some possible configurations. 16
2.4 Terminating a High-Speed Network. 18
2.5 Placement of Termination Resistors on a Low-Speed Network. 18
2.6 Encoding of dominant and recessive bits. 19
2.7 The CAN data frame format. 20
2.8 The CAN identifier format. 21
2.9 The control field format. 21
2.10 The CRC and Acknowledgement formats. 22
2.11 Masks and filters for message reception. 24
2.12 Bit sequence after detection of an error. 26
2.13 Node error states. 27

3.1 Worst case response time, busy period and time critical instant for M_i. 32
3.2 An example showing the need for a fix in the evaluation of the worst-case response time. 33
3.3 Static allocation of TxObjects. 35
3.4 Temporary queuing of outgoing messages. 36
3.5 Priority inversion for the two buffer case. 36
3.6 Priority inversion for the single buffer case. 37
3.7 Priority inversion for the two buffer case. 39
3.8 Priority inversion when the TxObject cannot be revoked. 40
3.9 double software queue. 41
3.10 Trace of a priority inversion for a double software queue. 41
3.11 The Tx polling task introduces delays between transmission. 42
3.12 Priority inversion for polling-based output. 42
3.13 Latency distribution for a message with polling-based output compared to interrupt-driven output. 43

List of Tables

2.1 Typical transmission speeds and corresponding bus lengths (CANopen) 16
2.2 Bus cable characteristics 17
2.3 Acceptable voltage ranges for CAN transmitters and receivers 19
2.4 27

3.1 Summary of properties for some existing CAN controllers. 34