POLITECNICO DI TORINO

SEDE DI TORINO

SCUOLA DI DOTTORATO

PhD Course in Computer and Control Engineering – XXVII cycle

PhD Dissertation

Deterministic and Flexible Communication for Real-Time Embedded Systems

Tingting HU

Tutor
Prof. Riccardo Sisto

Coordinatore del corso di dottorato
Prof. Matteo Sonza Reorda

Internship Tutor
Dott. Ivan Cibrario Bertolotti

February 2015

To my family

Summary

The research work conducted during the PhD program was focused on real-time embedded systems communication, in particular, for what concerns the *determinism* and *flexibility* perspectives. To this aim, research activities were carried out based on a widespread real-time communication protocol, namely, the *Controller Area Network* (CAN). Indeed, CAN is the de facto standard in automotive and more recently, it also gained popularity in industrial automation and various kinds of networked embedded systems.

Determinism is of paramount importance in communication, since it directly affects timing accuracy. However, deterministic communication cannot be achieved without a comprehensive understanding of possible sources of delay and jitter in a CAN-based distributed system, and more importantly, their countermeasures. Even if all the other sources of jitter are removed or mitigated, the *bit stuffing* (BS) mechanism the CAN physical layer relies on still introduces a *non-negligible* amount of *variability* into the frame transmission time.

To solve this problem, first of all, two novel encoding schemes, namely 8B9B and VHCC, were designed and implemented to prevent bit stuffing in the CAN payload. More specifically, VHCC enhanced the fixed length 8B9B payload encoding in terms of encoding efficiency by means of packing sub-byte information. After that, a *Zero Stuff-bits CRC* (ZSC) mechanism was invented to tackle remaining bit stuffing jitter in the Cyclic Redundancy Check (CRC) of a CAN frame. In this way, it is possible to eliminate bit stuffing jitter *completely* from all over the frame and achieve *jitterless* CAN communication. Concerning this activity, an Italian patent application has been submitted and a European extension is under preparation.

For what concerns flexible communication, we extended CAN in both the real-time and non real-time domains. On the one hand, the application scenarios of CAN were largely broadened by making it support the most widely used general-purpose communication protocol, namely the *Internet Protocol* (IP). On the other hand, in the real-time domain, a transport protocol was designed and implemented to support *Modbus* on CAN. Modbus is an application layer real-time communication protocol widespread in industrial and building automation. This work was adopted by industry for local subsystem communication. And more importantly, it paves the way for CAN in building automation.

Knowledge and experience gained in embedded system design and development are readily applied to further research and transferred into technology suitable for real-world applications.

Acknowledgements

It was a great honor for me to work with Prof. Riccardo Sisto and Prof. Ivan Cibrario Bertolotti during the PhD. There is so much to learn from them, not only because of the deep and broad knowledge they have, but also their passion for research and the way they do it. Special thanks to Prof. Ivan Cibrario Bertolotti, who encouraged me a lot to challenge myself to try different things and have fun in doing research.

Contents

List of Tables

List of Figures

List of Abbreviations

Symbol	Meaning
P	Original payload byte, before encoding
Z	Encoded payload 9-bit value
$f(\cdot)$	Payload encoding function
J	Jitter
p	Payload size, before encoding
DLC	Data field size, after encoding
t_{bit}	Controller Area Network (CAN) bit time
n_X	Length of CAN message section X
id	CAN message identifier
C	CAN message transmission time
h	Number of bits added by bit stuffing
w	Number of consecutive bits at the same value
H	CAN frame header
D	CAN data field
R	CAN Cyclic Redundancy Check (CRC)
U	Unstuffed portion of the CAN frame

Introduction

Nowadays, embedded systems are becoming more and more popular. They are omnipresent not only in our daily life but also in different sections of industry. Even simple consumer appliances like microwave ovens, washing machines, etc., include several microcontrollers to perform predefined sets of functions or procedures. Embedded systems also play a significant role in the development of smart homes and building automation. What's more, they are widely adopted in the transportation industry, especially automotive. For instance, cars are generally equipped with more than 10 embedded nodes, including those used for Anti-lock Braking System (ABS). For what concerns industry automation, embedded systems are deployed in production lines to carry out activities like motion control, packaging, etc.

In the past, the development of embedded systems has witnessed the evolution from *centralized* to *distributed* architectures. This is because, first of all, the easiest way to cope with the increasing need for computing power is to share the load among a larger number of processors. Secondly, centralized systems can not scale up as well as distributed systems as complexity grows. Last but not the least, with time, it becomes more and more important to integrate different subsystems, not only horizontally but also vertically. This goal is easier to achieve with a distributed architecture. For example, the use of buses and networks at the factory level makes it much easier to (remotely) integrate them into the factory management hierarchy so as to support better business decisions.

The main concerns of embedded systems design and development are *different* from general-purpose systems. Embedded systems are generally equipped with limited resources, for instance, small amount of memory and low clock frequency, leading to the need of better code optimization strategies. Secondly, industrial-grade embedded systems are assumed to function correctly and sustainably even under extreme working conditions. This requires a high degree of *reliability*, which encompasses hardware, software and communication protocol design. Another major consideration is *cost*, including both hardware and software. Especially, nowadays software cost is growing and becomes as important as the former one. If existing applications and software modules can be largely reused, this could significantly save time and effort, and hence, reduce cost when integrating different subsystems or upgrading current systems with more advanced techniques and technologies.

Generally, embedded systems also enforce requirements on *real-time* performance. For instance, it could be *soft* real-time in the case of consumer appliances and building automation, instead of *hard* real-time for critical systems like ABS and motion control. Two main aspects directly related to real-time are *delay* and *jitter*. Delay is the amount of time taken to complete a certain job, for example, how long does it take a command message to reach the target and be executed. Variability in delay gives rise to jitter. For example, jobs are completed sometimes

sooner, sometimes later. Hard real-time systems tolerate much less jitter than their counterparts and they require the system behaves in a more *deterministic* way. This is because, in a hard real-time system, deadlines must always be met and any jitter that results in missing the deadline is unacceptable. Instead, this is allowed in soft real-time systems, as long as the probability of occurrence is sufficiently small. This also explains why jitter is generally more of concern than delay. Besides, as it can be seen, in several circumstances, if the commitment to real-time is not met, it could easily lead to *safety* issues, which is another big concern in embedded systems.

Embedded systems are generally network-based. In particular, Controller Area Network (CAN) is a real-time communication protocol, which was originally introduced for automotive applications and is nowadays widely adopted in a variety of networked embedded control systems. It is a serial bus which allows multi-master access to a shared medium. Bus contention is resolved by a distributed non-destructive arbitration mechanism, which makes use of the priorities implicitly assigned to colliding messages. Hence, it represents a quite *reliable* communication channel. The fact that even low-end microcontroller chips are equipped with at least one CAN controller makes it quite *cost effective*. Not even to mention that, due to the bus architecture, no extra intermediate devices are required. Most importantly, it provides off-the-shelf real-time support by means of the above-mentioned priorities, which could be adopted to optimize control scheduling.

At the physical layer, CAN makes use of *Non-Return to Zero* encoding with *Bit Stuffing* (BS). More specifically, every time 5 consecutive bits at the same value are transmitted on the bus, the CAN controller will automatically transmit a *stuff bit* at the *opposite* level. BS ensures that, regardless of the frame content, a sufficient number of edges appear in the signal sent on the bus. They are exploited by receivers to synchronize their clock with the transmitter's so that frames can be received correctly. Unfortunately, BS also introduces undesirable jitter into the frame transmission time, which in turn affects quality of control. What's more, the number of stuff bits inserted into a frame depends not only on the frame length but also on its *content*.

Bit stuffing applies to the header, payload, as well as the Cyclic Redundancy Check (CRC) parts of a CAN frame. Stuff bits added to the header do not lead to communication jitter. This is because, the header is generally fixed and known in advance for a certain bit stream. Hence, the number of stuff bits added to it is also *constant*. Instead, the payload comes from the application layer and it varies from one message to another. Hence, it is impossible to predict beforehand the precise number of stuff bits added to each individual payload. For what concerns the CRC, it is even harder to prevent stuff bits from being added to it. First of all, its exact value depends on both the header and the payload. What's more, it is calculated by the CAN controller hardware at run time. There is no simple remedy for it.

In the past, several approaches have been proposed to prevent bit stuffing from the payload, including XOR-based approaches, Software Bit Stuffing (SBS) as well as the Eight to Eleven Modulation (EEM). They work by modifying the payload in a *revertible* way so that less/no stuff bits will be added by the CAN controller during transmission. XOR-based approaches carry out an exclusive OR operation between the original payload and a specific alternating bit pattern. They reduce the likelihood of bit stuffing, but there is *no* guarantee that it is completely prevented. SBS performs bit stuffing in advance in software so that no more than 4 consecutive bits at the same value can be found in the data field. Instead, EEM encodes every byte of the original payload into a 11-bit pattern so that no stuff bits are required in the encoded data field. Both SBS and EEM can completely prevent bit stuffing from the payload.

We proposed a fixed-length payload encoding scheme, namely 8B9B. Basically, every single byte of the original payload is translated separately to a suitable pattern made up of 9 bits. The data field is obtained by concatenating all these patterns in the original order. For what concerns the 9-bit patterns, no more than 2 consecutive bits at the same value can be found at the beginning and at the end of it, as well as no more than 4 consecutive bits at the same value can be found within the pattern. As a result, when they are concatenated, no more than 4 consecutive bits at the same value can be found in the encoded payload.

With respect to SBS, it is more time efficient since SBS works bit-by-bit while 8B9B is byte-by-byte. At the same time, 8B9B exhibits lower communication overhead than EEM. In addition, lookup tables are adopted in both 8B9B and EEM to facilitate encoding/decoding process, but the tables used by 8B9B are just about *one tenth* the size of those used by EEM. From this point of view, 8B9B is also memory usage friendly. Last but not the least, 8B9B codec has been highly optimized so that encoding/decoding can be performed in a completely deterministic way, even on dissimilar architectures, in a couple of microseconds. This is important, because otherwise communication jitter is just traded for software processing jitter.

A small limitation of 8B9B is that some padding is needed at the end of the data field to make the data field still an integer number of bytes. This is simply because the data field is obtained by concatenating multiple 9-bit patterns. In order to further improve the amount of information embedded in the data field so as to make better use of allocated bandwidth, we proposed another encoding scheme which is not only able to perform byte-by-byte encoding but also able to pack *sub-byte* application-level information, instead of padding, in the data field. This enhanced scheme is called Variable-length High-performance Code for CAN (VHCC). It makes use of an important *nesting* property of the codebook used by 8B9B, that is the same codebook can be used to perform *any* N bit to N+1 bit encoding, where N is within the range of 1 to 8, while preserving the same properties as 8B9B. This indicates that when the 9-bit patterns are concatenated with the encoded sub-byte information to build the data field, bit stuffing is still prevented.

An alternative approach to make even better use of the padding field is to exploit it for the prevention of bit stuffing from the CRC as well. In this way, transmission jitter due to bit stuffing can be prevented all over a frame, which leads to deterministic communication. However, as aforementioned, it is not a trivial task since the CRC is calculated by hardware at run time and its value highly depends on the payload content. Another important evidence is that, no existing solutions are available to solve it. However, by exploring the *linearity* property of the CRC value calculation done in CAN, theoretical analysis proves that it is always possible to tune the CRC to a value that does not require bit stuffing, by just making use of 3 bits at the very end of the data field. This mechanism was called Zero Stuff-bit CRC (ZSC). What's more, ZSC is *compatible* with other existing approaches, which are able to prevent BS from the payload. An Italian patent application concerning ZSC has been submitted. A European extension is under preparation.

Besides introducing jitter into CAN frame transmission, bit stuffing also *weakens* the CRC error detection capability in a significant way. More specifically, as pointed out in several works, just *two* channel errors may lead to an error scenario that goes *undetected* by the CRC instead of six, as its Hamming distance suggests. This severely affects the *integrity* of data exchanged over the network. Simulation results showed that, by simply adopting suitable payload encoding mechanism like 8B9B, the residual error probability can be reduced by two orders of magnitude in the best case.

Another topic investigated intensively during the PhD is to extend the *flexibility* of CAN. This goal has been achieved by broadening the set of high-level protocols supported by CAN. In this way, we make CAN a more appealing choice to system designers as more freedom is given to the application level. At the same time, good support for various protocols saves time and effort for system development and reduces time to market. Main high-level protocols supported on CAN include: SAE J1939 and ISO 11783 for communication/diagnostics among vehicle components, CANopen and DeviceNet for industrial automation, ARINC 825 for local subsystem communication in civil aviation industry. We further extended the flexibility of CAN in two directions: the non real-time and real-time domains.

For what concerns the non real-time domain, support for the Internet Protocol (IP) is the ideal option since it is the most widely adopted protocol ever and an enormous quantity of software has already been developed for it. To this aim, we designed and implemented the IP over CAN (IoC) protocol, which permits IP datagrams to be exchanged between CAN segments and other network segments based on, for instance, Ethernet. In this way, CAN segments which are deployed at the factory level and devoted to real-time purposes can be easily integrated into Intranet, which makes non real-time activities like diagnostics, remote configuration and system monitoring much easier to carry out. When integrating two different types of subsystems, *coexistence* between them is often the major concern. In particular, the interference introduced by the non real-time traffic to the real-time performance of the CAN segment has been carefully studied and proper countermeasures are adopted to reduce it to the minimum. In addition, experimental results also showed that the non real-time performance is still comparable to what can be achieved on an Ethernet link.

For what concerns the real-time domain, CAN has been extended to support Modbus, which is a well-known serial communication protocol widely adopted for connecting industrial electronic devices and building automation equipment. Existing data link layer support for Modbus are RS485 and Ethernet. RS485 is becoming obsolete due to its extremely low bit rate ($\leq$ 19200 bps), while Ethernet link is becoming more and more popular, since it relaxes significantly the limitation on link speed. However, the penalty to pay for extra cabling and intermediate devices like switches is *not* trivial. CAN is a proper compromise between link speed (1 Mbps) and cost. To this purpose, a transport protocol, namely Modbus CAN, has been designed, implemented and formally verified. Experimental results show that, counterintuitively, Modbus CAN demonstrated better real-time performance than Modbus TCP in terms of both communication delay and jitter. Last but not the least, Modbus CAN has been adopted in two industrial application scenarios, namely, local subsystem communication in building automation and backbone communication in professional cooking appliances.

The dissertation is divided into 3 parts. Part I provides a general introduction to CAN and explains the main sources of delay and jitter in a CAN-based distributed system. Part II focuses on deterministic communication in CAN. It presents in detail the 8B9B, VHCC, and ZSC mechanisms along with their effects on CAN error detectability. Part III concludes the dissertation by providing insights on flexible CAN communication by means of the IP over CAN and Modbus CAN protocols. At the end, we draw some conclusions.

Part I

Controller Area Network (CAN)

Chapter 1

The CAN Protocol

1.1 Introduction

CAN stands for *Controller Area Network*, which is a real-time communication protocol that allows microcontrollers to communicate with devices without the need for a host computer. It was first released as a vehicle bus standard by Bosch in 1983. Although it was originally conceived for automotive applications, nowadays it has also become popular in industrial automation and networked embedded systems because CAN is very stable and quite inexpensive. Many recent microcontrollers, even low-end ones, embed one or more CAN controllers, which makes this communication technology very cost-effective and easy to implement.

The latest version of CAN specification is CAN 2.0 [82], which was published by Bosch in 1991. This specification is divided into two parts: part A and part B, where part A is for the standard CAN frame format with 11-bit identifiers and part B defines an extended CAN frame format, in which 29-bit identifiers are used. With the extended frame format, it is possible to define a larger address range. By convention, frames with 11-bit identifiers are called CAN 2.0A frames whereas frames using 29-bit identifiers are commonly called CAN 2.0B frames. For the sake of simplicity, just CAN 2.0A frames are considered in this paper, as it is by far the most popular format adopted in real-world applications. The same kind of reasoning, however, can be applied with little or no change to the CAN 2.0B frame format as well. The data link layer of the CAN protocol has also been standardized in ISO 11898-1 [44], which is consistent with Bosch CAN 2.0 specification.

Figure 1.1 (next page) demonstrates the architecture of a CAN bus. Two or more nodes can connect to the same bus and communicate with each other. The complexity of a node can range from a simple I/O device up to an embedded computer with a CAN interface and sophisticated software running on it. From the hardware point of view, each node contains at least a CAN controller and a CAN transceiver. Generally, the CAN controller is part of the microcontroller. The CAN controller stores the received frames, which can then be fetched by the microcontroller for processing, as well as transmits frames coming from the microcontroller onto the bus when it is free. The CAN transceiver is responsible for converting CAN bus levels to the levels that the CAN controller uses or vice versa, for the incoming or outgoing data stream respectively. Each node is able to send and receive messages, but *not* simultaneously. This is simply because, CAN

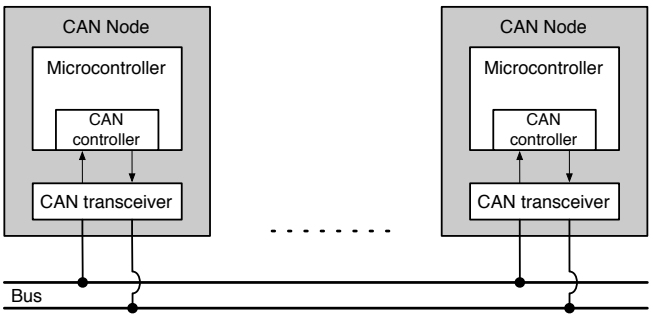

Figure 1.1. CAN bus architecture (single segment).

bus is a collision channel and only one node is allowed to transmit at a time. In this case, while a node is transmitting, other nodes can just receive from the bus. What's more, CAN bus is also a broadcast channel, which means that when a frame is sent on the bus, all nodes connected to the same bus are able to receive it. However, they can choose to receive the frame or ignore it, depending on whether they are intended to process the message or not.

CAN supports bit rates in the range of lower than 1 Kbps up to 1 Mbps. Nodes connected to the same CAN bus should be configured to use a uniform and fixed bit rate, starting from their own clock generators. The maximum bit rate of 1 Mbps is possible only when the bus length is less than 40 m. This property is essential to guarantee the arbitration mechanism work properly. More information about this can be found in Section 1.3. Instead, a longer network distance can be supported by lowering down the bit rate. For example, the bus can be extended to 500 m when a bit rate of 125 Kbps is used.

If higher bit rate is needed, it is possible to seek CAN FD (CAN with Flexible Data-Rate [83]) which has been released by Bosch just in 2012 to cope with increased communication requirements. With respect to CAN 2.0, a different frame format is used, which allows up to 64 bytes of application-level data to be transmitted in a frame instead of 8 bytes in a classic CAN frame (both CAN 2.0A and CAN 2.0B frames), and it is also possible to switch to a faster bit rate after the arbitration is decided. According to the CAN FD specification [83], CAN FD devices are able to coexist with existing CAN 2.0 devices in the same network. However, this is possible in the condition that the same frame format and the same bit rate are used by both types of devices. This means that CAN FD devices function like CAN 2.0 devices in this case. My work is done mainly based on CAN 2.0, on which most industrial applications are deployed. As a result, the following discussion will just focus on CAN 2.0.

In this chapter, first of all, a short introduction to the CAN frame format will be given in Section 1.2. The arbitration mechanism which is used to resolve bus contention when more than one node attempt to transmit together will be discussed in Section 1.3. In a distributed control system, synchronization among different nodes is important to guarantee that they can communicate with each other correctly. More information about this will be given in Section 1.4. The bit stuffing mechanism discussed in Section 1.5 is used to assist and make sure that synchronization is carried out properly. Last but not the least, error handling in CAN is discussed in Section 1.6.

1.2 CAN Frame Format

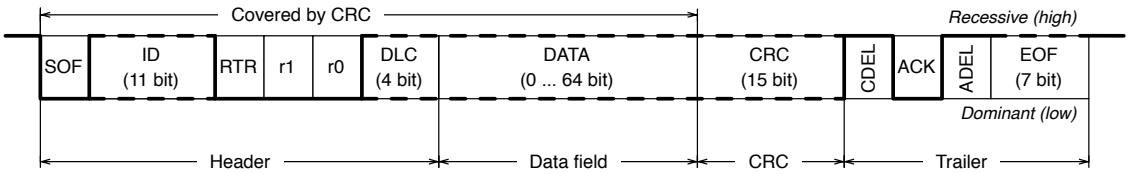

Figure 1.2. CAN 2.0A data frame format (before bit stuffing).

Figure 1.2 shows the basic CAN 2.0A (with 11-bit identifiers) data frame format. At the physical level, the CAN bus can assume two possible logical values, *recessive* and *dominant*. When multiple nodes transmit simultaneously, the dominant level prevails on recessive and the bus stays at the recessive level when it is idle. By convention, the recessive bus level is denoted as a *high* level, and the dominant level as *low*, like shown in the figure. A data frame consists of:

- The *Start Of Frame* (SOF) dominant bit. It marks the beginning of a frame and gives receivers the opportunity to synchronize with the transmitter.
- The *arbitration* field, made up of the message Identifier (ID) field and the Remote Transmission Request (RTR) bit. This is the portion of frame on which bitwise arbitration takes place. More information about the arbitration mechanism can be found in Section 1.3.

  Different IDs, first of all, represent different types of message. The ID does not indicate anything like the destination address of the message, but it tends to summarize the contents and meaning of the message. Since CAN bus by itself is a broadcast channel, this allows efficient implementation of message filtering on individual nodes. When a message arrives at a node, the node can choose to act on it or not depending on whether or not the message is of its interest. Secondly, the value of an ID indicates the priority of the corresponding type of message. The higher the value, the lower the priority. This point will be better justified with information given in Section 1.3.

  The two different meanings of the identifier are used at different layers of the OSI hierarchy for different purposes: priority information is used at the data link layer when there is bus contention while the identifier also works as a data tag at the application layer. They are represented by the same field of the frame so as to minimize the overhead in CAN.

  The RTR bit is used to distinguish between two kinds of CAN frame: data frame and remote frame[1]. This bit is always dominant in data frames.
- The *control* field, including two reserved bits (r1 and r0) and the Data Length Code (DLC). The DLC indicates the length of the CAN data field—where the message payload provided

[1]If a node acts as a receiver for certain data, this node can request the source node to transmit data frame by sending a remote frame. The remote frame and the corresponding data frame share the same ID. A remote frame has no data field. The DLC field of a remote frame indicates the expected data filed length of the corresponding data frame. Remote frame is out of the scope of this dissertation. No further information will be given.

by the application layer is placed—expressed in bytes. Both reserved bits are transmitted as dominant in 2.0A frames. On the other hand, in 2.0B frames, r1 is redefined as the Identifier Extension (IDE) bit and always transmitted as recessive.

- The variable-length *data* field, which consists of 0 to 8 bytes, containing the message payload.
- The *CRC* field, containing the Cyclic Redundancy Check of the message. It is calculated on all bits from SOF to the end of the data field, included.
- The *CRC* and *ACK delimiters* (CDEL and ADEL). They are transmitted as recessive by the transmitting node and act as a timing buffer around the ACK slot.
- The *ACK slot* (ACK), which is driven to dominant by receivers to acknowledge that they have received the message successfully. In Figure 1.2, a successful reception is assumed.
- The *End Of Frame* (EOF) field that delimits the end of the message. It consists of seven recessive bits.

By convention, the part starting from the SOF until the last bit of DLC is called the *header*, whereas the last part including CDEL, ACK, ADEL and EOF is named the *trailer* of a CAN frame.

1.3 Arbitration

When the bus is idle, any node could start the transmission of a message. If more than one node would like to start transmitting messages simultaneously, the bus access conflict should be resolved. In CAN, it is achieved by the *bitwise arbitration* mechanism which makes use of the arbitration field, mainly the Identifier (ID) part. For CAN 2.0A frames, the ID consists of 11 bits, which can be represented as `ID_10 ... ID_0`. The most significant bit is `ID_10` and it is transmitted first onto the bus.

Basically, the arbitration mechanism implemented in CAN works as follows: During the arbitration phase, every transmitter compares the level of the bit it transmits with the level that is monitored on the bus at that bit time. If they match, the node remains in arbitration and continues to send. Instead, when a bit at recessive is sent and a dominant level is monitored, the node lost arbitration and it should withdraw from arbitration and will not send any more bits on the bus. This indicates another node sent dominant on the bus, which prevails. However, if a node tries to send a dominant bit whereas a recessive level is monitored, it will flag a bit error.

The above process is repeated for all bits in the arbitration field. After arbitration is done, only one node is allowed to keep transmitting on the bus. If there are two messages with exactly the same arbitration field, it will lead to bus contention when both nodes are going to transmitting the remaining parts of their messages. As a consequence, within a single CAN bus, the message IDs should be uniquely allocated. We can also see that, in every arbitration round, the message with the lowest numeric ID wins the arbitration and is transmitted successfully, whereas the others are delayed and retransmitted later. Looking at a sequence of arbitration rounds, the smaller the numeric ID is, the higher the probability the message associated with it will be sent. In this sense,

IDs also determine the *priority* of messages. This topic is addressed formally by schedulability analysis [19].

When a Data frame and a Remote frame corresponding to the same ID are sent at the same time, the last bit of the arbitration field, that is the Remote Transmission Request (RTR) bit, is used to distinguish between them. Data frames always have '0' at that bit, whereas remote frame is represented with '1'. As a result, the node which would like to transmit a Data frame will win the arbitration.

A node which loses arbitration will become a receiver for the current frame time and it re-enqueues its message for the next transmission opportunity, that is when the bus becomes idle again. At that time, if there are still some messages with higher priorities, it will lose arbitration again. As a result, for messages corresponding to real-time traffic with tight timing constraints, ID with higher priority should be allocated for them.

A node which would like to transmit a message, when the bus is idle, will first send a SOF bit. All nodes on the bus will take this bit time as the start of a frame. If they would like to send a message as well, they will compete for bus access by transmitting their arbitration fields.

Even though arbitration is quite effective in resolving bus access contentions, it also brings some limitations into CAN physical layer. In order to guarantee arbitration works reliably, the signal corresponding to a bit value should be able to propagate to the most remote node on the bus and come back within one bit time. The sampling of the correct bit value from the bus is important to help nodes, which would like to transmit at the same time, to decide whether they should withdraw from the arbitration or not. This limits the *data rate* for a certain bus length due to bus propagation delay: for example, according to the CAN specification [44], the maximum bit rate recommended for a bus length of 40 meters is 1 Mbps, when using an unshielded twisted pair (UTP) cable and ordinary transceivers.

1.4 Synchronization

In CAN communication, transmitter and receivers generally correspond to different nodes of a system and they are running according to their own clocks. When a transmitter is sending a message, the message is sent according to the transmitter's clock. If the receivers' concept of time is different than the transmitter's, for instance, their clocks drift away from each other, it could lead to incorrect reception of messages. As a consequence, synchronization is required.

However, the transmitter clock is not explicitly conveyed along with data on the CAN bus. Receivers have to recover it by themselves and keep their own clocks synchronized with the transmitter clock. This is done at the *bit* level on a *message by message* basis. This makes sense since there is no need to synchronize the clock when the transmitter is not transmitting, namely the bus is idle. Actually, clock synchronization is achieved on the receivers by looking at recessive to dominant edges in the input stream.

As shown in Figure 1.3 (next page), the receiver divides each bit period into an integer number of *quanta* (12 in the figure). The number of quanta contained in a bit time is configurable and can be any value between 8 and 25 [31]. Quanta boundaries (or the length of a quantum) are defined according to the receiver's time base, which is derived from the controller reference clock and never changed.

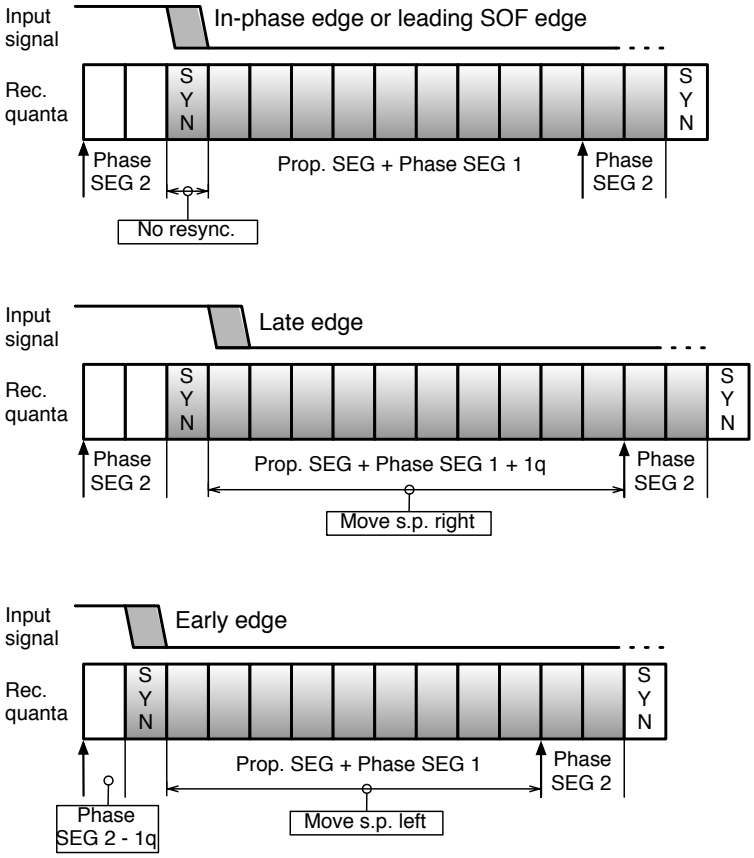

Figure 1.3. CAN bit resynchronization.

The first quantum of a bit is called *synchronization segment* (SYN in the figure). The remaining quanta are divided into two parts: The *propagation time segment* plus *phase segment 1* (sometimes they are called TSEG1 overall, 9 quanta in the figure) and *phase segment 2* (TSEG2, 2 quanta)[2]. At their boundary lies the *sampling point*, where the receiver samples the bus level to ascertain the bit value. It is crucial that the sampling point is positioned appropriately to guarantee that the bit value is read correctly.

The SYN segment is always one quantum long and is used to check whether resynchronization should take place or not. An edge is expected to lie within this segment. The propagation time segment is the time buffer given to allow the signal reach all nodes on the same bus. In other words, it is used to compensate the signal propagation delay on the network. Instead, the phase segments 1 and 2 are used to compensate for edge phase errors. As will be better discussed in the following, these segments can be lengthened or shortened by resynchronization.

[2]The number of quanta for each segment is configurable. Those numbers presented in the text are just an example to help demonstration. More information can be found in [82].

The beginning of a new message is marked by the SOF bit, whereas a message is separated from the preceding message by the *interframe space*, which contains at least 3 recessive bits. Hence, the very first recessive to dominant edge in a CAN message is the leading edge of the SOF bit. This edge constitutes a *hard* synchronization point, because it unconditionally leads receivers to consider the quantum this edge fell into as the SYN of the SOF bit sent by the transmitter, regardless of their previous synchronization state. It is noteworthy that, even after a hard synchronization, the SOF bit as understood by a receiver does not necessarily start immediately after the leading edge of the incoming bit stream, because up to one quantum of uncertainty persists. Depending on where the edge falls within a quantum time (towards the beginning of a quantum or towards the end), the controller software may or may not have enough time to interpret this edge correctly and mark the beginning of the message within the current quantum. Otherwise, it can just mark the next quantum as the SYN of the SOF bit. For the same reason, the time location of the SOF bit may differ by up to one quantum across receivers, even neglecting propagation delays.

As a general rule (with some exceptions better specified in [82, 31]), the edges coming after hard synchronization are used to keep the receiver clock in phase with the transmitter clock and compensate any drift. This is done by comparing their actual positions with respect to the expected bit boundaries according to the receiver clock. To this purpose, there are three possible situations:

- Any edge falling within SYN (as shown in the top of Figure 1.3) is considered in phase with respect to the receiver clock and will not trigger any resynchronization.
- Instead, an edge within TSEG1 (as shown in the middle of Figure 1.3) is considered a "late" edge. In order to compensate, the receiver temporarily lengthens TSEG1 by the amount of measured lateness in quanta (1 quantum in the figure), thus moving the sampling point forward in time and resynchronizing the receiver clock for the next bit.
- The opposite happens when the edge lies within TSEG2 of the previous bit in the receiver's opinion (bottom of Figure 1.3), that is, when the edge is "early." In this case, the receiver designates the quantum in which the edge fell as the SYN of the next bit. This shortens TSEG2 of the previous bit by the amount of earliness. In this case, it moves the sampling point of the current bit backward in time and, once again, resynchronizes the receiver clock.

In a word, a receiver tries to keep in sync with the transmitter's clock by checking where the incoming edge lies with respect to the SYN quantum of a bit time, and then adjusting its concept of time accordingly.

A configuration parameter of CAN controllers, called *synchronization jump width* (SJW) and expressed in quanta, sets an upper limit on the length variation that may be introduced in either TSEG1 or TSEG2, and hence, avoids gross resynchronization errors due, for instance, to noise spikes. The previous examples are unaffected by SJW settings, because its minimum value is 1 quantum.

It is worth remarking that only the recessive to dominant edge is used for resynchronization. This is because the transition from dominant to recessive level is slower (meaning taking longer time) than the opposite transition, due to the hardware implementation. For low bit rate, it is possible to use the dominant to recessive edge for synchronization. However, it becomes a bottleneck when bit rate increases. As a result, the synchronization on edges from dominant to recessive became obsolete when upgrading from CAN protocol version 1.1 to 1.2.

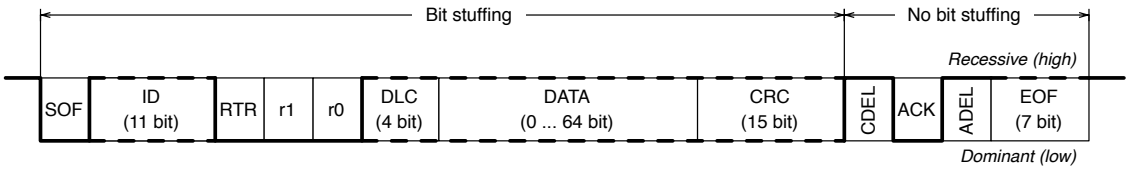

Figure 1.4. CAN frame fields affected by bit stuffing.

1.5 Bit Stuffing Mechanism

The physical layer of CAN relies on the non-return to zero (NRZ) encoding scheme with bit stuffing (BS). Basically, every time the CAN controller in the transmitting node detects that 5 consecutive bits with the same value have to be sent over the bus, it inserts a dummy bit (also called *stuff* bit) at the opposite level. For example, if the original sequence is `01111101 00000001...`, the sequence of bits sent on the bus will be `011111(0)01 00000(1)001...` (from now on, stuff bits are shown within parentheses). On the one hand, due to bit stuffing, the time taken for the transmission of any given frame is not fixed but depends on its content. On the other hand, this rule is very simple and, most importantly, it can be easily reverted by receivers without any "side information". In fact, the symmetric process is carried out by the CAN controller in every receiving node in order to get the original bit stream back.

Bit stuffing ensures that, regardless of frame contents, a sufficient amount of edges appear in the signal sent over the bus, so that receivers can synchronize their digital phase locked loop (DPLL) circuits correctly as explained in Section 1.4. For the same reason, bit stuffing is used in other kinds of link, too [43, 18]. More specifically, if the bit stream to be sent contains a bit sequence of more than 5 consecutive bits at the same level, for example, `0000000` or `1111111`, it will become `00000(1)00` or `11111(0)11` after applying the bit stuffing rule. As we can see, the insertion of a stuff bit is always able to introduce a recessive to dominant transition into the bit stream, which is a key requirement for the bit resynchronization mechanism discussed in Section 1.4.

Not all parts of the frame are encoded according to the BS rules: in particular, BS only applies to those fields from the start of the frame (SOF) up to the cyclic redundancy check (CRC) included. The remaining bits (that is, from the CRC delimiter up to the end of the frame) are not affected by BS. Figure 1.4 shows the main fields affected by bit stuffing in a typical CAN 2.0A frame.

It is worth remarking that, the data field can contain any value, for instance, seven bits at recessive, which is of the same format as EOF. It is important to not mistake part of the data field as the end of frame. As we can see, bit stuffing can help distinguish bit sequences belonging to normal data field from specific frame delimiters as well.

Moreover, carefully selected bit stuffing violations are used to convey other kinds of information on the bus. For instance, the *active error* flag, consisting of a sequence of 6 dominant bits, is transmitted to flag bus errors and globalize them as it will be discussed in Section 1.6.

It is well known that the "raison d'être" for BS lies in its higher efficiency when compared to

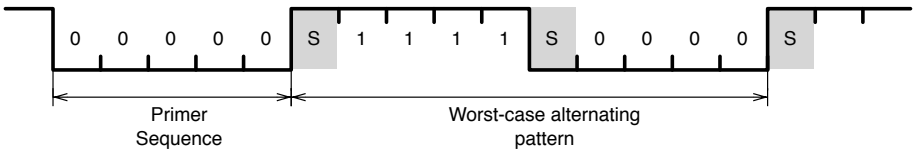

Figure 1.5. Worst-case scenario for BS efficiency.

other encoding techniques (e.g., Manchester encoding[3]). In theory, the worst case for the encoding efficiency of BS (as pointed out in [69]), occurs when the bit stream to be sent consists of 5 bits at the low level, followed by a repeated pattern made up of a group of 4 bits at the high level followed by 4 bits at the low level. Such a pattern is depicted in Figure 1.5, and yields a (worst-case) encoding efficiency equal to about 80% (one stuff bit is inserted every 4 bits in the original stream). In the following, sequences of 5 bits at the same value, either `0` or `1`, are referred to as *primer sequences*.

The 5 initial bits act as a sort of "detonator", which causes the inclusion of the first stuff bit (at the high value) in the encoded stream. Such a stuff bit, together with the following 4 bits (at the high value) in the original stream give rise to a sequence of 5 identical bits that triggers, in its turn, the insertion of an additional stuff bit. This holds for all the subsequent groups of four bits and causes a "domino" effect. It is worth noting that stuff bits are added automatically by the CAN controller and there is no way the mechanism can be disabled. Of course, another worst-case sequence exists, where the logic values of bits are simply reverted: the sequence begins with 5 bits at the high level in this case.

1.6 Error Handling

The CAN specification [44] defines five different kinds of error. Two of them (*bit* and *acknowledgment* error) are detected only by transmitting nodes.

1. A *bit* error happens when the bit value sent by the transmitter is different from the one shown up on the bus. The arbitration field and the ACK bit are two exceptions to this definition, as it will be better explained in the following.

2. An *acknowledgment* error occurs when the bit level of ACK slot seen by the transmitter is not dominant. This indicates that no receiver has received the frame correctly and acknowledged the reception.

The transmitter can detect these two types of error by means of *monitoring* the bus, that is, comparing the bit levels being transmitted with the bit levels detected on the bus. When an error condition is detected and the transmitter is *error active*, it will start to send an *active error* flag on the bus. More information about this will be provided later.

[3] With Manchester encoding, each data bit is expressed by at least one transition.

As mentioned above, the ACK bit and the arbitration field should not be taken into account when considering bit errors. The main reason is that, in both cases, more than one transmitter is allowed in the system within the same bit time. In fact, as explained in Section 1.3, during the arbitration phase, if node A sends a recessive bit while the other nodes are sending a dominant bit, in this case, node A will see the bus at the dominant level which is different from what it transmitted. However, this just means that node A has a priority lower than the others and it will withdraw from the arbitration phase. Instead it will not flag a bit error.

As we can see, both bit error and acknowledgment error are quite simple and they can be detected by just referring to a bit time. Hence, transmitter-detected errors will not be discussed further in the dissertation. Instead, the other three kinds of error are detected by receivers and they are more complex. For this reason, they are of more interest and the following discussion will focus only on the CAN receiver behavior. The other three types of errors are:

3. A *stuff* error occurs when a receiver detects more than 5 consecutive bits at the same level in the part of the frame encoded by bit stuffing, that is from SOF up to and including the CRC field.

4. A *CRC* error is detected when the CRC calculated by the receiver during reception does not match the CRC found in the frame, the one sent by the transmitter.

5. A *form* error is detected when a bit that shall be at a certain level is received at the opposite level. If the CDEL or ADEL bit in a CAN 2.0A frame being received is dominant, this is an example of form error.

   This kind of error should be distinguished from the bit error on the transmit side. Form error concerns more about the correct format of a CAN frame. A bit error on the transmit side may or may not lead to a form error on the receiver side. For example, a bit error in the CDEL bit does, whereas a bit error in the data field will not be seen as a form error on the receiver side because the receiver does not know which value it should be. Instead the receiver may flag a CRC error.

Assuming that the receiving controller is *error active* (a comprehensive discussion of other controller operating modes with respect to errors is beyond the scope of this dissertation, more information can be found in the CAN specification [44]), a key operation that it must perform upon detecting an error is to make all the other nodes aware of the situation, by transmitting an *active error* flag on the bus. This is referred to as *error globalization*. The active error flag consists of a sequence of 6 dominant bits. The flag is selected on purpose to violate the bit stuffing rules, which are essential to CAN communication, so that it will not be taken as part of a normal frame.

Furthermore, it must increment its receive error counter by an amount that depends on the kind of error. Since the same counter is decremented by one upon successful reception of a message, its value is intuitively related to the error rate. It is used by the controller to switch between different error-related operating modes, and by software to assess the bus quality.

It should also be noted that, since the receive error counter is an aggregate value by its own definition, it cannot convey any information about the nature of a specific error instance. No other error reporting mechanisms are mandated by the CAN specification, although specific implementations are allowed to provide them.

Chapter 2

Sources of Delay and Jitter in CAN Communication

Figure 2.1 (next page) depicts the main sources of jitter in a CAN-based communication system. As mentioned in Chapter 1, two or more nodes can be connected to the CAN bus and communicate with each other. Each node is made up of hardware, including at least a CAN controller, and the software running on it. First of all, device drivers are needed to help the application tasks talk with the hardware, and vice versa. In a distributed embedded system, different control algorithms could run concurrently, even on the same node. As the complexity of a system grows and also to meet the real-time requirements, generally a Real Time Operating System (RTOS) is adopted. In turn, it can host diverse communication protocol stacks on top of it. As shown in the figure, each component of the system could contribute to different sources of jitter, from application level tasks down to communication bus.

The sources of communication delay and jitter in an RTOS-based control system can be roughly classified into two categories, namely task level and communication level, depending on the components responsible of them.

2.1 Task-Level Delay and Jitter

In its simplest form, a control algorithm to be executed by an RTOS-based environment can be implemented as a cyclic task τ. The activation of the control algorithm depends on internal timers as well as external events, like network inputs I_j shown in Figure 2.2 (next page). Depending on the specific type of the inputs, various outcomes of the control algorithm are possible and they lead to different actions to be performed, which are embedded in network outputs O_j shown in the same figure. As we can see, the execution of τ by the RTOS is always subject to some amount of delay and jitter, represented by the gray boxes in the figure. This can be further divided into two types and they are due to two distinct reasons:

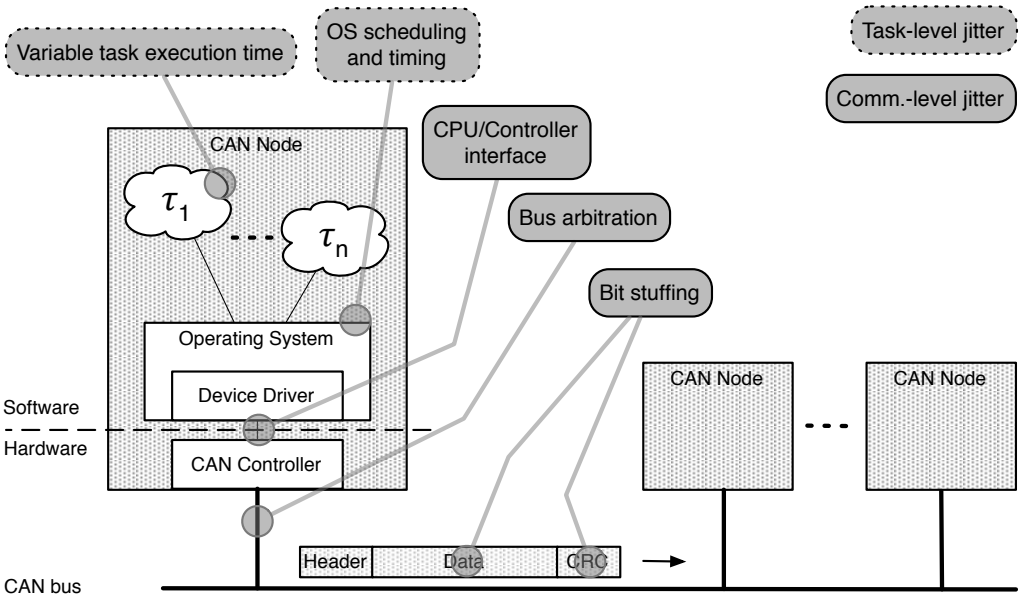

Figure 2.1. Main sources of jitter in a CAN-based communication system.

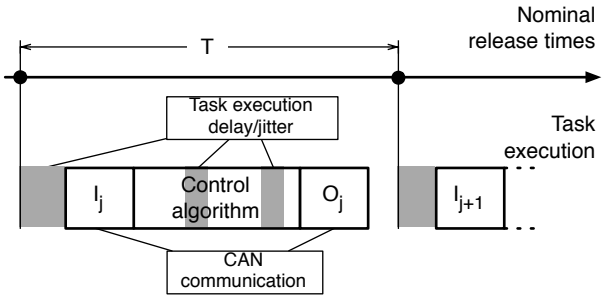

Figure 2.2. Main sources of task-level delay and jitter in an RTOS-based control system.

Variable task execution time

First of all, if τ is not the highest-priority task in the system, the beginning of its execution can be postponed with respect to the nominal instant of release, by a variable amount of time, due to the *interference* it incurs from any higher-priority task. During its execution, it can also be preempted by any higher-priority task that becomes ready in the meantime. In this case, its execution can subsequently be delayed further. As a result, its network input and output operations, represented by the I_j and O_j blocks in the figure, will jitter. This is a well-known aspect of RTOS scheduling theory [90], which allows the worst-case latency and response time jitter that will affect a certain task to be predicted accurately, by considering possible concurrent tasks, their priorities, their nominal release time as well as their periods. The delay and jitter could be reduced through an appropriate *priority assignment*.

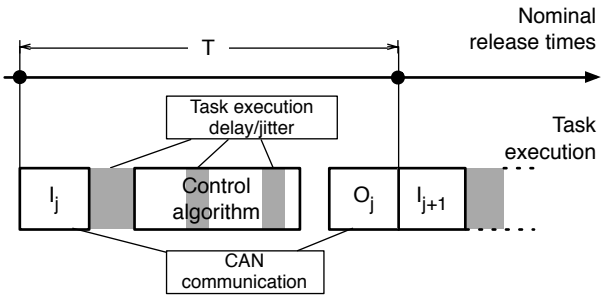

Figure 2.3. Synchronize input and output to reduce task-level jitter.

A simple approach to remove this kind of jitter is to synchronize the input and output operations performed by the control task at each execution with its nominal release time by means of either a hardware interrupt handler or a very high priority task [33]. More specifically, as shown in Figure 2.3, the network inputs will be consumed at the beginning of the cycle time, and the network outputs will be made available to the rest of the system at the end of the cycle time (regardless of when the outputs have been calculated and became ready), while the task execution can take place anywhere within the cycle time. In this way, even if the execution time jitters, the network outputs are always provided to the rest of the system in a deterministic way. The drawback is that, since the outputs are made available just at the end of a period, they can be consumed in the *next* cycle by other tasks, which take those network outputs as their inputs. This introduces a "one sample" delay in the control loop in order to compensate for the execution time jitter. Other more sophisticated task models to address the issue are described, for instance, in [54].

OS Scheduling and timing

Secondly, even though τ is given the highest-priority task in the system, its execution will still be subject to some delay and jitter due to other reasons, not contemplated in the task model. For example:

- In a monolithic RTOS[1], interrupt handlers implicitly have got a priority higher than any regular task in the system. In a modern RTOS, this kind of jitter can be addressed by *temporarily* masking off some interrupt sources within time-critical tasks. They can be unmasked when time-critical tasks complete their work.

  Actually, interrupts can be masked off at two distinct levels: interrupt controller level or processor level. In the first case, it is possible to configure interrupt controller registers to disable the generation of interrupts for individual sources. Instead, the latter means that the hardware could still generate interrupts, but the CPU is not going to handle them until they are unmasked. A simple example to implement masking at the processor level is: since it is common on embedded systems that interrupts are also priority-based, the CPU can select

[1] Operating system architecture in which the entire OS works in kernel space.

to serve interrupt requests only above a certain priority level. An intuitive way to mask off some interrupt sources is to adjust sort of "baseline" priority which is used by the CPU as a reference.

The main difference is that by disabling interrupts, no interrupt will be generated for those sources until they are enabled again. In this case, it is possible to miss events if they are of interests. For this reason, this method is more suitable when applications or the design do not require them and then they can be safely disabled. Another advantage of doing this is that noise on those interrupt lines will not cause problem either. It is obvious that here we are referring to the processor level mask off since after processing time-critical tasks, we would still like to serve any pending interrupt interesting for other tasks.

An alternative solution for jitter introduced by interrupt handling is to keep the execution time of interrupt handlers as short as possible. In this way, jitter is just minimized but can not be completely removed.

- Moreover, the RTOS time-triggered task wakeup mechanisms as well as the task scheduling algorithm may require a *variable* amount of time to be performed. The time-triggered task wakeup mechanisms are implemented to support timeouts which could be specified for tasks or synchronization objects of a task.

  Their execution time varies because traditionally they are implemented based on *queues*, which internally are commonly represented as *linear lists*. In typical designs of RTOS, a task has three states: running, ready, blocked. Running means that the task is currently running on the CPU, ready indicates that the task is ready to be executed, instead a task is in the blocked state when it is waiting for some events, I/O for instance. Correspondingly, the RTOS manages several queues pertaining for different states. For a priority-based preemptive scheduling, generally the ready queue is sorted according to the priority of tasks. When the task currently executing on the CPU finishes its job, the scheduler needs to pick up a task from the ready queue to be executed next. In this case, it should be the one with the highest priority. With a sorted queue, this operation can be completed in constant time. However, another side of the story is that, quite often, it is also necessary to move tasks from other queues to the ready queue. This requires to walk through the list and find the right entry in the list for the task. The execution time of the scheduler varies, depending on the priority of the task being inserted as well as other tasks that are also in the same state in the meantime, which may be different from time to time.

  It is similar for wakeup mechanisms. One additional parameter taken into account when maintaining the list for delayed tasks is their expiration time. In FreeRTOS, a representative RTOS, the scheduler checks the list for delayed tasks every time one tick time elapses. Tick is the minimum unit of time measurement provided by FreeRTOS. If it finds any task whose absolute wakeup time is less than the current time, which means it timeouts, those tasks will be moved to the ready list. Checking completes when the first delayed task with unexpired time is found.

  This kind of jitter can usually be reduced to less than one hundred clock cycles by adopting a more sophisticated algorithm for timer management, such as the one based on *timing wheels* [96], instead of traditional ones, relying on *linear lists* ordered by expiration time.

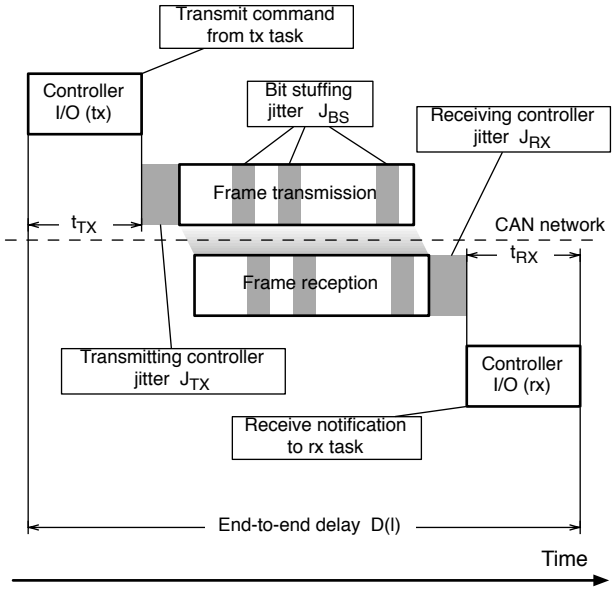

Figure 2.4. Main sources of CAN communication-level delay and jitter.

Briefly speaking, a timing wheel is a construct with a *configurable* fixed-size array. Each element of the array, also called time slot, represents a unit of time. For instance, if the size of the array is 10 and the length of the time unit is 20 ms, it is possible to specify a timeout which is a multiple of the unit time. If the current element being referred to indicates a timeout of 60 ms, then the next element gives a timeout value of 80 ms. Each time slot contains a list of timers which will expire at the corresponding timeout value. It wraps around to 0 if the maximum amount of time interval which could be specified has elapsed, for instance 200 ms in the example. This is quite convenient to implement relative timeouts. It is also important to remark that those lists in timing wheels are much shorter than those in traditional ones. As a consequence, jitter introduced by iterating through the lists could be reduced significantly.

It should be noted that, even if all sources of task-level jitter are either mitigated or removed as discussed above, some important amount of jitter will still be present in the system, due to the network communication itself, represented by the I_j and O_j blocks of Figure 2.2.

2.2 Communication-Level Delay and Jitter

2.2.1 Sources of delay and jitter

Even in a seemingly simple network, such as CAN, the transmission of a message from a task (called *tx task* in the following) to another (*rx task*) is a relatively complex affair, summarized in Figure 2.4. With respect to Figure 2.2, Figure 2.4 represents a single message transfer on the CAN bus. Each step of the transfer gives rise to delay and jitter.

In order to transmit, the tx task must first of all interact with the CAN controller, either directly or through a set of software layers. When carefully designed and implemented, the jitter introduced by the software layers could be negligible. The interaction consists of a fixed sequence of Input–Output (I/O) operations, executed in a *constant* amount of time, which sets up the frame to be sent. The preparation culminates by issuing a "start transmission" command to the controller. Afterwards, the real transmission is subject to different sources of delay and jitter:

1. **Delay and jitter due to blocking or arbitration**. After the command has been sent, the actual frame transmission on the bus will take place after a *variable* amount of time, because the bus may be busy at the moment (*blocking*), or the controller may lose bus arbitration, and thus, defer the transmission. More specifically, CAN follows a *non-preemptive priority-based* access scheme. This means that when the bus is busy, even if the frame being transmitted on the bus currently represents a message with lower priority, all other nodes must wait until its transmission has been completed before attempting access to the bus. Instead, when the bus is idle, due to the bitwise arbitration, the actual frame transmission may be delayed for multiple frame times, depending on the number of nodes trying to transmit data at the same time and their priorities. If a node loses arbitration, it will try to retransmit the frame when the bus becomes idle again. However, the same situation can happen at that time as well. This could easily lead to a chain of effects. The worst-case response time for any message can be evaluated through schedulability analysis [19]. Jitter due to blocking and bitwise arbitration are generally referred to as "frame-level" jitter. As we can see, it is at the magnitude of a couple of frame time.

2. **Delay and jitter due to bit stuffing**. In CAN, for a given payload size, the length of the physical-layer frame, as transmitted on the bus, is not fixed due to the well-known *bit stuffing* mechanism. Indeed, the physical layer of CAN relies on the *non-return to zero* encoding scheme with bit stuffing. Every time the CAN controller in the transmitting node detects that 5 consecutive bits with the same value have been sent, it inserts a *stuff bit* at the opposite level. This indicates that the physical frame length, and hence, its transmission time, also depends on the payload *contents*.

   The worst case, in theory, occurs when the bit stream to be sent consists of, e.g., 5 bits at `0`, followed by a repeated pattern made up of 4 bits at `1` followed by 4 bits at `0`, that is, `00000(1)1111(0)0000(1)...` [69]. This means that the theoretical maximum number of stuff bits that can be added to a frame in CAN, is equal to $\lfloor (34 + 8p - 1)/4 \rfloor = 8 + 2p$, where p is the size in bytes of the payload and instead 34 is the total size in bits of the CAN 2.0A frame header and the CRC. For CAN frames at maximum DLC, namely 8, the number of stuff bits could be up to 24 bits. The best case is that no stuff bits are added at all. This means that jitter due to bit stuffing can be up to 24 bit times. This jitter is often referred to as "bit level" jitter. In practice, the maximum number of stuff bits is always one or two bits less than the above theoretical value, since part of the header has a fixed value (e.g., SOF and the two reserved bits) while DLC is closely related to the payload.

When CAN is used to support event-driven communications, as in the automotive domain, in normal operating conditions "bit-level" jitter is usually much lower than the "frame-level" jitter.

Seemingly, the contribution of BS to the overall jitter could be neglected in these cases. However, a change in the effective duration of the transmission time of any frame affects both the blocking time for higher priority messages and the interference on the lower priority ones. Therefore, bit-level jitters do indeed affect frame-level jitters as well. In order to prevent such kind of dependency, schedulability analysis techniques always assume that the maximum number of stuff bits is added to every message.

When CAN is used to implement tightly-synchronized distributed systems [67], jitters caused by BS can be even more annoying. In order to meet the real-time requirement in such kind of systems, first of all, suitable techniques usually needs to be adopted in order to avoid frame-level jitters, for example:

- A simple approach is using very-high-priority frames for time-critical messages: when the highest-priority CAN identifier is used to this aim, frame-level jitter decreases to just one frame time at worst, corresponding to the blocking time.
- As a simpler alternative, bus contentions can be avoided by using CAN according to a *master-slave* approach. In such kind of systems, only one pair of master-slave nodes is allowed to work on the bus at a single time. The master will not send the next command until it gets the reply for the previous command from the slave. As a consequence, the role of arbitration mechanism is divested during normal system operations.
- In the case *time division multiple access* (TDMA) is exploited, messages are assigned disjoint time slots and they are allowed to be transmitted only in their own slots. Hence, it makes arbitration unnecessary as only one message will be transmitted at a certain instant. TTCAN, which is standardized in ISO 11898-4 [46], is an example belonging to this category. It will be explained in a more detailed way in Section 2.2.2.

After applying those approaches, frame-level jitter can be either reduced to the minimum or completely removed. Countermeasures are usually required in order to reduce bit-level jitters as well, since message reception times should not depend on the specific values carried in the payload. A thorough discussion of bit-level jitter and possible techniques to tackle it, including those designed and implemented by us, will be given in Chapters 3-7. It turned out that it is possible to completely prevent it all overall the frame by adopting appropriate mechanisms, which makes deterministic communication possible.

The aforementioned jitters are mainly due to the way CAN works, namely coming from the CAN protocol itself. Instead, the CAN controller hardware can still introduce additional sources of delay and jitter into the communication:

3. **CPU-CAN controller interface delay and jitter on the transmit side**. It is also important to remark that, even if the bus is idle at the time of the transmission request and no other nodes attempt a transmission at the same time, the CAN controller can *still* introduce a certain non-negligible amount of delay and jitter. Basically, the CAN controller works according to its own notion of bit time. The "start transmission" command issued by the CPU can arrive at the controller interface *anywhere* within a bit time. However, the transmission can be carried out at the next bit time in the best case. This introduce a jitter of up to one bit time into the communication. The lower the CAN bit rate is, the larger the jitter.

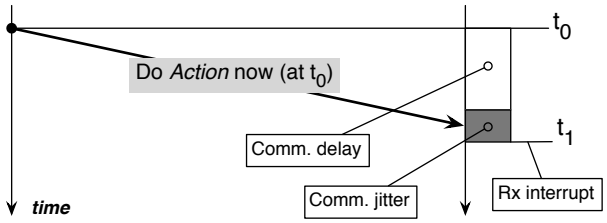

Figure 2.5. Ordinary asynchronous communication.

4. **Notification delay and jitter on the receive side**. After the message has been completely received by the CAN controller, the rx task has to be notified. The details of the notification mechanism (for instance, polling by the rx task or an interrupt request issued by the controller) depend on the implementation, but it will nevertheless represent an additional source of jitter.

Generally, the notification jitter is much *smaller* than the CPU-CAN controller interface jitter. This is because: first of all, if the notification mechanism is polling based, it is generally implemented as *tight* polling loop on the CAN receiver status. Secondly, if notification is done through interrupt request, the time it takes is the propagation delay of the interrupt signal from the interrupt line to the CPU, passing through the interrupt controller hardware. As a result, the notification jitter should be much smaller in both cases than the jitter on the transmit side.

If jitter introduced by the hardware is considered as a whole, we believe that the CPU-CAN controller interface jitter on the transmit side is mainly responsible for it. A jitter compensation technique has been proposed by us. As experimental results explained in [9] shown, jitter introduced by the hardware can be reduced to a *negligible* amount and more importantly, it is *no longer* bit rate dependent.

After the rx task gets the notification, it will typically retrieve the payload from the controller and proceed. As on the transmit side, this interaction consists of a fixed sequence of I/O instructions, and hence, it takes *constant* time to perform.

In principle, an additional source of delay that ought to be considered is the *signal propagation time* on the CAN bus. However—since it is a *fixed* delay that only depends on the electrical properties of the bus—it is not a source of jitter. Moreover, for a short bus, it is likely to be negligible with respect to other delays. For this reason, it will be quietly neglected in the following.

2.2.2 Time-Triggered CAN (TTCAN)

In order to better demonstrate how TTCAN works, first of all, ordinary asynchronous communication will be briefly recalled. Since standard CAN is commonly used in automotive and industrial automation, Figure 2.5 depicts the ordinary asynchronous communication for control in such kind of systems, focusing on the timing perspective.

The requester (on the left), representing a control unit for instance, sends a message to the target (on the right), representing an actuator. The message indicates which *Action* the target must perform. However, the message does not indicate *when* the action shall be performed. Implicitly,

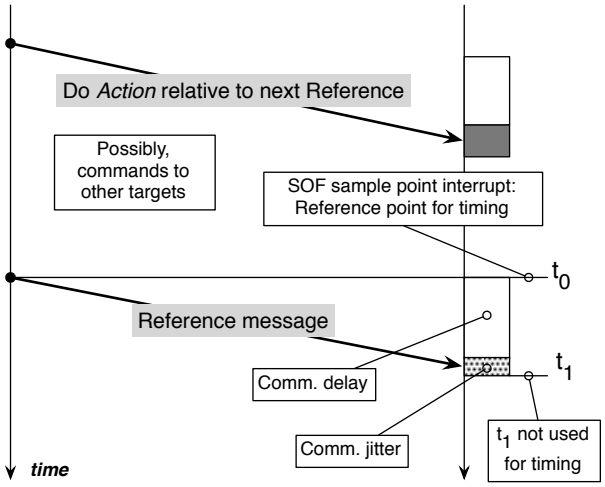

Figure 2.6. Time-triggered TTCAN communication (simplified).

it indicates that the target must perform the action "now". The best approximation of "now" from the target's point of view is to perform the action as soon as it becomes aware that the message arrived, for example, upon receiving a rx interrupt from the CAN controller. As we can see, the concept of "now" is not the same for the requester (to whom "now" is when it sends the message, that is, t_0) and the target (to whom "now" is when it receives the message, that is, t_1).

At the communication level, if *no* countermeasures are adopted, the *Action* is affected by a certain amount of transmission *delay* (as shown by the white rectangle), determined by the message length and the bit rate, as well as some *jitter* (as shown in the dark gray rectangle), which are due to arbitration-related blocking and/or bit stuffing.

This kind of communication is straightforward and easy to implement. It is also quite efficient from the communication point of view, because no other kinds of messages are sent on the bus except those strictly required. If no specific countermeasure is taken, for better quality of control, the delay can be compensated by the requester, if it knows the message length and bit rate (quite likely) by either sending the message in advance or taking the delay into account in the control algorithm. However, the jitter *remains* there. As a result, actuation will still be affected by jitter.

This kind of problem can be solved by adopting a time-triggered approach, like TTCAN. It is a higher layer protocol based on the CAN data link layer protocol. It provides support for distributed real-time control and multiplexing for use within critical parts of in-vehicle systems, like engines.

Figure 2.6 demonstrates in a simplified way the main concept and the general principle of TTCAN. More detailed information can be found in [51, 46]. As discussed before, in TTCAN, access to the bus for transmission becomes time-triggered, that is, nodes are constrained to send messages only within certain time slots assigned to them, according to a *predefined* cyclic schedule. In this way, the medium access scheme changes from carrier sense multiple access (CSMA) as it is in standard CAN to time division multiplexed access (TDMA). As a result, arbitration can no longer introduce any blocking and, more importantly jitter, into the communication because two nodes will never attempt a transmission in the same slot.

The position of a time slot is specified with respect to a *global time*, shared among all nodes on the bus. A consistent notion of the global time among all nodes is fundamental to enforce disjoint time slots. In order to provide this global time, a set of nodes[2] are responsible of sending a *Reference* message cyclically. The distance between two *Reference* messages determines the length of the cycle time. Each cycle contains a predefined number of time slots for all nodes.

For simplicity, Figure 2.6 just shows a single pair of nodes on the bus, representing a requester and a target. This is the basic element to build up a representative TTCAN system. As before, the requester[3] (on the left) sends a message to the target (on the right) to indicate that the target should perform a certain *Action*. It is implicit that the requester transmits the message within its own time slot. The target does not execute the action immediately, that is, when this message is received. Generally, the time the action should be carried out by the target is specified to be a *positive* value relative to the next *Reference* message time. This means that normally the action corresponding to the message received in this cycle time will be performed by the target in the next cycle time.

It should be noted that *Reference* messages are *special*. When a TTCAN controller receives them, it takes the *SOF sampling point* of this message as a timing reference, denoted t_0 in the figure, to implements the concept of global time as well as synchronize its own time with the global time if there is discrepancy between them. Typically, the timing reference is passed to software as well, by means of an interrupt. All nodes perform all their internal activities, including both software and hardware related activities, *synchronously* with respect to the global time. Those activities include: sampling inputs (software part, for example preparing sampling devices for reception), actuating outputs (software part, for example calculating the outputs and preparing the messages to be send) as well as sending messages on the bus through the TTCAN controller hardware. Of course, the access to the bus is done within targets' own time slot.

To sum up, the *Reference* messages are used for two purposes: first of all, by the TTCAN controller to understand more precisely where their time slots are. Secondly, by the software to synchronize processing with communication.

Even though the receive interrupt of command messages (dark gray rectangle) and of *Reference* messages (dotted rectangle) are still affected by delay and jitter, this does not introduce any jitter in *actuation*. This is because:

- It is assumed that the bus is idle when *Reference* messages are transmitted. As a consequence, they will not be affected by either blocking or arbitration jitter. Instead, they may still be affected by bit stuffing jitter. However, the complete reception of *Reference* messages is *not* used to build the concept of global time, only the *SOF sampling point* is. Since SOF is the very first bit of the message, and due to the fact that arbitration and blocking are not involved, the arrival time of SOF bit is always *deterministic*. As a consequence, it can be used as a *reliable* reference for timing. In other words, the difference $t_1 - t_0$, as shown in the figure, becomes irrelevant because in TTCAN the actuation is performed with respect to t_0 instead of t_1, which is the reference used in ordinary asynchronous CAN communication.

[2] A protocol coordinates those nodes to avoid single points of failure and minimize timing disruptions when one of these nodes fails.

[3] The requester may or may not be the same node that sends *Reference* messages. The complexity added to the system in the two different cases and possible side effects are not discussed here for clarity.

- For what concerns the command messages, actions specified in them will not be carried out immediately when they are received. Instead, they will be performed at a predefined time relative to the next *Reference* message. This indicates that, even if the reception of command messages is affected by bit stuffing, which introduces jitter into the transmission, it doesn't affect the actuation time.

To sum up, in each cycle, requesters send commands to targets (in their own time slots) and the targets collect those commands. Actions corresponding to them will be carried out in the next cycle time. Within the current cycle time, and within time slots assigned to them, targets also perform actions (communication part, namely access to the bus) corresponding to commands received in the previous cycle time. As we can see, actions are still done cyclically.

In other words, if we just look at a single target within a single cycle, it receives/collects commands to be executed in next cycle, performs software operations and carries out actions (communication part, namely access to the bus) corresponding to commands received in the previous cycle. If we zoom out and follow a target's behavior in several cycles, it works exactly like a *pipeline*.

The main advantages of TTCAN with respect to ordinary asynchronous communication is that: actuation is affected neither by arbitration-related jitter (since it is time-triggered), nor jitter due to bit stuffing (even though the complete reception of *Reference* message is still affected by bit stuffing). Actuation, or even better control can be performed in a extremely precise way.

However, the side effect is that normal CAN controllers cannot be used in TTCAN for two reasons: first of all, they lack the ability to provide a timing reference at the SOF sampling point. Secondly, it should also be remarked that since transmission to the bus by a certain node can just be done within the assigned time slot, transmission by the hardware should also be time-triggered. However, normal CAN controllers do not support a time-triggered transmission schedule. Another disadvantage is that, it is less efficient from the communication point of view. For example, *Reference* messages unavoidably consume some bandwidth and the time slot reserved to a node could stay empty if that node has nothing to do in a certain cycle.

Part II

Deterministic Communication in CAN

Since it was introduced about 20 years ago, Controller Area Network (CAN) [44] has steadily gained popularity and is nowadays adopted in a variety of embedded, networked control systems. One peculiar facet of the CAN protocol is its *bit-stuffing* (BS) mechanism, an efficient way to ensure that a sufficient amount of edges appear in the signal sent over the bus, and thus, guarantee a proper receiver clock synchronization. However, due to BS, the actual length of a message sent over the bus depends not only on the *size* of its payload but also on its *content*. As a consequence, the exact duration of frame transmissions cannot be known in advance, leading to a certain degree of jitter on response times. The jitter introduced by bit stuffing can be up to 24 bit times, and more importantly it is *bit rate* dependent.

Communication jitter may be, at least in some demanding cases, a limiting factor in the design of a networked control system, both in general [56, 30, 100, 1, 98], and for CAN in particular [67, 69, 99, 84, 77, 55]. Even in the case of simple systems that rely on the master-slave approach, the non-constant duration of CAN messages leads to annoying jitters on actuation.

Although new-generation isochronous protocols—e.g., FlexRay and some real-time Ethernet solutions—are the best option for a new design of a high-performance dependable control system [28], CAN is still a viable option in many other cases [29]. In fact, CAN technology is undoubtedly really stable, well-known and widespread. For these reasons, most microcontroller families from all major vendors include at least one member that embeds, at no extra cost, a CAN controller. As a consequence, when design constraints are not so tight and the focus is instead on reducing system cost, complexity, and time-to-market, many designers still prefer to stick to CAN.

Recently, some researchers started considering the use of conventional CAN also for applications with tight timing constraints, as witnessed by some proposals made for software-based synchronization techniques in CAN. However, a high degree of determinism can be obtained in a distributed system only if the duration of frame transmissions over the network is fixed and known in advance (unless a time-triggered approach is followed). However, due to BS, this is not the case in CAN.

As shown in Chapter 1, bit stuffing applies to three parts of a CAN frame, namely the header, data field and the CRC. The header in CAN frames is made up of the message identifier, DLC and few additional bits whose value is fixed (e.g., SOF and the two reserved bits). Stuff bits in the header are typically not a problem, since in real-world applications both the message identifier and the payload size for a given message stream are usually *not* allowed to change over time. As a consequence, the number of stuff bits that are added by the CAN controller to the frame header is *fixed* and known in advance by the system designer. This means that it does not lead to communication jitters. A number of design hints on this subject are reported in [68].

Instead, the main contribution to BS jitter comes from the data field, as it constitutes the major part of a CAN frame and it comes directly from the application which indicates that its value varies from time to time. Several approaches, like XOR-based approaches [69, 67, 65], software bit stuffing (SBS) [66], eight-to-eleven modulation (EEM) [64], have been proposed in the past in order to tackle with the BS jitter in the data field. The common rationale behind these solutions is to encode the payload in software, so that the hardware insertion of stuff bits in this part of the frame by the CAN controller is either reduced or prevented completely.

Following the same principle, a fixed-length encoding scheme, namely 8B9B which is able to *completely* prevent bit stuffing in the CAN data field, is designed by us. As shown in Chapter 3, 8B9B operates by separately encoding each byte of the payload with a 9-bit codeword, in such a

way that more than 4 subsequent bits at the same level can never be found in the data field. 8B9B also enables efficient software implementations: a high-performance codec was purposely developed for two popular dissimilar embedded platforms, which proves that encoding and decoding can be carried out in just a couple of microseconds and their execution time is always *deterministic*. When comparing to existing approaches, 8B9B exhibits both good encoding efficiency and high jitter reduction capability, even taking into account several different realistic traffic models. The result is presented in Chapter 4. In addition, theoretical analysis proves that 8B9B is the *optimal* choice in its class and this work also leads to an enhanced encoding mechanism called Variable-length High-performance Code for CAN(VHCC), as discussed in Chapter 5. It permits to pack sub-byte information in the encoded data field of CAN frames so that the encoding efficiency is further improved while stuff bits are still completely prevented from the data field.

The CRC field comes immediately after the data field and is the last part of the frame to which BS applies. Unlike the preceding fields, for which countermeasures exist to prevent BS jitter, there is no simple remedy to avoid the insertion of stuff bits in the CRC, because it is computed in hardware by the CAN controller at *runtime*. We proposed a Zero Stuff-bit CRC mechanism, namely ZSC, which is presented in Chapter 6. It makes use of a small piece of the data field–actually just 3 bits are adequate–and then it becomes *always* possible to shape the CRC to a value that stuff bit will *never* be added to it by the hardware during transmission. When ZSC is used together with those encoding schemes that are able to prevent stuff bit completely from the data filed, it is able to achieve truly *deterministic* transmission time on the bus. In this way, it makes the adoption of CAN in systems with tight real-time constraints much easier to achieve.

Last but not the least, for automotive and safety-related applications, *functional correctness* is a main requirement. Those kinds of applications are generally developed as a distributed control system. In order to fulfill the requirement, *data integrity* across the network must be preserved regardless of channel errors, which cannot be completely prevented. The most common way of dealing with errors is to adopt suitable *error detection* mechanisms in the data-link protocol layer and above. However, as pointed out in literature [14, 94], the bit stuffing mechanism interferes with the error detection mechanisms provided in CAN in a significant way. Theoretic analysis and simulation results show that encoding schemes like 8B9B, which are designed to prevent bit stuffing jitter from the payload of a CAN frame, are able to reduce the residual error probability and enhance error detection in CAN. Detailed information can be found in Chapter 7.

Chapter 3

Fixed Length 8B9B Payload Encoding

The Controller Area Network bit stuffing mechanism, albeit essential to ensure proper receiver clock synchronization, introduces a significant, payload-dependent jitter on message response times, which may worsen the timing accuracy of a networked control system. Accordingly, several approaches to overcome this issue have been discussed in literature.

This chapter presents a novel software payload encoding scheme, namely, *8B9B*, which is able to guarantee that no stuff bits will ever be added to the data field by the CAN controller during transmission, and hence, lessens jitters considerably. Particular care has been put in its practical implementation and its subsequent evaluation, to show how the simplicity and inherent high performance of the scheme make it suitable even for low-cost, embedded architectures.

3.1 State of the Art

As several literature papers point out, jitters in CAN, which are due to BS, can be reduced by modifying the payload on the transmitter side so as to remove primer sequences. This process must be *revertible* on the receiver side. For instance, the *XOR* technique in [69] carries out the exclusive OR (EXOR) between the original payload and a specific pattern consisting of an alternating bit sequence. Although *XOR* reduces the likelihood that stuff bits are inserted in the data field—especially when real data is taken into account—there is no guarantee that they are completely prevented.

This approach was enhanced in [67, 65], where EXOR was applied selectively to either the payload as a whole (*SXP*) or on single bytes separately (*SXB*). These techniques were formerly denoted Nolte B and C, respectively. Despite achieving reduced jitter (especially *SXB*), encoding efficiency decreases because of the need to include information about the EXOR process (whether or not, and where to apply it) while encoding time increases. Moreover, some stuff bits may still be added to the data field nevertheless.

Other techniques were subsequently introduced, which are able to prevent the insertion of stuff bits in the data field completely. As an example, the *SBS* approach [66] performs a software bit stuffing in advance on the original payload, so that one software stuff bit is added every time 4 consecutive bits are found at the same value. Processing time increases consequently with respect to *SXP* and *SXB*. In a similar way, *EEM* is a fixed-length 8-to-11 modulation scheme [64], which encodes every byte in the original payload as an 11-bit pattern and proved to be faster than *SBS*.

3.2 8B9B Codec

The *8B9B* encoding scheme is straightforward. Basically, every single byte of the original payload is translated separately to a suitable pattern made up of 9 bits. The encoded bit stream is then obtained by concatenating all these patterns in the original order. Clearly, this does not apply to frames with an empty payload (i.e., when the data field length DLC is 0). Encoded 9-bit patterns must satisfy two properties:

1. The first, basic requirement is that none of them is allowed to include primer sequences. For instance, the pattern `010000011` is unsuitable for our purposes.

2. The second requirement is that primer sequences must never appear in the encoded bit stream, not even across pattern boundaries. In order to meet this additional constraint, all patterns that include 3 (or more) bits at the same level, at either the beginning or the end, have to be discarded as well. For instance, the pattern `010101000` is not suitable because, when followed by the (legitimate) pattern `001010101`, it would give rise to the sequence `010101000 001010101`, which includes a primer sequence.

A simple algorithm was developed to find exhaustively all the 9-bit patterns that satisfy both of the above constraints. The resulting set of candidates includes 258 different elements. This confirms that every single-byte value (from 0 to 255) can be encoded by means of a unique 9-bit pattern. To this aim, 256 patterns were reserved. Two additional patterns exist in the set, which are not used to encode any data byte value. They were labeled J (`001000010`) and K (`110111101`) and could be adopted as escape sequences, but this aspect falls outside the scope of this chapter.

Basically, the above ordered set of patterns represents a *forward lookup table* (FLT) for the direct replacement of bytes in the original payload with corresponding 9-bit patterns. Let P be the original data byte, Z the corresponding *8B9B*-encoded value, and $f(\cdot)$ the encoding function carried out through the FLT. The forward translation process can be synthetically expressed as $Z = f(P)$.

Because of the properties the patterns which were selected satisfy, if Z is a valid pattern also $\sim Z$ is necessarily valid, where "$\sim$" represents the *complement* operator (bitwise NOT), which inverts every single bit in the pattern it is applied to. Since candidate patterns were obtained in increasing numerical order, a basic property of the binary representation of numbers (that is, $2^9 - 1 - Z = \sim Z$) implies that, overall, the FLT is "specular". In formulas

$$Z = f(P) \Leftrightarrow \sim Z = f(\sim P) \ . \tag{3.1}$$

By exploiting this property only the first half of the FLT is required to be stored in real implementations (see Table 3.1).

3.2.1 Break Bit and Padding Field

The reasoning above takes into account the data field alone. Unfortunately, the preceding parts of the frame may contribute to the creation of a primer sequence together with the beginning of the data field. The DLC field is not in complete control of the user, since it specifies the size in

P_{16}	Z_2	P_{16}	Z_2	P_{16}	Z_2	P_{16}	Z_2
00	001000011	20	001101101	40	010100001	60	011001101
01	001000100	21	001101110	41	010100010	61	011001110
02	001000101	22	001110001	42	010100011	62	011010001
03	001000110	23	001110010	43	010100100	63	011010010
04	001001001	24	001110011	44	010100101	64	011010011
05	001001010	25	001110100	45	010100110	65	011010100
06	001001011	26	001110101	46	010101001	66	011010101
07	001001100	27	001110110	47	010101010	67	011010110
08	001001101	28	001111001	48	010101011	68	011011001
09	001001110	29	001111010	49	010101100	69	011011010
0a	001010001	2a	001111011	4a	010101101	6a	011011011
0b	001010010	2b	010000100	4b	010101110	6b	011011100
0c	001010011	2c	010000101	4c	010110001	6c	011011101
0d	001010100	2d	010000110	4d	010110010	6d	011011110
0e	001010101	2e	010001001	4e	010110011	6e	011100001
0f	001010110	2f	010001010	4f	010110100	6f	011100010
10	001011001	30	010001011	50	010110101	70	011100011
11	001011010	31	010001100	51	010110110	71	011100100
12	001011011	32	010001101	52	010111001	72	011100101
13	001011100	33	010001110	53	010111010	73	011100110
14	001011101	34	010010001	54	010111011	74	011101001
15	001011110	35	010010010	55	010111100	75	011101010
16	001100001	36	010010011	56	010111101	76	011101011
17	001100010	37	010010100	57	011000010	77	011101100
18	001100011	38	010010101	58	011000011	78	011101101
19	001100100	39	010010110	59	011000100	79	011101110
1a	001100101	3a	010011001	5a	011000101	7a	011110001
1b	001100110	3b	010011010	5b	011000110	7b	011110010
1c	001101001	3c	010011011	5c	011001001	7c	011110011
1d	001101010	3d	010011100	5d	011001010	7d	011110100
1e	001101011	3e	010011101	5e	011001011	7e	011110101
1f	001101100	3f	010011110	5f	011001100	7f	011110110

Table 3.1. 8B9B forward lookup table (encoding process).

bytes of the payload—represented on 4 bits as a binary value. For this reason, unlike the payload, it cannot be encoded. For example, when the DLC value is 8 (1000) and the first encoded pattern

Original payload size (byte)	Data field in the *8B9B*-encoded frame: Size (byte)	DLC value	BB value	BB size (b)	9-bit pattern sequence (b)	PAD size (b)
0	0	0000	—	0	0	0
1	2	0010	1	1	9	6
2	3	0011	0	1	18	5
3	4	0100	1	1	27	4
4	5	0101	0	1	36	3
5	6	0110	1	1	45	2
6	7	0111	0	1	54	1
7	8	1000	1	1	63	0
8	—	—	—	—	—	—

Table 3.2. 8B9B-encoded data field vs. original payload.

is `001010101`, the resulting (partial) bit sequence is ... `1000 001010101`..., which includes a primer sequence.

A simple remedy is inserting a single bit, called *break bit* (BB), in the very first position of the data field. The value of BB is opposite to the last DLC bit, i.e., it is either `1` when the DLC value is even or `0` on the contrary. Thanks to BB, bit strings with 2 or more bits at the same value cannot appear immediately before the first 9-bit pattern in the encoded payload, for any value of DLC between 1 and 8, included. Only when DLC = 0 or DLC = 15, the BB could be preceded by one stuff bit at its same level, appended to the DLC by the CAN controller. In any case, the occurrence of a primer sequence affecting the data field is completely precluded.

In theory, BB is mandatory only when the DLC is 3 (`0011`), 7 (`0111`) or 8 (`1000`). The reason why it is required also in the first case is that, a stuff bit at `1` could be possibly inserted just after the `00` bit pair, hence giving rise to the sequence `00(1)11`. In practice, two choices are available for the codec: either BB is included only when strictly required or it can be maintained for any frame size. In this chapter we opted for the latter choice, since it does not worsen efficiency appreciably and makes the codec simpler and faster.

A second aspect to be considered is that the bit sequence obtained by the *8B9B* translation process is generally not aligned to a byte boundary. As a consequence, the last byte in the data field may not be filled up completely by actual data (i.e., the encoded payload). In this case, a variable-size *padding* field (PAD) is foreseen to align the encoded bit stream to the next 8-bit boundary. The transmitting node sets PAD to a suitable value that does not cause the insertion of any stuff bit, e.g., an alternating bit pattern (`0101`...).

3.2.2 Encoding Example

Let us consider a very simple one-byte datum at the value `0xf0`. In conventional CAN, this value would fit directly into a single-byte data field (DLC = 1), and would result in the transmitted (partial) sequence ... `0001 1111 (0) 0000 (1)`... (only the DLC and data fields are shown). As can be seen, two stuff bits (shown in parentheses) have been inserted.

Table 3.2 shows that the DLC value in the corresponding *8B9B*-encoded frame is 2 (`0010`). As the DLC is even, BB is set to `1`, which is placed as the starting bit of the data field. Then, every byte in the payload (only one in this case) is translated. According to Equation (3.1), byte `0xf0` is first complemented (it starts with a bit at `1`), which yields `0x0f`; a lookup operation on the FLT in Table 3.1 returns the 9-bit pattern `001010110`, which, when complemented, yields `110101001`. Finally, a padding is appended to the end of the *8B9B*-encoded payload, which consists of 6 bits set to the alternating bit pattern mentioned before. Overall, the transmitted (partial) sequence is `...0010 1 110101001 010101...`, where the DLC, BB, encoded payload (one pattern) and PAD are shown. As expected, no additional stuff bit needs to be inserted into the data field when the frame is transmitted.

Incidentally, *8B9B* is also able to improve the error detection capabilities of CAN slightly. In fact, no invalid pattern should be found in the received payload in normal operating conditions: their presence is an evidence that a transmission error has occurred. This constitutes an additional error detection mechanism and increases reliability further at the expense of an extra decoding overhead. More information about this can be found in Chapter 7.

3.3 Implementation and Optimization

The encoding and decoding algorithms discussed in the previous section have been implemented in the ISO C language [48] and then cross-compiled, using different toolchains, for two dissimilar architectures. Those architectures have been chosen because they are at the extremes of the range of microcontrollers currently adopted for CAN-based applications. Namely, the NXP LPC2468 microcontroller [71] (`arm7` in the following of this chapter) is a low-end component based on the ARM7TDMI-S processor core designed in 2001 and running at 72 MHz. Instead, the NXP LPC1768 microcontroller [73] (`cm3` in the following of this chapter) is based on the contemporary ARM Cortex-M3 processor core running at 100 MHz.

The base version of the C code has then been optimized in several ways, better discussed in the following, by working at both the toolchain and source code levels. Neither hand-written nor hand-optimized machine code has been developed, because one of the implementation goals was to show that it is possible to achieve good optimization results across different processor architectures—by working exclusively at the C language level and without necessarily having a thorough knowledge of the processor machine language level.

For both platforms, the toolchain consists of open-source components only, namely: `binutils` (assembler, linker, librarian), the `gcc` compiler collection, and the `newlib` runtime library. Platform initialization files and linker scripts have been taken from the FreeRTOS [2] operating system.

3.3.1 Determinism and Performance Optimization

The first optimization goal was to achieve full execution time *determinism*, that is, make the execution time of the encoding/decoding algorithms independent from message contents. After all, if full determinism were impossible to achieve, the original source of jitter (CAN bit stuffing) would be simply replaced by another one (data processing jitter due to the encoding/decoding layer,

called J_P in the following of this chapter). On the other hand, a (hopefully linear) dependency on the payload size p is both expected and acceptable. For this purpose, the code has been reworked in two different ways:

- First, all conditional *statements* have been replaced by conditional *expressions*, in order to make their execution time independent from predicate truth. For instance, the execution time of a conditional statement in the form `if (f<0) f = ~f` depends on the sign of `f`, because the bitwise complement and the assignment are performed only if `f` is negative.

  On the contrary, assuming that `f` is an 8-bit variable, the execution time of `m = ((f<0) ? 0x00 : 0xFF)` is constant, because both sides of the conditional expression have got the same structure. Then, the expression `m^f` can subsequently be used to obtain either `f` itself or the bitwise complement of `f` depending on its sign, again, in constant time.

- Moreover, by leveraging a property of the 8B9B encoding and decoding algorithms, it was possible to write down the code so that all iteration statements are executed a number of times that only depends on the payload size, rather than contents.

All in all, after both optimizations have been carried out, the code contains just one loop for the encoder and one for the decoder, and there is a single execution path within these loops. Both loops are always executed once for each payload byte to be encoded and decoded.

The second goal of the optimization was to improve code *performance* as much as possible, to minimize the additional end-to-end response time introduced by the encoding and decoding process. Three kinds of optimization have been identified and implemented.

1. **Code/data placement**

   Microcontrollers usually support different kinds of memory such as Flash memory, Dynamic RAM (DRAM), and Static RAM (SRAM). Each kind of memory has its own peculiar behavior in terms of access speed and determinism. The goal of this optimization step was to force the toolchain to place the code, data, and stack segments of the encoding and decoding routines in the most appropriate place. Even if this is a relatively straightforward decision, most toolchains are unable to take it autonomously. Moreover, as it will be better discussed in Section 3.4, their default placement rules are far from being optimal.

2. **Computed masks**

   In the base implementation shown in [11], several auxiliary arrays hold the bit masks used on every iteration of the encoding and decoding loop, to split the 9-bit value obtained from the forward lookup table into two output bytes during encoding, and join them again during decoding. In this optimization, the masks have been computed directly, as a function of the loop index. Since this computation can be carried out in the processor registers, the extra memory accesses to the arrays are avoided. The arrays themselves can be deleted to save memory, too.

3. **Load/store reduction**

   The base version of the encoding algorithm, on each iteration, loads one input byte and stores some data into two adjacent output bytes. This is because, 8 bits of original payload

lead to 9 bits of encoded payload, which should be stored into two adjacent bytes. For efficiency consideration, the store operation is carried out at the byte level. As a consequence, one of the output bytes overlaps with those of the previous iteration, and hence, each output byte except the first one is accessed twice. With this optimization, a local buffer has been introduced to carry forward the common output byte from one iteration to the next, in order to store it only once. The decoding algorithm has been optimized in the same way, too.

Since these optimizations are independent from each other, each optimization step started from the previous one.

3.3.2 Memory Footprint Optimization

The starting point of the memory footprint optimization is the basic symmetry property of the FLT, Equation (3.1). With the help of this property, as stated in Section 3.2, it is possible to store only half of the table and reduce its size from 256 to 128 9-bit entries. For efficient access, the size of each FLT entry must nevertheless be an integral number of bytes, and hence, each entry actually occupies 16 bit in memory. However, the most significant bit of the FLT is always zero in the first half of the table, as shown in Table 3.1, whereas it is always one in the second half. Therefore, it is not necessary to store this bit explicitly and the FLT can be further shrunk down to 128 8-bit entries, that is, one quarter of its original size.

The quickest method to revert the *8B9B* encoding scheme on the receiver side is to use a reverse lookup table (RLT) that represents the inverse of the encoding function $f(P)$. Since f is injective, but not surjective, not every 9-bit pattern actually corresponds to a valid encoder output. As a consequence, each RLT entry must store two distinct pieces of information: an indication of whether or not the given value of Z is valid, and the original data byte, that is, the value of P for the given Z, if any. The entry size is therefore 9 bits, 1 bit for the flag and 8 bits for P, corresponding to 16 bits in memory. In turn, the total size of the full RLT is 512 16-bit entries.

As for the FLT, the RLT size can be halved by exploiting symmetry, which means that 256 entries are sufficient. Moreover, when doing so, only the 7 least significant bits of each P must be stored explicitly, because the most significant bit of P can be inferred from the most significant bit of Z. Overall, the RLT can hence be reduced to 256 8-bit entries. Of these, 128 entries correspond to either invalid patterns or the J and K escape codes.

Further memory savings would be possible, by noticing that the numerically lowest valid pattern in the halved FLT is `001000011` (67_{10}) while the highest is `011110110` (246_{10}). This means that only 180 RLT entries must actually be stored in memory, whereas the others can be inferred by range checking. However, this opportunity has not been further considered, because preliminary experiments have shown that the overhead associated with it would be excessive.

3.4 Experimental Results

The performance of the code discussed in Section 3.3 has been evaluated by encoding and decoding a set of 10^7 uniformly-distributed, pseudo-random messages of varying lengths. On both architectures, the time spent in the encoding and decoding routines, as a function of the payload size p, has been measured by means of a free-running, 32-bit counter clocked by the CPU

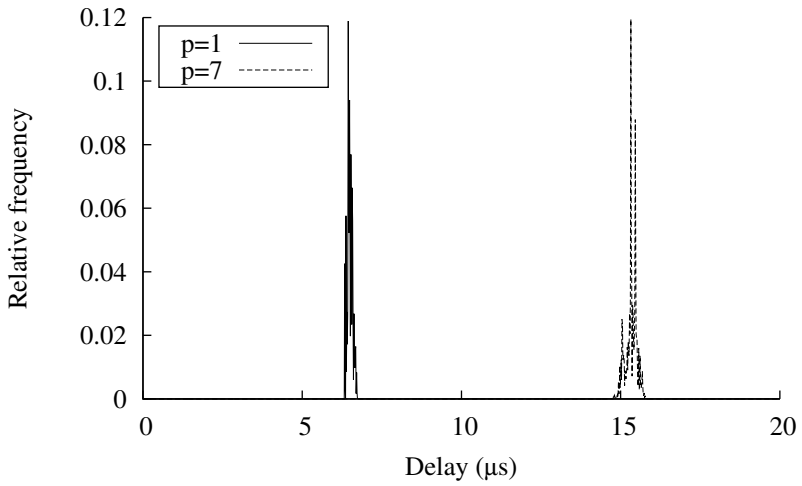

Figure 3.1. Frequency distribution, DRAM-resident `arm7` encoder delay.

core clock. In this way, it was possible to collect a cycle-accurate encoding and decoding delay measurement for each message. Working on a large number of pseudo-random messages was convenient to single out any data dependency. External sources of measurement noise were avoided by running the experiments in a very controlled environment, with interrupts disabled and unused peripheral devices powered down.

3.4.1 Code/Data Placement

A first interesting insight into system behavior is given by the experiments carried out on the `arm7` architecture with the default code/data placement set by the toolchain. On the evaluation board being used, this implies that code, data, and stacks are stored in an off-chip DRAM. Figure 3.1 shows the frequency distribution of the encoder delay in this case, for a payload size of $p = 1$ and $p = 7$ bytes.

The experimental data clearly show that the encoder is unacceptably slow, namely, the amount of time needed to encode a payload of 7 bytes is 15 μs on average. At a bit rate of 500 kbps this corresponds to about $7.5t_{\text{bit}}$, where t_{bit} represents the CAN bit time. Moreover, the delay is also affected by a significant amount of jitter, which increases as p grows, and is in the order of 1 μs in the worst case.

The slowness can be justified by considering that external memory is accessed by means of a 16-bit data bus, to reduce costs. On the other hand, jitter is due to *DRAM refresh cycles*, which stall any instruction or data access operation issued by the CPU while they are in progress. For this architecture, both issues were solved by moving the critical code, data and stack into the on-chip SRAM. After this was done, all the subsequent measurements exhibited *no jitter*.

It must also be remarked that this behavior is not specific to the `arm7` architecture, but is just a special case of a more widespread phenomenon. For instance, the `cm3` architecture exhibited a very similar behavior (not shown in this chapter for conciseness), when the forward and reverse

lookup tables were stored in Flash memory. In this case, the jitter was due to a component—known as *Flash accelerator*—whose purpose is to mitigate the relatively large access time of Flash memory (up to 5 CPU clock cycles), by anticipating future Flash memory accesses. Although the prediction algorithm is quite effective with instruction fetch, it does not work equally well for forward and reverse table lookup, and hence, it introduces a variability in the table access time. Like in the previous case, the solution was to force the compiler to allocate the forward and reverse lookup tables in on-chip SRAM instead of Flash.

3.4.2 Delay Model

Figure 3.2 (plots 1–3, next page) shows the outcome of the optimizations discussed in Section 3.3.1 on the `arm7` architecture. Unlike Figure 3.1, it shows the delay as a function of p. The sample distribution is not shown because, after appropriate code/data placement, the sample variance was zero in all experiments. In other words, during the experiments J_P was consistently below the timer resolution, that is, one CPU clock cycle.

Intuitively, the delay needed to encode or decode a message, where p is the payload size in bytes, can be modeled by $\hat{t}(p)$, defined as

$$\hat{t}(p) = \begin{cases} b & \text{if } p = 0 \\ b + l + qp & \text{if } p > 0 \end{cases} , \tag{3.2}$$

where b is the base delay incurred, regardless of the payload length, because of the function prologue and epilogue, l is the additional processing time needed to set up the encoding or decoding loop when $p > 0$, and q is the time needed to process each byte of the message.

The model parameters have been derived from the experimental data as follows: b has been measured directly, q has been obtained by a linear least-squares fit of the experimental data for $p > 0$, and l has been obtained by difference between the extrapolation of the linear fit to the case $p = 0$ and b. To evaluate the accuracy of the fit, instead of the commonly-used norm of the residuals, the maximum absolute value of the residuals ρ has been used, because the worst-case prediction error is of interest in this case. It is defined as

$$\rho = \max_{p=1}^{7} \left| \hat{t}(p) - t(p) \right| , \tag{3.3}$$

where $t(p)$ is the measured delay and $\hat{t}(p)$ is the predicted delay for a given p. $t(0)$ was not considered in (3.3), because the case $p = 0$ is handled in a special way by the algorithms.

Table 3.3 (rows 1–4, next page) lists the model parameters for the software versions presented in Section 3.3.1, derived from the corresponding experiments. The first interesting result is that the prediction error ρ was zero in all cases, that is, the behavior of the `arm7` implementation always followed the linear model (3.2) perfectly.

On the other hand, the same experiments performed on the more modern `cm3` architecture gave rather different results, even with the same toolchain configuration and options. They are shown in Figure 3.3 (next page) for what concerns the decoder delay, but the encoder's behavior is very similar. Although the sample variance was still zero in all cases—meaning that the architecture nevertheless behaved in a fully deterministic way—the clean, linear relationship between p and the encoding/decoding time of the `arm7` architecture was lost.

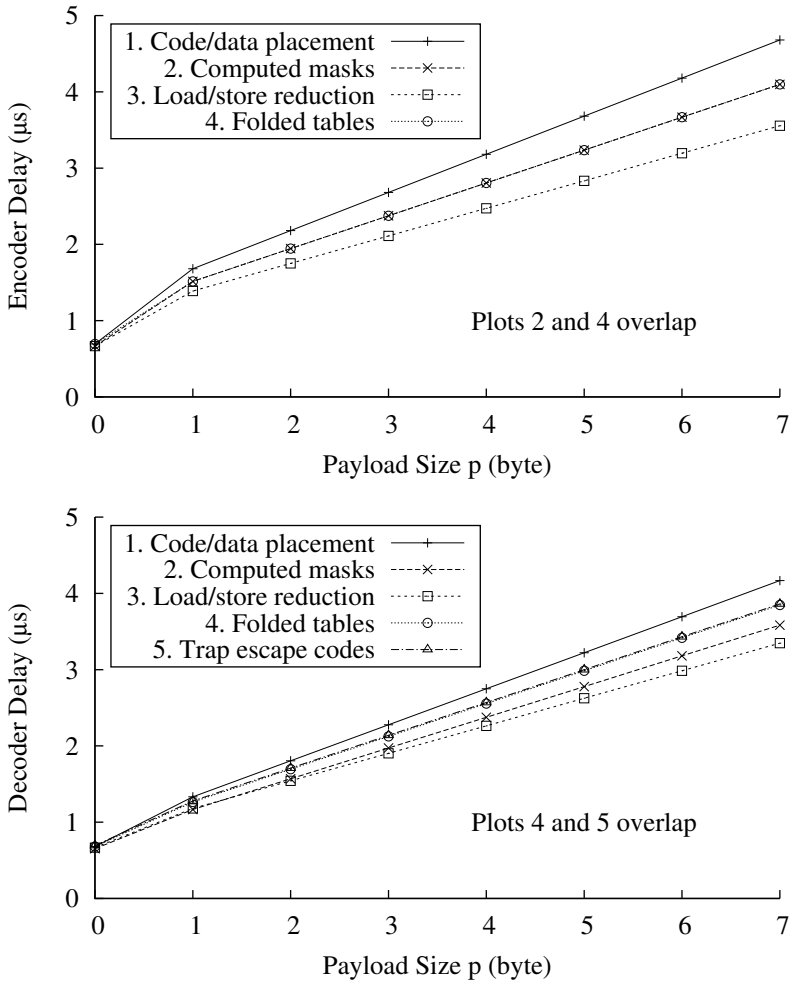

Figure 3.2. Encoder/decoder delay, `arm7` architecture.

Code version	$\rho = 0$ for all measurements					
	Encoder (μs)			Decoder (μs)		
	b	l	q	b	l	q
1. Code/data placement	0.69	0.49	0.50	0.68	0.18	0.47
2. Computed masks	0.67	0.42	0.43	0.65	0.11	0.40
3. Load/store reduction	0.67	0.36	**0.36**	0.67	0.15	**0.36**
4. Folded tables	0.69	0.39	0.43	0.68	0.15	0.43
5. Trap escape codes	—	—	—	0.69	0.15	0.43

Table 3.3. Encoder/decoder delay model, `arm7` architecture.

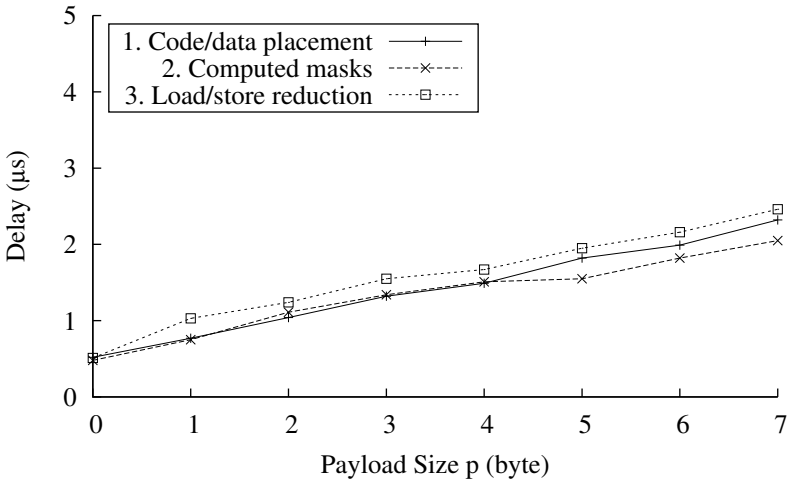

Figure 3.3. Decoder delay, `cm3` architecture, loop unrolling enabled.

After some further experimentation, the reason of the peculiar behavior was identified in a compiler optimization, known as *loop unrolling*. In this particular scenario, the compiler unrolled the encoding and decoding loops by a factor of two, and this gave rise to the up/down pattern in the delays, depending on whether p is odd or even. When this optimization was turned off linearity was almost completely restored, as shown in Figure 3.4 (plots 1–3, next page), at a small performance cost.

Interestingly enough, loop unrolling never took place on the `arm7` architecture (Figure 3.2). This difference is most probably due to the different compiler versions in use and to the architectural dissimilarities between `arm7` and `cm3`.

The linear model (3.2) was then fitted onto the new set of data, obtaining the results shown in Table 3.4 (rows 1–3, next page). Since ρ is not zero, perfect linearity was not achieved yet. However, the values of ρ also show that the worst-case prediction error of the linear delay model is less than or equal to 42 ns. This value is likely to be acceptable in most applications, both in absolute terms (less than 5 clock cycles on the `cm3` platform) and when compared to other sources of delay misprediction in the system as a whole.

It is important to remark again that a delay prediction error ρ does *not* imply that there is any uncertainty or jitter in the delay itself. On the contrary, it only means that the use of a linear delay prediction from Table 3.4 instead of the actual measured data plotted in Figure 3.4 may introduce a *systematic* prediction error less than or equal to ρ.

Regarding raw performance, it is important to remark the efficacy of the source-level code optimizations described so far. As shown in Tables 3.3 and 3.4, when combined together, they brought the encoding time from 0.50 μs to 0.36 μs (−28%) per byte on the `arm7` architecture. The improvement on the `cm3` architecture was even better, from 0.32 μs to 0.21 μs (−34%) per byte. The optimizations were less effective, but still remarkable, on the decoder: −23% on `arm7` and −18% on `cm3`. In absolute terms, the encoder and decoder loops were both reduced to 26 clock cycles per byte on `arm7` and to 21 and 23 cycles, respectively, on `cm3`.

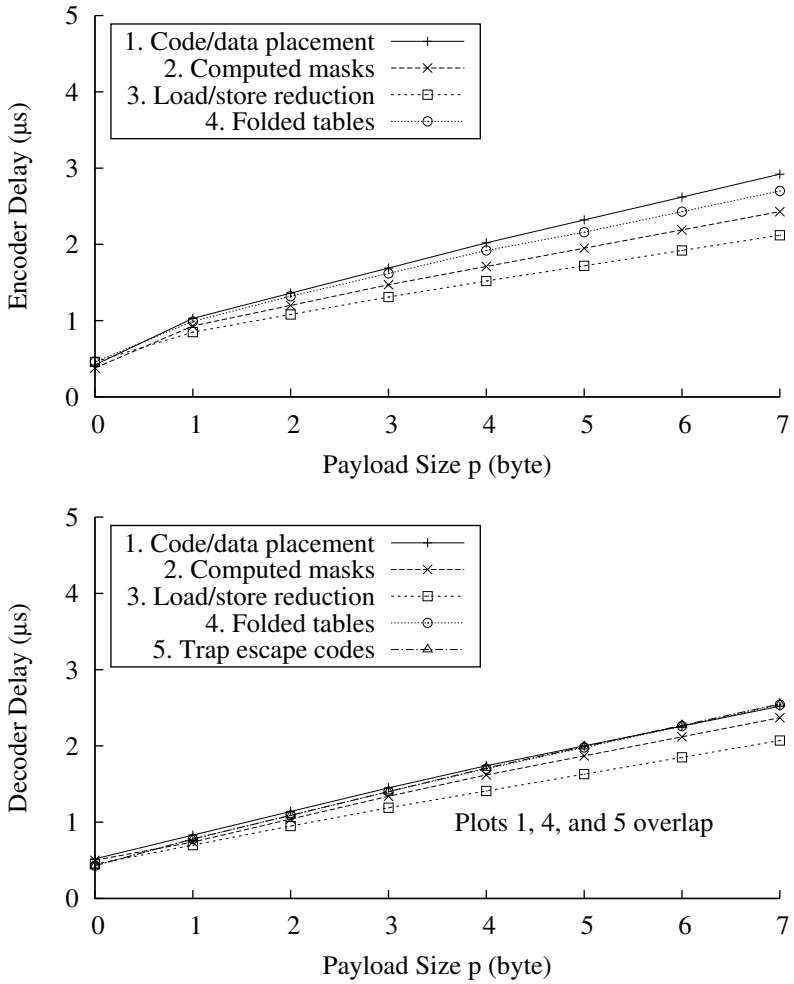

Figure 3.4. Encoder/decoder delay, `cm3` architecture, loop unrolling disabled.

	Encoder				Decoder			
Code version	b (μs)	l (μs)	q (μs)	ρ (ns)	b (μs)	l (μs)	q (μs)	ρ (ns)
1.	0.42	0.31	0.32	26	0.52	0.06	0.28	34
2.	0.38	0.32	0.25	21	0.50	0.00	0.27	34
3.	0.46	0.20	**0.21**	21	0.45	0.04	**0.23**	20
4.	0.46	0.29	0.28	**42**	0.43	0.08	0.29	21
5.	—	—	—	—	0.43	0.07	0.30	26

Table 3.4. Encoder/decoder delay model, `cm3` architecture.

Architecture and code version	Encoder/Decoder (byte) code (total)	code (loop)	table	stack
`arm7`				
1. Code/data placement	228/160	80/76	512/1024	32/28
2. Computed masks	212/144	76/72	512/1024	28/24
3. Load/store reduction	200/148	68/68	512/1024	28/24
4. Folded tables	234/164	88/88	128/256	32/28
5. Trap escape codes	—/168	—/88	—/256	—/28
`cm3`				
1. Code/data placement	164/116	64/60	512/1024	28/32
2. Computed masks	144/100	48/58	512/1024	20/28
3. Load/store reduction	136/96	46/48	512/1024	24/20
4. Folded tables	160/112	66/66	128/256	28/20
5. Trap escape codes	—/116	—/66	—/256	—/20

Table 3.5. Detailed encoder/decoder footprint for all code versions.

3.4.3 Memory Footprint/Performance Trade-Off

Table 3.5 (rows 1–3) lists the footprint of all code versions presented so far. Unlike the optimizations presented in Section 3.3.1, the further updates to the code discussed in Section 3.3.2 are not "one-way", because they entail a trade-off between footprint reduction and loss of performance/determinism.

For what concerns encoder and decoder table folding, measured performance results are shown in Figures 3.2 and 3.4 (plot 4). The corresponding linear delay model parameters can be found in the fourth row of Tables 3.3 and 3.4. In both cases, the Boolean negation operations needed to fold the table were implemented by means of the C exclusive or operator with appropriate bit masks, to avoid introducing any data dependency in the delay.

Moreover, a last version of the decoding function has been developed, which checks for invalid patterns and escape codes—discussed in Section 3.2—in the encoded byte stream. Although the raw performance of this version ought to be inferior to the previous ones by intuition, it turns out that—if the additional checks are carefully coded—the actual overhead is as small as 0.01 μs (1 clock cycle) per byte on `cm3`. The overhead on `arm7` is zero, because the compiler is able to embed the additional checks in the existing instruction stream.

As it can be seen in Table 3.5, the footprint reduction due to table folding more than offsets the extra code size.

3.5 Codec Encoding Efficiency

The data field in CAN frames—and, typically, the message payload as well—are encoded on an integral number of bytes. Hence, the *codec encoding efficiency* can be defined as the ratio between the size p of the original payload and the DLC value in the encoded frame. Only *XOR* does not cause any overhead. On the contrary, part of the data field is unavoidably wasted in all the other

Payload size p (byte)	Data field size DLC (byte)					
	XOR	***SXP***	***SXB***	***SBS***	***EEM***	***8B9B***
0	0	0	0	0	0	0
1	1	2	2	2	2	2
2	2	3	3	3	3	3
3	3	4	4	**5**	**5**	4
4	4	5	5	**6**	**6**	5
5	5	6	6	**7**	**7**	6
6	6	7	**8**	—	—	7
7	7	8	—	—	—	8
8	8	—	—	—	—	—

Table 3.6. Codec encoding efficiency.

approaches. *SXP*, *SXB*, *SBS* and *EEM* have been conceived originally for the use with the *shared-clock* (S-C) synchronization protocol, which means that some additional information (slave ID) has to be included in the frame. In order to carry out a fair comparison we assumed that only the original payload is encoded in the data field.

As said in Section 3.2, because of the translation process of the payload into 9-bit patterns, the final *8B9B*-encoded sequence is exactly one byte larger than the original payload. In the case of *SXP*, only one bit is required in order to specify whether or not the EXOR is applied to the payload; in practice, this means that one byte is wasted. Instead, in *SXB* a bit map has to be included, which specifies on a per-byte basis if EXOR was applied. BS should be prevented in this part of the frame too: if the payload is 5 bytes or less, one byte is sufficient to store the bit map, which can be encoded at the beginning of the data field as $\mathtt{S}_1\,\mathtt{M}_1\,\mathtt{M}_2\,\mathtt{S}_2\,\mathtt{M}_3\,\mathtt{M}_4\,\mathtt{M}_5\,\mathtt{S}_3$, where bits $\mathtt{S}_1$, $\mathtt{S}_2$, and $\mathtt{S}_3$ have the same function as the break bit (they are at the opposite level as the preceding bit), whereas each bit $\mathtt{M}_y$ specifies whether or not EXOR was applied to the byte in position y of the payload. If the payload is larger than 5 bytes, the bit map should take two bytes. This means, in *SXB* 6 bytes at most can fit in the payload (7 bytes are also allowed, if no countermeasures are taken to prevent stuff bits in the bit map).

Concerning *SBS*, one software stuff bit may be required every 3 original bits at worst. This case corresponds, e.g., to the sequence $\boxed{0}000\boxed{1}111\boxed{0}000\boxed{1}\ldots$ for the data field, where boxes denote software stuff bits (the first one takes into account a possible contribution due to the frame header). A padding is required in *SBS* at the end of the encoded bit sequence to make its length fixed. The size of the data field has to be selected so that the worst case depicted above can be tackled as well. Finally, in *EEM* every byte of the payload is encoded on 11 bits as $\mathtt{B}_1\,\mathtt{S}_1\,\mathtt{B}_2\,\mathtt{B}_3\,\mathtt{B}_4\,\mathtt{S}_2\,\mathtt{B}_5\,\mathtt{B}_6\,\mathtt{B}_7\,\mathtt{S}_3\,\mathtt{B}_8$, where $\mathtt{S}_1$, $\mathtt{S}_2$, and $\mathtt{S}_3$ have the same purpose as BB whereas $\mathtt{B}_k$ is the bit in position k of the considered byte.

In Table 3.6 the DLC value is shown versus the original payload size p. As can be seen, the resulting data field in *8B9B* is never longer than the other approaches, except *XOR*. Moreover, it is the only solution that is able to encode also 7-byte payloads, besides *SXP* (and *XOR*). If the contribution of stuff bits, which are possibly added to the data field by the CAN controller, is taken into account as well, *8B9B*, *SBS*, and *EEM* manage to improve, on the average, their encoding efficiency with respect to XOR-based approaches.

Summary

This chapter presented *8B9B*, an encoding scheme designed to prevent bit stuffing within the data field of CAN messages, a property especially useful in tightly synchronized systems. Although several other methods with the same purpose have been proposed in the recent past, we believe that *8B9B* can outperform them when the typical requirements of small embedded systems are taken into account. Namely, it provides a balanced blend of determinism, encoding efficiency, codec speed and footprint.

The encoding scheme is completely transparent to both the sending and receiving applications, as well as other mechanisms aimed at reducing or removing frame-level jitter. Since no assumptions are made on how the payload is used and formatted by the upper protocol layers, the proposed approach is completely general-purpose. Moreover, it is possible to apply *8B9B* only to jitter-sensitive messages, because encoded messages can coexist with plain ones. In this way, full backward compatibility is achieved. As it is usual in CAN, the distinction between the two kinds of message can easily be done through the message identifier. The only limitation is that the payload to be encoded cannot be larger than 7 bytes.

In order to show that the *8B9B* scheme is suitable for practical use, it has been implemented on two popular families of microcontrollers and then thoroughly evaluated. The results show that the code is very fast, exhibits a small memory footprint and does not introduce any processing jitter. Overall, the proposed mechanism can therefore be profitably adopted in existing projects and solutions, even if the underlying hardware platform has got a limited processing power.

Chapter 4

Performance of 8B9B versus Related Work

Bit stuffing in CAN is likely to cause jitters on message reception that, in specific cases where timing accuracy is relevant, may worsen the quality of the control algorithm noticeably. As discussed in Chapter 3, several solutions have appeared in the past years that are aimed to tackle this issue, which are based on a suitable encoding of the payload of the message carried out in software by the transmitting node.

In this chapter, two efficient approaches are considered, namely XOR and 8B9B encoding, and their performance evaluated by means of both theoretical analysis and experimental campaigns. In particular, jitter reduction capability and overall encoding efficiency (which encompasses both software codec and the BS encoder) were taken into account and compared to the case when plain CAN is adopted, that is, when no attempt is made to reduce jitters.

As already discussed in Chapter 3, applying a XOR mask to the payload of the message prior to its transmission may help to reduce (statistically) the number of stuff bits. The XOR scheme is extremely simple and maintains the same communication efficiency as plain CAN. Albeit selective XOR schemes (SXP, SXB) [65] provide reduced jitters, at the expense of a reduced codec encoding efficiency, they are still unable to completely prevent stuff bits in the payload. For these reasons, the XOR scheme has been selected for comparison.

Conversely, SBS [67], EEM [64] and 8B9B [11] are all able to ensure that no stuff bit at all is added to the data field. Since these schemes are quite similar, only 8B9B was analyzed here, because it is conceptually simpler than the others. SBS is done on a bit-by-bit basis, while for EEM 11 bits are needed to encode every 8 bit of the original payload while just 9 bits are required for 8B9B.

In order to facilitate performance comparison, the size of sections for CAN 2.0A frames is reported in Table 4.1 (next page). The same reasoning can be applied to CAN 2.0B frames as well. The overall number of bytes in the original message payload, to be stored in the CAN data field (possibly after encoding) has been denoted p, whereas the size of other fields (Message header, CRC, unstuffed portion) has been denoted in bits.

Frame section	Size (b)	Notation
Message header	19	n_{H}
Payload	$0 \ldots 64$	$8p$
CRC field	15	n_{R}
Unstuffed portion	10	n_{U}

Table 4.1. Size of the sections in a CAN frame.

4.1 Theoretical Performance Analysis

In order to carry out a comparison among the different solutions available to reduce jitters in CAN-based control systems, which depend on BS, suitable performance indices have to be defined. Two of the most interesting indices, on which we will focus in the following text, are *jitter reduction ability* and *overall encoding efficiency*, which encompasses both the software codec and the BS encoder. In order to assess these quantities, the time C taken to transmit a frame over the bus—also known as *transmission time*—is evaluated for the considered approaches.

This is not a completely trivial task: in fact, because of bit stuffing, C depends on the message identifier *id* and the actual value v embedded in the payload—besides its size p. As a consequence, in CAN such a quantity can be correctly expressed as a function $C(id, p, v)$, and can be computed exactly only when all parameters are known. Mechanisms conceived to reduce jitters, such as XOR and 8B9B, can be described as a function f that translates the original payload to the encoded payload, that is, $\langle p, v\rangle \xrightarrow{f} \langle p', v'\rangle$.

In real applications, *id* and p are seldom changed at run time for any given message stream. On the contrary, v is used to embed the instantaneous values of the signals exchanged between devices in the control system (process data). Thus, it typically varies over time and cannot be known in advance. This means, that the value of C for any given message cannot be computed once and for all. Nevertheless, it could be quite useful having some partial information on it. To this aim, C can be modeled and analyzed as a random variable.

4.1.1 Performance Indices

Since every frame is always made up of an integral number of bits, all times—and, in particular, C—will be expressed in the following as a number of bit times (t_{bit}). This makes our analysis independent of the bit rate on the network. The best way to describe C is by providing its *probability mass function (pmf)*[1]. An additional, more concise way to describe C is through typical statistical indices: average value, standard deviation, minimum and maximum, percentiles and so on. Each one of them can be used to compare the performance of different encoding schemes in specific application contexts.

- The average value $\overline{C}$ and standard deviation s_C provide a generic insight about performance. For instance, $\overline{C}$ can be used to characterize the *mean encoding efficiency*, whereas s_C gives a rough indication about *jitter reduction capability*.

[1] In probability theory and statistics, *pmf* is a function that gives the probability that a discrete random variable is exactly equal to some value.

- Percentiles constitute a very useful index for designing control systems with *soft* real-time requirements. For instance, $\mathbb{P}_{99}$ specifies the maximum transmission time experienced by 99% of the messages. For applications that are neither safety nor time-critical, where timing accuracy only affects the quality of the produced goods, such an information is certainly adequate.

- In the case of *hard* real-time systems, the maximum and minimum values, denoted $C_{\max}$ and $C_{\min}$, are the most interesting indices, along with the jitter J. For any given message stream, J represents the width of the interval over which C may range, that is, $J = C_{\max} - C_{\min}$. It is worth noting that what can be found through simulation is just an estimation of the true maximum/minimum values $C_{\max}$ and $C_{\min}$. Nonetheless, care was taken to select for the analysis a proper (and large) set of messages, so as to provide meaningful results.

4.1.2 Traffic Models

Providing figures for the above indices for a "generic" traffic is mostly pointless. In fact, when the generation law used for the values included in the payload changes, the performance indices vary accordingly. For instance, in [69, 68] a distinction was made between the traffic captured from a real network and fictitious traffic, generated artificially by a suitable simulator. What's more, as pointed out in [67], when the real application from which traffic is logged changes, also results vary accordingly. As a consequence, traffic cannot be simply classified as either real or simulated.

In the following, three different *traffic models* have been introduced for signals, which have been denoted D, A, and W. They correspond to the traffic generated by well-defined classes of devices found in factory automation environments.

- D models the traffic generated (or consumed) by *digital* I/O devices in discrete factory automation, for example, messages generated by proximity sensors and on/off commands sent to binary actuators. Usually, each signal is encoded on a single bit but, for efficiency reasons, several signals are packed together in the payload (e.g., as in the PDO mapping technique in CANopen [7]). The level (either `0` or `1`) of every signal is related to the (binary) state of a specific part of the physical system (e.g., it can specify whether or not the piece which is being manufactured is in the correct position in the machine tool). For this reason, transitions from one state to the other take some time, and the probability of being in the two states are typically not the same. As a consequence, the likelihood of having patterns of multiple bits at the same value across the payload is not negligible.

  Every bit of the payload is modelled according to a discrete-time, Markov chain. Its two states represent the level of the binary signal, either `0` or `1`. This generation law can be tuned by means of two parameters, $\chi_{0\to1}$ and $\chi_{1\to0}$, which represent the transition probability from `0` to `1` and vice versa. For the experiments, the values $\chi_{0\to1} = 1/256$ and $\chi_{1\to0} = 16/256$ have been chosen, corresponding to the steady-state probabilities of having a bit at `0` or `1` of $\pi_0 = \chi_{1\to0}/(\chi_{0\to1} + \chi_{1\to0}) \simeq 0.94$ and $\pi_1 = \chi_{0\to1}/(\chi_{0\to1} + \chi_{1\to0}) \simeq 0.06$, respectively. For every bit, the number of consecutive samples at the same level is geometrically distributed with an average of $\mu_0 = (1 - \chi_{0\to1})/\chi_{0\to1} = 255$ for `0` and $\mu_1 = (1 - \chi_{1\to0})/\chi_{1\to0} = 15$ for `1`.

- Model A represents traffic generated (or consumed) by *analog* I/O devices, including encoders, in continuous factory automation (process automation, motion control applications, etc.). Every analog signal is encoded on a given number of bits (16 bits is a typical size) and, again, more than one signal can be packed in one message payload. As the value encoded in any one of such signals is related to a physical quantity, it is unlikely that it may change abruptly. This means that there is a high likelihood that the values encoded in subsequent messages will differ by just a small amount. As a consequence, most of the values that fit into a given range will be generated.

  In the experiments, up to 4 16-bit counters are packed in the payload. They are incremented by one when a new message is generated and wrap around to zero after reaching their maximum value. If the size of the payload is an odd number of bytes, the least significant byte of the last counter is discarded. We tried several generation laws for traffic model A, and we found that the one just described is able to provide minimum/maximum values for C which are closer to the theoretical bound.

- The goal of the last model, W, is to deal with all the cases that cannot be reasonably described as traffic generated by either digital or analog devices. For instance, the transmission of the fragments of a bigger data block in a segmented transfer, or a list of command OP codes in a part program, may fall under this case. Here, the generation law is assumed to be completely arbitrary, that is, every byte of the payload is a uniformly-distributed random value, so that the way the payload state space is explored resembles the classic Monte Carlo methods [50] closely. This is clearly a quite generic assumption about the traffic. Nevertheless, this traffic model can be useful to analyze those streams in which no clear correlation can be found (or, better, is known) between subsequent messages.

It is worth remarking that many other, more specific generation laws can be defined, besides the three ones listed above. Nonetheless, we believe that they are able to provide a satisfactory taxonomy of the main kinds of traffic in real-world industrial applications for the purpose of our performance analysis.

4.1.3 Bounds on Transmission Times

Statistical indices depend on the number of samples taken into account in the evaluation campaign. For this reason, they must be considered just as estimations. This is particularly true in the case of the minimum and maximum values, obtained experimentally. In the following, they will be denoted using the "hat" modifier, i.e., $\widehat{C}_{\text{min}}$ and $\widehat{C}_{\text{max}}$, respectively.

In order to assess the reliability of such estimations, the minimum and maximum values have been compared to the related theoretical lower and upper bounds, respectively. Such bounds are denoted using the "tilde" modifier, i.e., $\widetilde{C}_{\text{min}}$ and $\widetilde{C}_{\text{max}}$. In the best case, no stuff bits are, in theory, added. This means, that for CAN messages with standard identifier, $\widetilde{C}_{\text{min}}$ can be computed as

$$\widetilde{C}_{\text{min}}^{\text{CAN}} = n_{\text{H}} + 8p + n_{\text{R}} + n_{\text{U}} = 44 + 8p \ . \tag{4.1}$$

As pointed out in [68], the worst-case sequence concerning BS is made up of 5 bits at `0`, followed by an alternating pattern made up of 4 bits at `1` and 4 bits at `0`. Only the part of the

frame from the SOF bit up to the CRC field (included) is encoded through the BS rules. As a consequence, the maximum number $h_{\max}$ of stuff bits that can be added to the frame is

$$h_{\max} = \left\lfloor \frac{n_{\mathrm{H}} + 8p + n_{\mathrm{R}} - 1}{4} \right\rfloor = 8 + 2p \ . \tag{4.2}$$

This leads to a value for $\widetilde{C}_{\max}$ equal to

$$\widetilde{C}_{\max}^{\mathrm{CAN}} = \widetilde{C}_{\min}^{\mathrm{CAN}} + h_{\max} = 52 + 10p \ . \tag{4.3}$$

Studies on CAN revealed that this worst-case never takes place, because some parts of the header are fixed (SOF and reserved bits) whereas others are not uncorrelated from the payload (DLC). Indeed, the actual worst-case transmission time $C_{\max}$ is always one or two bits shorter than $\widetilde{C}_{\max}$.

The above bounds hold for plain CAN, as well as for the XOR approach. In the case of 8B9B, no stuff bits at all are added to the data field [11]. However, because of the specific encoding scheme, the size of the data field is always one byte higher than the original payload (i.e., DLC = $p + 1$).

The exact number h_{R} of stuff bits added to the CRC cannot be known in advance. Nevertheless, since the CRC is 15-bit long and, in 8B9B, it may be preceded (at worst) by two bits at the same level [11], no more than 4 stuff bits may be added ($h_{\mathrm{R}} \leq 4$). Instead, the number h_{H} of stuff bits added to the message header can be evaluated exactly, if *id* and p are known. Generally speaking, as the header consists of 19 bits, no more than 4 stuff bits can be inserted here ($h_{\mathrm{H}} \leq 4$). As a consequence

$$\widetilde{C}_{\min}^{\mathrm{8B9B}} = n_{\mathrm{H}} + 8(p+1) + n_{\mathrm{R}} + n_{\mathrm{U}} = 52 + 8p \ , \tag{4.4}$$

$$\widetilde{C}_{\max}^{\mathrm{8B9B}} = \widetilde{C}_{min}^{\mathrm{8B9B}} + \max(h_{\mathrm{H}}) + \max(h_{\mathrm{R}}) = 60 + 8p \ . \tag{4.5}$$

As can be seen, when $p > 4$ the worst-case transmission time in 8B9B is lower than CAN. When messages with a 7-byte payload are considered, $\widetilde{C}_{\max}$ decreases from 122 to 116 bit times. This implies, that also in the case arbitration is exploited (e.g., in the automotive context), 8B9B could be profitably adopted in order to reduce the frame transmission times used in schedulability analysis [19].

In typical real CAN-based applications, *id* and p never change at run time for any given message stream. For this reason, in the following experimental campaigns, one specific identifier was chosen. Generally speaking, transmission times depend on the particular identifier of the message which is being exchanged. In order to keep notation as simple as possible, the identifier will not be included explicitly. For example, $\widehat{C}_{\max}$ and $\widehat{C}_{\min}$ will be used in the place of $\widehat{C}_{\max}(id)$ and $\widehat{C}_{\min}(id)$, respectively. Conversely, the above expressions for $\widetilde{C}_{\min}$ and $\widetilde{C}_{\max}$ do not depend on it. They can be profitably replaced by other expressions, which take into account the exact number of stuff bits added to the header and provide tighter bounds for $C_{\max}$ and $C_{\min}$. For CAN (and XOR) they are

$$\widetilde{C}_{\min}^{\mathrm{CAN}}(id) = n_{\mathrm{H}} + h_{\mathrm{H}} + 8p + n_{\mathrm{R}} + n_{\mathrm{U}} = 44 + h_{\mathrm{H}} + 8p \ , \tag{4.6}$$

$$\widetilde{C}_{\max}^{\mathrm{CAN}}(id) = \widetilde{C}_{\min}^{\mathrm{CAN}}(id) + \left\lfloor \frac{w_{eq} + 8p + n_{\mathrm{R}} - 1}{4} \right\rfloor = 47 + h_{\mathrm{H}} + \left\lfloor \frac{w_{eq} + 2}{4} \right\rfloor + 10p \ . \tag{4.7}$$

where w_{eq} is the number of bits at the same level that are found at the end of the header (including the related stuff bits). Because of BS rules, $1 \leq w_{eq} \leq 4$. In the case of 8B9B, instead, we get

$$\widetilde{C}_{\min}^{\text{8B9B}}(id) = n_{\text{H}} + h_{\text{H}} + 8(p+1) + n_{\text{R}} + n_{\text{U}} = 52 + h_{\text{H}} + 8p \ , \tag{4.8}$$

$$\widetilde{C}_{\max}^{\text{8B9B}}(id) = \widetilde{C}_{\min}^{\text{8B9B}}(id) + \max(h_{\text{R}}) = 56 + h_{\text{H}} + 8p \ . \tag{4.9}$$

By using the expressions above, the theoretical maximum jitter can be evaluated as

$$\widetilde{J}^{\text{CAN}}(id) = 3 + \left\lfloor \frac{w_{eq} + 2}{4} \right\rfloor + 2p \tag{4.10}$$

for plain CAN and XOR, whereas for 8B9B it is

$$\widetilde{J}^{\text{8B9B}}(id) = 4 \ . \tag{4.11}$$

It is worth noting that the actual values of $C_{\min}$ and $C_{\max}$ for any given message stream can be found only via exhaustive exploration of the space of the possible payloads, which is usually unfeasible. Nevertheless, from the above considerations, it follows that the properties below hold for the involved quantities

$$\widehat{C}_{\max} \leq C_{\max}(id) \leq \widetilde{C}_{\max}(id) \leq \widetilde{C}_{\max} \ , \tag{4.12}$$

$$\widehat{C}_{\min} \geq C_{\min}(id) \geq \widetilde{C}_{\min}(id) \geq \widetilde{C}_{\min} \ . \tag{4.13}$$

By means of these expressions, an accurate estimation of $C_{\min}$ and $C_{\max}$ is possible.

4.2 Experimental Evaluation and Comparison

A simulation campaign was carried out for evaluating the performance indices concerning C for the 3 cases where plain CAN, XOR and 8B9B are adopted to encode the payload of messages. For every approach, 3 distinct sets of experiments were carried out for the traffic models D, A, and W, defined in Section 4.1.2.

The notation Gp_S is used in the following to denote experiment configurations, where $G \in \{D, A, W\}$ is the generation law for the payload, p is the payload size in bytes, and $S \in \{\text{CAN}, \text{XOR}, \text{8B9B}\}$ represents the encoding scheme. For instance, $D4_{\text{XOR}}$ means that the payload embeds process data that resemble those produced by digital I/Os, whose overall size is 4 bytes and that are encoded through the XOR scheme. In the experiments, p varied from 1 to the maximum allowed size (8 bytes for CAN and XOR, 7 bytes for 8B9B), and a single $id = 2\text{AA}_{16}$ was considered, since its contribution to bit stuffing is predictable and fixed. Data frames with no payload, as well as remote frames, were not considered here as the related transmission time can be evaluated right in the design phase.

Experiments were accomplished through a simulator, which generates a stream of messages m_i according to the chosen traffic model, computes the transmission times C_i, collects their frequency distribution, and calculates the performance indices defined in Section 4.1.1.

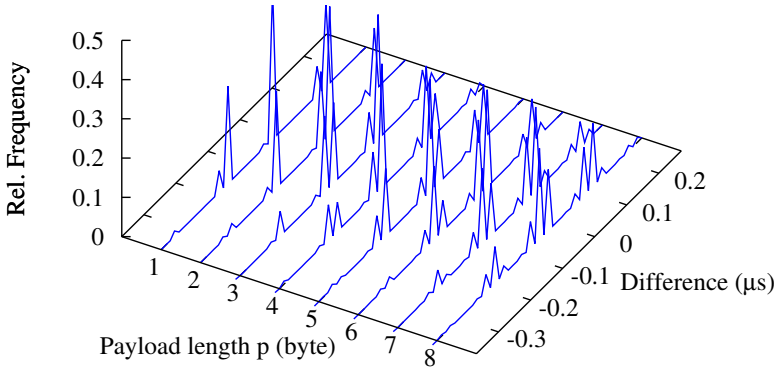

Figure 4.1. Simulated versus measured C_i.

4.2.1 Validation of the Simulator

The most complex part of the simulation software is the module that, given a CAN message identifier *id* and a payload of size p, formats the message m_i, calculates the message CRC, determines the bit stream actually sent on the bus, including stuff bits, and then computes its transmission time C_i. To validate the module, the simulated C_i was compared with the actual transmission time obtained with a real CAN controller, namely, the one embedded in the NXP LPC2468 microcontroller, running at 500 kbps in self-receive mode. The measurement was taken in the Wp_{CAN} case, with $p = 1, \ldots, 8$, collecting 10^7 samples for each value of p. Figure 4.1 shows the frequency distribution (z axis) of the difference (y axis), as a function of p (x axis). It can easily be seen that the difference is within $\pm 0.3\,\mu s$, that is, one order of magnitude below the CAN controller bit time ($2\,\mu$s). This result provides strong evidence that the simulated behavior conforms to reality, whereas the difference is caused by measurement noise and jitter at the interface between the CPU and the CAN controller itself.

4.2.2 Bit Stuffing in Plain CAN

The first set of experiments had the goal of investigating the bit stuffing behavior of plain CAN for different traffic models, in order to establish a reference baseline for the evaluation of the bit stuffing reduction techniques. Hence, it involved the Dp_{CAN}, Ap_{CAN}, and Wp_{CAN} configurations for $p = 1, \ldots, 8$. For every configuration, 10^8 messages were exchanged, for statistical significance.

Figure 4.2 (next page) shows the probability mass functions obtained from the experiments. Namely, it shows the relative frequency (z axis) of the message transmission time C (y axis) depending on the payload length p (x axis).

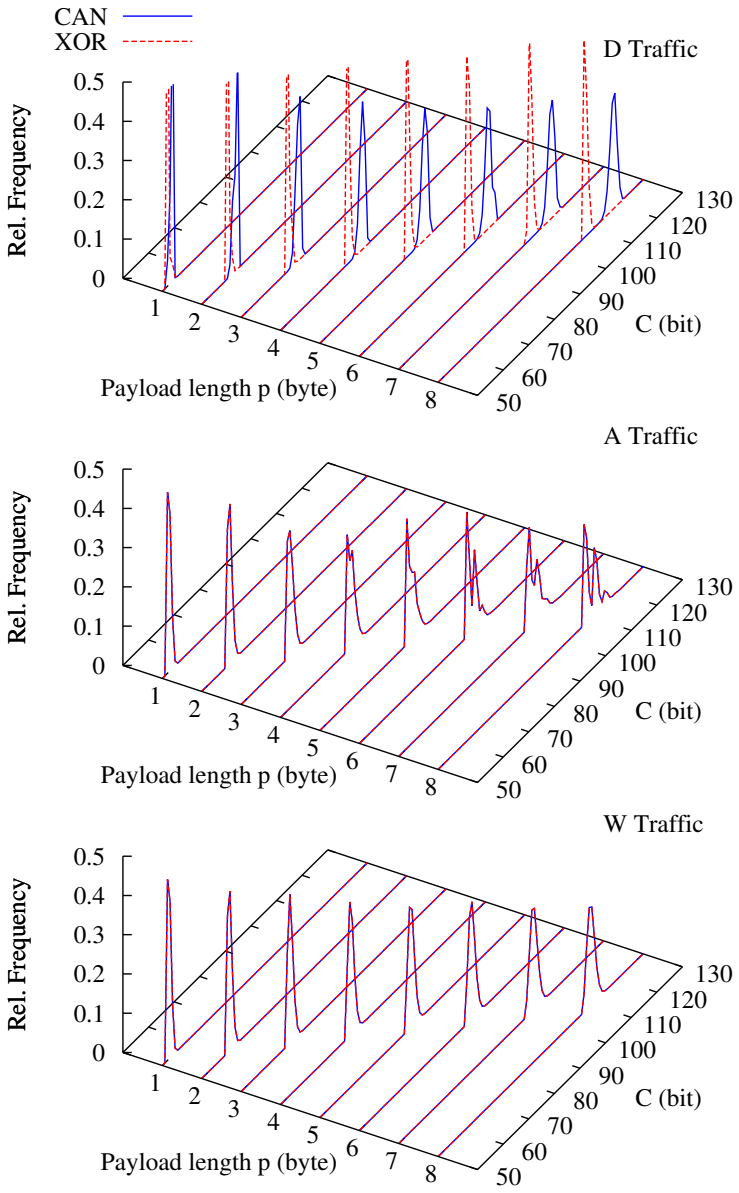

Figure 4.2. Pmf of plain CAN versus XOR.

Experiment	$\overline{C}$	s_C	$\mathbb{P}_{99}$	$\widehat{C}_{\min}$	$\widehat{C}_{\max}$	$\widehat{J}$
$D1_{\mathrm{CAN}}$	54.6	0.60	55	53	56	3
$D2_{\mathrm{CAN}}$	64.3	0.88	65	61	66	5
$D3_{\mathrm{CAN}}$	73.2	0.89	75	69	76	7
$D4_{\mathrm{CAN}}$	82.8	1.00	85	77	86	9
$D5_{\mathrm{CAN}}$	91.9	1.11	94	85	95	10
$D6_{\mathrm{CAN}}$	101.4	1.29	104	94	106	12
$D7_{\mathrm{CAN}}$	110.6	1.28	113	103	115	12
$D8_{\mathrm{CAN}}$	**119.4**	1.33	122	111	124	13
$A1_{\mathrm{CAN}}$	53.7	0.74	56	53	56	3
$A2_{\mathrm{CAN}}$	62.0	0.89	64	61	66	5
$A3_{\mathrm{CAN}}$	70.3	1.16	73	69	76	7
$A4_{\mathrm{CAN}}$	78.6	1.41	82	77	85	8
$A5_{\mathrm{CAN}}$	86.8	1.70	91	85	96	11
$A6_{\mathrm{CAN}}$	95.0	1.99	100	93	106	13
$A7_{\mathrm{CAN}}$	103.8	2.33	110	101	116	15
$A8_{\mathrm{CAN}}$	110.8	2.65	118	108	126	**18**
$W1_{\mathrm{CAN}}$	53.7	0.74	56	53	56	3
$W2_{\mathrm{CAN}}$	62.0	0.89	64	61	66	5
$W3_{\mathrm{CAN}}$	70.3	1.03	73	69	77	8
$W4_{\mathrm{CAN}}$	78.6	1.13	82	77	87	10
$W5_{\mathrm{CAN}}$	86.8	1.20	90	85	96	11
$W6_{\mathrm{CAN}}$	95.0	1.29	98	93	105	12
$W7_{\mathrm{CAN}}$	103.8	1.45	107	101	113	12
$W8_{\mathrm{CAN}}$	110.8	1.49	115	108	121	**13**
$D1_{\mathrm{XOR}}$	53.2	0.51	55	53	56	3
$D2_{\mathrm{XOR}}$	61.2	0.52	63	61	66	5
$D3_{\mathrm{XOR}}$	69.4	0.69	71	69	75	6
$D4_{\mathrm{XOR}}$	77.4	0.63	79	77	83	6
$D5_{\mathrm{XOR}}$	85.5	0.66	87	85	91	6
$D6_{\mathrm{XOR}}$	93.6	0.69	95	93	99	6
$D7_{\mathrm{XOR}}$	102.5	0.74	105	101	109	8
$D8_{\mathrm{XOR}}$	**108.7**	0.76	111	108	115	7
$A1_{\mathrm{XOR}}$	53.7	0.74	56	53	56	3
$A2_{\mathrm{XOR}}$	62.0	0.89	64	61	66	5
$A3_{\mathrm{XOR}}$	70.3	1.16	73	69	76	7
$A4_{\mathrm{XOR}}$	78.6	1.41	82	77	85	8
$A5_{\mathrm{XOR}}$	86.8	1.70	91	85	96	9
$A6_{\mathrm{XOR}}$	95.0	1.99	100	93	106	13
$A7_{\mathrm{XOR}}$	103.8	2.33	110	101	116	15
$A8_{\mathrm{XOR}}$	110.8	2.65	118	108	126	18
$W1_{\mathrm{XOR}}$	53.7	0.74	56	53	56	3
$W2_{\mathrm{XOR}}$	62.0	0.89	64	61	66	5
$W3_{\mathrm{XOR}}$	70.3	1.03	73	69	77	8
$W4_{\mathrm{XOR}}$	78.6	1.13	82	77	87	10
$W5_{\mathrm{XOR}}$	86.8	1.20	90	85	96	9
$W6_{\mathrm{XOR}}$	95.0	1.29	98	93	104	11
$W7_{\mathrm{XOR}}$	103.8	1.45	107	101	114	13
$W8_{\mathrm{XOR}}$	110.8	1.49	115	108	121	13

Table 4.2. Performance indices of CAN and XOR.

The other performance indices discussed in Section 4.1.1 are listed in the upper part of Table 4.2 (previous page). In this table, and in the following ones, the 99th percentile of the (discrete) distribution of C has been defined as

$$\mathbb{P}_{99}(C) = x \quad \Leftrightarrow \quad \begin{cases} \Pr[C \leq x] \geq 0.99 \\ \Pr[C < x] < 0.99 \end{cases} , \tag{4.14}$$

while $\widehat{J}$ denotes the maximum measured jitter for each experiment

$$\widehat{J} = \widehat{C}_{\max} - \widehat{C}_{\min} \ . \tag{4.15}$$

A very important conclusion that can be drawn from the results is that the traffic model affects in a very noticeable way the *pmf* of C, as well as the other performance indices, as different subsets of the payload state space are explored. In particular, the more realistic traffic models D and A lead to results that are drastically different with respect to random traffic model W. This is particularly evident, for instance, from the values of the measured worst-case jitter across all possible payload lengths p, defined as

$$\widehat{J}_{\max} = \max_{p}(\widehat{J}) \ , \tag{4.16}$$

which is 18 b for $A8_{\text{CAN}}$, but only 13 b for $W8_{\text{CAN}}$. Even more importantly, this happens even if the mean transmission time $\overline{C}$ is practically the same for the two experiment configurations. Moreover, traffic models A and D drive the simulation closer to the theoretical maximum transmission time $\widehat{C}_{\max}^{\text{CAN}}(2\text{AA}_{16})$, with respect to traffic model W. Overall, these results highlight that classic Monte Carlo methods may be inadequate to thoroughly analyze CAN bit stuffing, even when a relatively large number of samples (10^8 in this case) is used.

4.2.3 Bit Stuffing in XOR

The same set of experiments described in Section 4.2.2 was repeated for the Dp_{XOR}, Ap_{XOR}, and Wp_{XOR} configurations with $p = 1, \ldots, 8$. The results are shown in Figure 4.2 and in the lower part of Table 4.2. They provide conflicting information about the bit stuffing reduction capability of XOR:

- On the one hand, for traffic model D, XOR is extremely effective to reduce the mean transmission time and bring, for instance, $\overline{C}$ from 119.4 b ($D8_{\text{CAN}}$) to 108.7 b ($D8_{\text{XOR}}$). At the same time, as it can be seen in the upper part of Figure 4.2 and looking at the $\widehat{J}$ values in Table 4.2, the pmf of C with XOR is also narrower. Namely, the measured jitter $\widehat{J}$ varies from a minimum of 3 b ($D1_{\text{XOR}}$) to a maximum of 8 b ($D7_{\text{XOR}}$), giving $\widehat{J}_{\max} = 8$ b in this case.

- On the other hand, as it is also remarked in [65], the effectiveness of XOR heavily depends on the statistical properties of the traffic. In fact, the *pmf* of CAN and XOR completely overlap for traffic models A and W, as it can be seen in the middle and lower part of Figure 4.2. For those kinds of traffic, the adoption of XOR does *not* bring any real advantage.

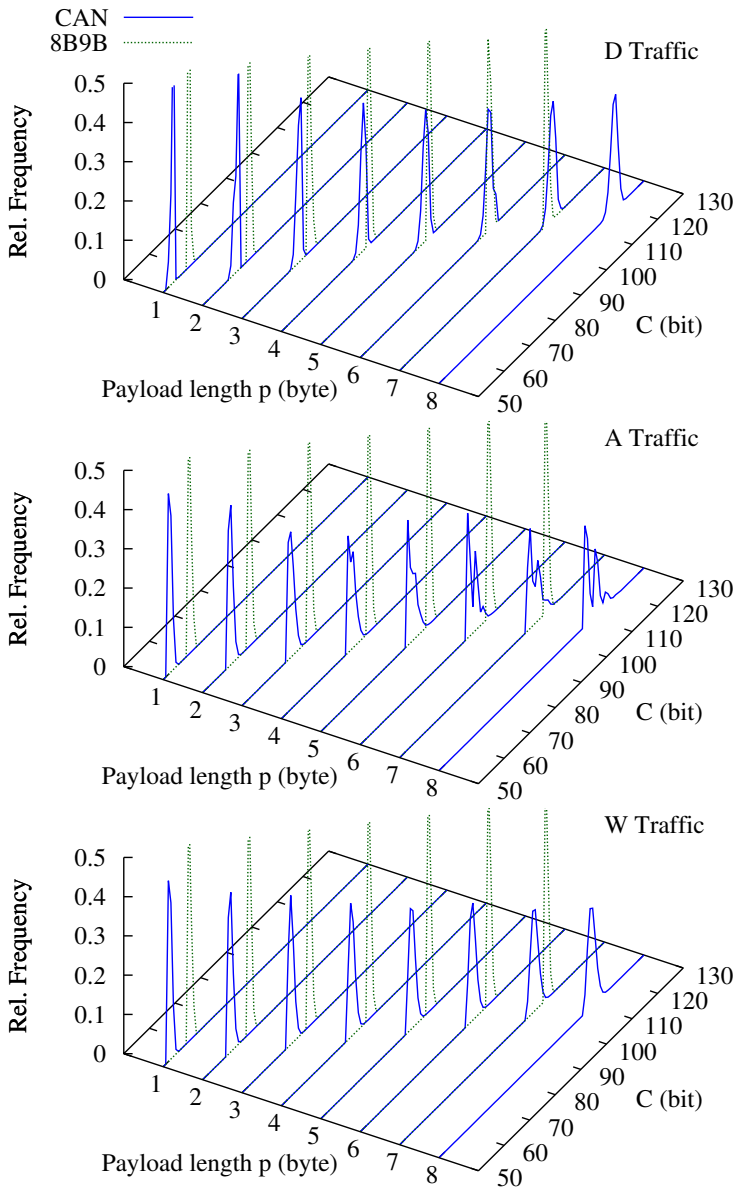

Figure 4.3. Pmf of plain CAN versus 8B9B.

4.2.4 Bit Stuffing in 8B9B

Figures 4.3 and 4.4 (next page) show the results obtained with experiment configurations Dp_{8B9B}, Ap_{8B9B}, and Wp_{8B9B}. For these experiments, the payload length was kept in the range $p = 1,\ldots,7$ because 8B9B cannot encode 8-byte payloads. The same figures also compare 8B9B with plain CAN and XOR, respectively. Figure 4.3 has the same layout as Figure 4.2. In Figure 4.4, only the *pmf* for D traffic is shown, because the behavior of CAN and XOR is exactly the same for the other two kinds of traffic. Table 4.3 (next page) contains the related 8B9B performance indices.

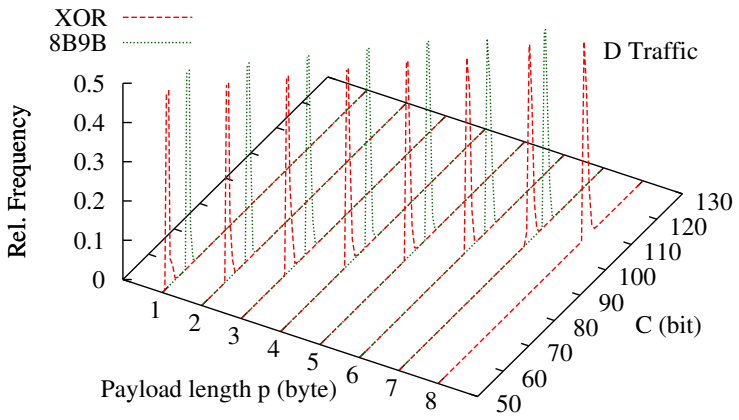

Figure 4.4. Pmf of 8B9B versus XOR, D traffic.

Experiment	$\overline{C}$	s_C	$\mathbb{P}_{99}$	$\widehat{C}_{\min}$	$\widehat{C}_{\max}$	$\widehat{J}$
$D1_{8B9B}$	61.1	0.34	62	61	63	2
$D2_{8B9B}$	69.3	0.49	71	69	72	3
$D3_{8B9B}$	77.4	0.63	79	77	81	4
$D4_{8B9B}$	85.4	0.54	87	85	88	3
$D5_{8B9B}$	93.5	0.59	95	93	96	3
$D6_{8B9B}$	101.7	0.73	103	101	105	4
$D7_{8B9B}$	108.5	0.67	110	108	112	4
$D8_{8B9B}$	—	—	—	—	—	—
$A1_{8B9B}$	61.5	0.64	63	61	63	2
$A2_{8B9B}$	69.4	0.59	71	69	72	3
$A3_{8B9B}$	77.5	0.64	79	77	80	3
$A4_{8B9B}$	85.4	0.59	87	85	88	3
$A5_{8B9B}$	93.4	0.59	95	93	96	3
$A6_{8B9B}$	101.5	0.63	103	101	105	4
$A7_{8B9B}$	108.4	0.61	110	108	111	3
$A8_{8B9B}$	—	—	—	—	—	—
$W1_{8B9B}$	61.5	0.64	63	61	63	2
$W2_{8B9B}$	69.4	0.59	71	69	72	3
$W3_{8B9B}$	77.5	0.64	79	77	81	4
$W4_{8B9B}$	85.4	0.59	87	85	88	3
$W5_{8B9B}$	93.4	0.59	95	93	96	3
$W6_{8B9B}$	101.5	0.63	103	101	105	4
$W7_{8B9B}$	108.4	0.61	110	108	112	4
$W8_{8B9B}$	—	—	—	—	—	—

Table 4.3. Performance indices of 8B9B.

From Figure 4.3, it can be seen that the behavior of 8B9B is completely consistent across the traffic models. Namely, also referring to Table 4.3, the performance indices are almost exactly the same. This is also true for the maximum jitter $\widehat{J}_{\max}$ that, in agreement with (4.11), is always $\widehat{J}_{\max} \leq \widetilde{J}^{8B9B}(id) = 4$, regardless of the traffic model.

	CAN/XOR			8B9B		
p	$\widehat{C}_{\max}$	$\widetilde{C}_{\max}(id)$	$\widetilde{C}_{\max}$	$\widehat{C}_{\max}$	$\widetilde{C}_{\max}(id)$	$\widetilde{C}_{\max}$
1	56	58	62	63	65	68
2	66	68	72	72	73	76
3	77	79	82	81	81	84
4	87	89	92	88	89	92
5	96	98	102	96	97	100
6	106	108	112	105	105	108
7	116	119	122	112	112	116
8	126	128	132	—	—	—

Table 4.4. Upper bounds on C ($id = 2\text{AA}_{16}$).

In absolute terms, the maximum amount of jitter of 8B9B is also strictly lower than the jitter of plain CAN and XOR in all cases.

For what concerns the maximum transmission time $C_{\max}$, in spite of its lower encoding efficiency, 8B9B performs better than plain CAN—regardless of the traffic model—for $p \geq 5$. The same is true with respect to XOR for traffic models A and W. On the contrary, traffic model D, which is of practical relevance, highlights that XOR can be quite effective. For this last traffic model, 8B9B encoding is not any better than XOR for any payload size.

Finally, in order to further verify the correctness of the simulation, the maximum transmission times $\widehat{C}_{\max}$—evaluated for a specific id over all the kinds of traffic—have been checked against the theoretical upper bounds described in Section 4.1.3, namely $\widetilde{C}_{\max}(id)$ and $\widetilde{C}_{\max}$. Minimum values have not been taken into account here, as they are less interesting. The related figures are reported in Table 4.4. As can be seen, property (4.12) always holds.

Summary

In this chapter, two schemes for reducing bit stuffing jitter in CAN-based control systems have been compared, i.e., XOR and 8B9B encoding. Their performance has been evaluated by means of a thorough statistical analysis carried out through numerical simulation, and results have been checked against the theoretical bounds on transmission times. Then, a comparison has been made among them that includes plain CAN as well.

As expected, irrespective of the traffic on the network, 8B9B is always able to achieve a noticeable reduction of jitters (both $\widehat{J}$ and s_C) with respect to both CAN and XOR. Concerning the worst-case transmission times, when the payload size is large, 8B9B is able to outperform CAN and, in several cases, XOR. An exception is given by the specific case of digital traffic, where long sequences of bits at the same value are likely to be found in the payload. In this case, XOR proved to be able to achieve the best performance for worst-case transmission times.

However, it should be noted that, unlike 8B9B, XOR is unable to provide any guarantee on its performance for *every* kind of traffic. This is because, in general, XOR is able to offer a "statistical" reduction of jitters, but this does not necessarily reduce the *worst-case* jitter. In turn, this limits its applicability in hard real-time contexts. Nevertheless, its implementation is extremely simple

and the encoding efficiency is as good as plain CAN. Therefore, it can be actually valuable in *soft* real-time control systems.

Results show that, despite encoding efficiency for 8B9B is unavoidably lower than CAN (and XOR), its higher degree of determinism can be profitably exploited in several scenarios. Besides offering tangible advantages when dealing with accurate synchronization of devices in hard real-time systems that rely on CAN for communication, it can be of help also in the automotive scenario—where nodes are not synchronized and arbitration is exploited to solve contentions on the bus. In the last case, in fact, schedulability analysis can be improved through the reduction of worst-case transmission times.

Chapter 5

Variable Length Payload Encoding

In Chapter 3 an encoding technique, namely 8B9B, which is able to completely prevent stuff bits in the data field of CAN frames, was described. The basic principle behind 8B9B is quite simple. Every byte in the original payload of the CAN message is translated separately to a codeword expressed as a 9-bit pattern. Two additional fields were also introduced in 8B9B, namely the *break bit* (BB) and the *padding field* (PAD). BB is located in the very first position of the data field, and is set at the opposite value than the least significant bit of the DLC. It prevents the occurrence of a primer sequence on the boundary between the DLC and data fields. The padding field, instead, is just a particular filling of the unused portion of the last byte in the encoded data field that, again, is aimed at preventing primer sequences.

Although the basic 8B9B mechanism is simple enough to allow for fast implementations, and intuitively it is able to achieve very good encoding efficiency, there is not any warranty that it is the best possible choice. In this chapter, this problem has been modeled formally using an inductive approach and it is proved that 8B9B is actually the optimal one in its class.

Moreover, analysis has also led to a new version of the codebook, which satisfies an interesting nesting property. Based on this, a new enhanced scheme, called Variable-length High-performance Code for CAN (VHCC), which permits to pack additional sub-byte information in the encoded data field of CAN frames, has also been introduced.

5.1 Codebook Construction

5.1.1 Requirements

Let us denote with the calligraphic letter $\mathcal{U}_s$ the universe of binary strings of length s and with the corresponding uppercase letter U_s the number of elements of $\mathcal{U}_s$, that is, let $U_s = |\mathcal{U}_s|$. An analogous notation will be used for the other sets defined throughout this section. Obviously, it is:

$$U_s = 2^s \quad . \tag{5.1}$$

Our goal is to build a set of codebooks $\mathcal{G}_{s,l,b,t}$ for integer values $s \geq 2$, $l \geq 1$, $b \geq 1$, and $t \geq 1$, so that each codebook satisfies the following properties:

- the elements of $\mathcal{G}_{s,l,b,t}$ are binary strings of length s, that is, $\mathcal{G}_{s,l,b,t} \subseteq \mathcal{U}_s$;

- all elements $g \in \mathcal{G}_{s,l,b,t}$ have at most l leading bits at the same value, at most b consecutive inner bits at the same value, and at most t trailing bits at the same value;
- the elements of $\mathcal{G}_{s,l,b,t}$ contain at least one bit value transition, that is,

$$\overbrace{0\ldots0_2}^{s \text{ bits}} \notin \mathcal{G}_{s,l,b,t}, \quad \overbrace{1\ldots1_2}^{s \text{ bits}} \notin \mathcal{G}_{s,l,b,t} \ ,$$

  this condition is included in the previous one when $s > \min(l, t)$;
- $\mathcal{G}_{s,l,b,t}$ contains enough codewords to uniquely encode a binary string of length $s-1$, that is, denoting with $G_{s,l,b,t}$ the number of elements of $\mathcal{G}_{s,l,b,t}$,

$$G_{s,l,b,t} \geq U_{s-1} \ . \tag{5.2}$$

When all these conditions are satisfied—and provided that $l + t \leq b$—it is trivial to prove that the bit string resulting from the concatenation of any number of elements of $\mathcal{G}_{s,l,b,t}$, even for varying s, will contain *no more than* b consecutive bits at the same value at any position, as long as l, b, and t are still kept the same for all codebooks. If the communication channel bit stuffing mechanism is triggered by a sequence of $b + 1$ bits at the same value (namely, the primer sequences), this is a *sufficient* condition for the complete absence of bit stuffing at the channel level. As a result, it is $b = 4$ for CAN.

Other points of interest are to determine the maximum value of s, denoted with s_m in the following, for which codebook construction is still possible and, in particular, property (5.2) still holds. For an efficient practical implementation of the encoding and decoding software modules based on lookup tables, which will be discussed in Section 5.2, it is also important to highlight any relationship between the codebooks $\mathcal{G}_{s,l,b,t}$, for varying $s = 2, \ldots, s_m$. This is because any relationship can be leveraged to reduce the size of the lookup tables, and hence, the footprint of the software.

5.1.2 Leading Bits

In the following, for any $s \geq 1$, $\mathcal{Q}_s$ denotes the set of strings of length s with exactly s leading (and trailing) bits at the same value. Moreover, for any $s \geq 2$ and $1 \leq k \leq s-1$, $\mathcal{V}_s(k)$ denotes the set of strings of length s with no more than $s-k$ leading bits at the same value *and* exactly k trailing bits at the same value.

A direct consequence of these definitions is that all the elements of $\mathcal{V}_s(k)$, regardless of the value of k, always have strictly less than s leading bits at the same value. Moreover, for any value of s, the set $\mathcal{Q}_s$ contains exactly two binary strings composed of either all zeros or all ones, namely

$$\mathcal{Q}_s = \{\overbrace{0\ldots0_2}^{s \text{ bits}}, \ \overbrace{1\ldots1_2}^{s \text{ bits}}\} \ \forall s \geq 1 \ . \tag{5.3}$$

Denoting with Q_s the number of elements of $\mathcal{Q}_s$, it is therefore

$$Q_s = 2 \ \forall s \geq 1 \ . \tag{5.4}$$

The construction of the sets $\mathcal{V}_s(k)$, as well as the calculation of how many elements belong to them, $V_s(k)$, is slightly more complex, but it can be performed effectively by induction.

Base Case: For $s = 1$, from (5.3) and according to the definitions given in Section 5.1.1, it is

$$\mathcal{Q}_1 = \{0_2, 1_2\} \ . \tag{5.5}$$

This is because the two 1-bit strings 0_2 and 1_2 both have one leading (and trailing) bit at the same value.

Induction: For $s \geq 2$, it is

$$\mathcal{V}_s(1) = \{x \wr \sim\mathrm{rb}(x) \mid x \in \mathcal{V}_{s-1}(k) \cup \mathcal{Q}_{s-1}\}, \ \ 1 \leq k \leq s-2 \ , \tag{5.6}$$

$$\mathcal{V}_s(k) = \{x \wr \mathrm{rb}(x) \mid x \in \mathcal{V}_{s-1}(k-1)\}, \ \ 2 \leq k \leq s-1 \ . \tag{5.7}$$

where the $\wr$ operator denotes string concatenation and the function $\mathrm{rb}(x)$ extracts the least significant (rightmost) bit of the binary string x given as an argument. The $\sim$ operator denotes Boolean one's complement. Informally speaking, the inductive construction of the sets $\mathcal{V}_s(k)$ proceeds as follows:

- The strings of length $s \geq 2$, with strictly less than s leading bits at the same value and exactly 1 trailing bit at the same value, belonging to $\mathcal{V}_s(1)$, can be obtained from any string of length $s-1$, belonging to either $\mathcal{V}_{s-1}(k)$, $1 \leq k \leq s-2$, or $\mathcal{Q}_{s-1}$, by appending an additional bit that is the complement of the rightmost one (5.6).
- The strings of length $s \geq 2$ with no more than $s-k$ leading bits at the same value and exactly k trailing bits at the same value, with $2 \leq k \leq s-1$ (the elements of $\mathcal{V}_s(k)$), can be built from any sequence of $s-1$ bits with no more than $s-k$ leading bits at the same value and exactly $k-1$ trailing bits at the same value (the elements of $\mathcal{V}_{s-1}(k-1)$) by appending an additional bit that is the same as the rightmost one (5.7).

Although (5.3) already gives a direct definition of $\mathcal{Q}_s$, it is worth noting anyway that the 2 strings of length $s \geq 1$ with s leading (and trailing) bits at the same value, the elements of $\mathcal{Q}_s$, can be trivially derived from the 2 sequences of length $s-1$ with $s-1$ leading (and trailing) bits at the same value, the elements of $\mathcal{Q}_{s-1}$, by appending an additional bit at the same value as the rightmost one. Namely

$$\mathcal{Q}_s = \{x \wr \mathrm{rb}(x) \mid x \in \mathcal{Q}_{s-1}\}, \ \ s \geq 2 \ . \tag{5.8}$$

Concerning the number of elements of $\mathcal{V}_s(k)$, denoted by $V_s(k)$, referring to (5.6) we can write

$$V_s(1) = \sum_{k=1}^{s-2} V_{s-1}(k) + Q_{s-1}, \ \ s \geq 2 \ , \tag{5.9}$$

because the sets $\mathcal{V}_{s-1}(k)$, $1 \leq k \leq s-2$, and $\mathcal{Q}_{s-1}$ are, by definition, all disjoint for a fixed s. From (5.7) we can similarly write

$$V_s(k) = V_{s-1}(k-1), \ \ 2 \leq k \leq s-1 \ . \tag{5.10}$$

By repeatedly applying (5.10) it is

$$V_s(k) = V_{s-k+1}(1) \ . \tag{5.11}$$

Substituting (5.4) and (5.11) into (5.9), we obtain

$$V_s(1) = \sum_{k=1}^{s-2} V_{s-k}(1) + Q_{s-1} = \sum_{k=2}^{s-1} V_k(1) + 2, \quad s \geq 2 \ . \tag{5.12}$$

From the previous expression we can also write

$$V_s(1) = V_{s-1}(1) + \left(\sum_{k=2}^{s-2} V_{s-k}(1) + 2 \right), \quad s \geq 3 \ , \tag{5.13}$$

where the rightmost part of the last expression, surrounded by parentheses, corresponds to the definition of $V_{s-1}(1)$. This is because, by letting $j = s - k$, we can write

$$\sum_{k=2}^{s-2} V_{s-k}(1) = \sum_{j=2}^{s-2} V_j(1) \tag{5.14}$$

Therefore, it is

$$V_s(1) = 2V_{s-1}(1) = 2^{s-2} V_2(1), \quad s \geq 3 \ . \tag{5.15}$$

Being $V_2(1) = Q_1 = 2$, from (5.4) and (5.9) we have, in general

$$V_s(1) = 2^{s-1}, \quad s \geq 2 \ . \tag{5.16}$$

From (5.11) we can eventually conclude that

$$V_s(k) = 2^{s-k}, \quad 1 \leq k \leq s-1 \ . \tag{5.17}$$

An additional consistency check of the above statements can be performed by noticing that, for any given s, the sets $\mathcal{V}_s(k), 1 \leq k \leq s-1$, and $\mathcal{Q}_s$ must cover the universe of the strings of length s, that is,

$$\bigcup_{k=1}^{s-1} \mathcal{V}_s(k) \cup \mathcal{Q}_s = \mathcal{U}_s, \quad s \geq 1 \ . \tag{5.18}$$

Since the sets being considered are all disjoint for a fixed s by definition, it must also be

$$\sum_{k=1}^{s-1} V_s(k) + Q_s = 2^s, \quad s \geq 1 \ . \tag{5.19}$$

In fact, from (5.17) and (5.4) it is

$$\sum_{k=1}^{s-1} V_s(k) + Q_s = \sum_{k=1}^{s-1} 2^{s-k} + 2 = 2 \left(\sum_{j=0}^{s-2} 2^j + 1 \right) = 2 \left(2^{s-1} - 1 + 1 \right) = 2^s \ . \tag{5.20}$$

The construction of the sets $\mathcal{V}_s(k)$ and $\mathcal{Q}_s$ by induction can also be depicted by means of the data flow diagram shown in Figure 5.1. In the figure, circles represent sets; they contain the name of the set and its cardinality. Beside each set, the strings belonging to that set and beginning with

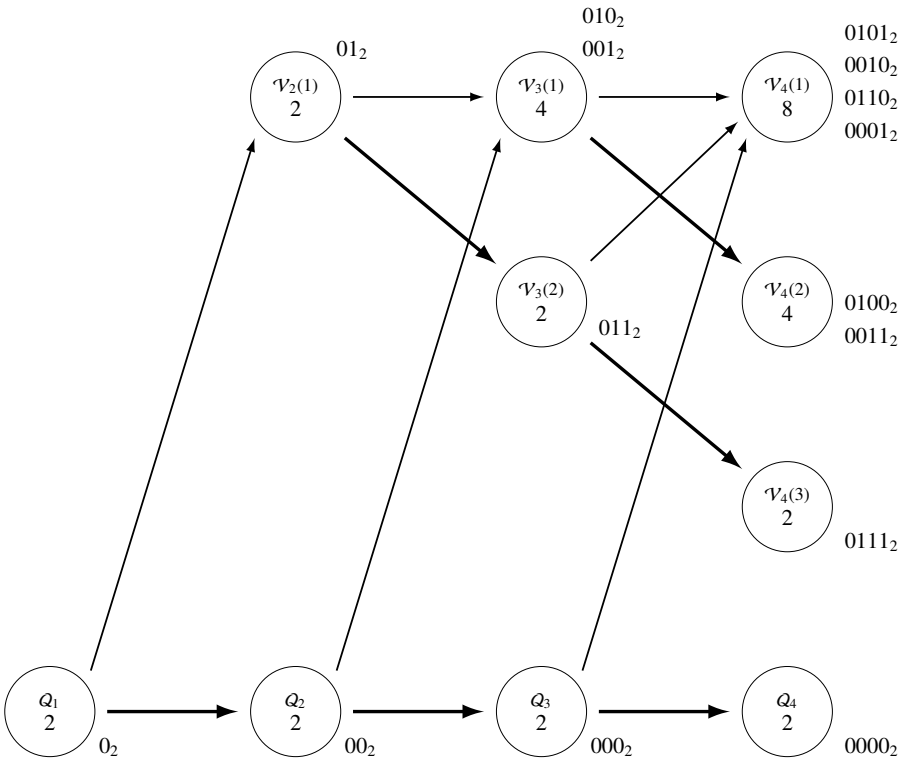

Figure 5.1. Construction of $\mathcal{V}_s(k)$ and $\mathcal{Q}_s$ by induction, for $1 \leq s \leq 4$.

0_2 are shown. Due to a property to be proved in Section 5.1.5, the complement of the strings shown in the figure belongs to the same set, too.

Thick lines represent the operation of appending to each element x of a set an additional bit equal to its rightmost bit, to obtain a new element y, that is, $y = x \wr \mathrm{rb}(x)$. Thin lines represent the operation of appending to each element x of a set an additional bit equal to the complement of its rightmost bit, to obtain a new element y, that is, $y = x \wr {\sim}\mathrm{rb}(x)$.

The strings of length $l + 1$ belonging to $\mathcal{V}_{l+1}(k), 1 \leq k \leq l$ will be used as a starting point to build longer strings, with at most l leading bits at the same value, by appending further bits. On the contrary, the strings belonging to $\mathcal{Q}_{l+1}$ must be excluded, because they have got $l + 1$ leading bits at the same value, and hence, already violate the requirements set forth in Section 5.1.1.

5.1.3 Inner Bits

It is possible to use the partial results presented in Section 5.1.2 to generate strings of length s, with at most l leading bits at the same value *and at most b consecutive inner bits* at the same value, assuming that $b > l$. The case $b \leq l$ is not of practical interest because it would make the assumption $l + t \leq b$, set forth in Section 5.1.1, impossible to satisfy.

The set denoted by $\mathcal{B}_{s,l,b}(k)$ will be formally defined as the set of strings of length $s > l$, which have got:

- at most l leading bits at the same value, and
- at most b consecutive inner bits at the same value, and
- exactly $1 \leq k \leq b$ trailing bits at the same value.

As before, we denote by $\mathcal{Q}_{s,l,b}$ the set of "invalid" strings, that is, strings of length s which do *not* satisfy the property set forth for any of the $\mathcal{B}_{s,l,b}(k)$ for the same s, that is, strings that have got either:

- more than l leading bits at the same value, or
- more than b consecutive inner bits at the same value.

As for $\mathcal{V}_s(k)$ and $\mathcal{Q}_s$, set generation is performed in an inductive way.

Base Case: The base case of the generation are the strings of length $s = l + 1$ belonging to $\mathcal{V}_{l+1}(k), 1 \leq k \leq l$. Consistently with the hypothesis $b \geq l$, we will let

$$\begin{aligned} \mathcal{B}_{l+1,l,b}(k) &= \mathcal{V}_{l+1}(k),\ 1 \leq k \leq l\ , \\ \mathcal{B}_{l+1,l,b}(k) &= \emptyset,\ l+1 \leq k \leq b\ . \end{aligned} \tag{5.21}$$

According to the definition of $\mathcal{Q}_{l+1,l,b}$, we can also let

$$\mathcal{Q}_{l+1,l,b} = \mathcal{Q}_{l+1}\ . \tag{5.22}$$

Induction: For $s \geq l + 2$, the induction proceeds according to the following formulae. First of all, it is

$$\mathcal{B}_{s,l,b}(1) = \{x \wr \sim\mathrm{rb}(x) \mid x \in \mathcal{B}_{s-1,l,b}(k)\},\ 1 \leq k \leq b\ , \tag{5.23}$$

because it is possible to build a string of length s that has no more than 1 trailing bit at the same value, by taking a string of length $s - 1$ that has no more than $1 \leq k \leq b$ trailing bits at the same value, and appending one more bit that is the complement of the rightmost one. This operation preserves the property that no strings generated in this way have either more than l leading bits at the same value or more than b consecutive inner bits at the same value.

Secondly, it is

$$\mathcal{B}_{s,l,b}(k) = \{x \wr \mathrm{rb}(x) \mid x \in \mathcal{B}_{s-1,l,b}(k-1)\},\ 2 \leq k \leq b\ , \tag{5.24}$$

because, by taking a string of length $s - 1$ with up to b consecutive inner bits at the same value, as well as $k - 1$ trailing bits at the same value, and appending one more bit at the same value as the rightmost one, a string of length s is obtained. That string has k trailing bits at the same value and, provided that $k \leq b$, it still has no more than b consecutive inner bits at the same value.

Finally, we can also write

$$\mathcal{Q}_{s,l,b} = \{x \wr 0_2 \mid x \in \mathcal{Q}_{s-1,l,b}\} \cup \{x \wr 1_2 \mid x \in \mathcal{Q}_{s-1,l,b}\} \cup \{x \wr \mathrm{rb}(x) \mid x \in \mathcal{B}_{s-1,l,b}(b)\}\ , \tag{5.25}$$

because an "invalid" string of length s can be obtained in two different ways:

1. By taking a string of length $s-1$ that already violates the requirement and appending one more bit, at *any* value. This case corresponds to the first two terms of (5.25).

2. By taking a string of length $s-1$ that satisfies the requirement, but has exactly b trailing bits at the same value, and adding one more bit at that same value to obtain a string with $b+1$ trailing bits at the same value. This case corresponds to the last term of (5.25).

Also in this case, it is worth remarking that the sets $\mathcal{B}_{s,l,b}(k)$, $1 \leq k \leq b$ and $\mathcal{Q}_{s,l,b}$ for a given s cover $\mathcal{U}_s$, that is,

$$\bigcup_{k=1}^{b} \mathcal{B}_{s,l,b}(k) \cup \mathcal{Q}_{s,l,b} = \mathcal{U}_s \ . \tag{5.26}$$

Denoting by $B_{s,l,b}(k)$ the number of elements of $\mathcal{B}_{s,l,b}(k)$ and referring to (5.23), taking into account that the sets $\mathcal{B}_{s-1,l,b}(k)$, $1 \leq k \leq b$ are all disjoint, it is

$$B_{s,l,b}(1) = \sum_{k=1}^{b} B_{s-1,l,b}(k) \ . \tag{5.27}$$

Then, from (5.24), it is

$$B_{s,l,b}(k) = B_{s-1,l,b}(k-1), \quad 2 \leq k \leq b \ . \tag{5.28}$$

Finally, from (5.25) we obtain that the number of elements of $\mathcal{Q}_{s,l,b}$, denoted by $Q_{s,l,b}$ is given by

$$Q_{s,l,b} = 2Q_{s-1,l,b} + B_{s-1,l,b}(b) \ , \tag{5.29}$$

in which the factor 2 comes from the fact that a string belonging to $\mathcal{Q}_{s,l,b}$ can be obtained from an element of $\mathcal{Q}_{s-1,l,b}$ by appending a bit at *any* value, that is, either 0_2 or 1_2.

By applying (5.28) repeatedly, it is

$$B_{s,l,b}(k) = B_{s-(k-1),l,b}(1), \quad s \geq l+k \ . \tag{5.30}$$

Substituting (5.30) back into (5.27) we can write, for $s \geq l+b+1$,

$$B_{s,l,b}(1) = \sum_{k=1}^{b} B_{s-k,l,b}(1) \ . \tag{5.31}$$

This formula shows that $B_{s,l,b}(1)$ can be calculated by means of a generalized Fibonacci sequence of order b. Due to (5.30), this property is also valid for $B_{s,l,b}(k)$, $1 \leq k \leq b$ when $s \geq l+k$.

The construction process of $\mathcal{B}_{s,l,b}(k)$ and $\mathcal{Q}_{s,l,b}$ for a case of interest is depicted in Figure 5.2 (next page). The notation is the same as in Figure 5.1, with a double, thin line denoting the operation of appending a bit at any value. To avoid cluttering the figure, only one exemplar string is shown for each set.

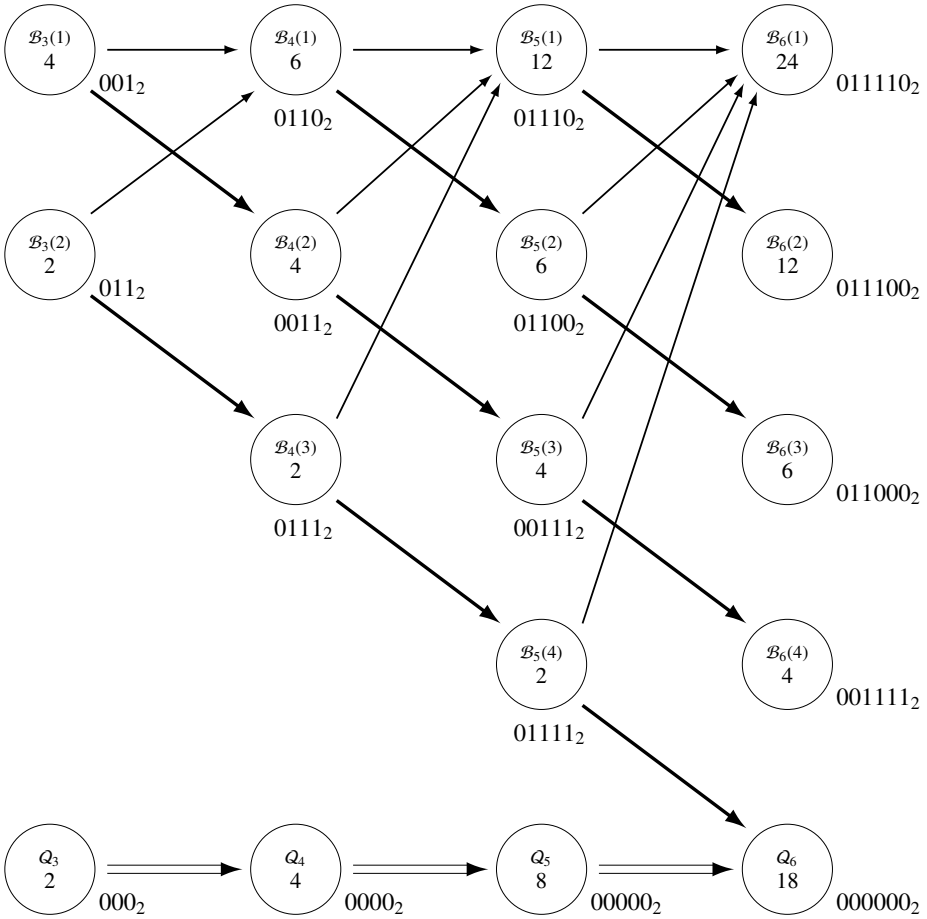

Figure 5.2. Construction of $\mathcal{B}_{s,l,b}(k)$ and $\mathcal{Q}_{s,l,b}$ by induction, for $l = 2$, $b = 4$, and $3 \leq s \leq 6$. Empty sets and indices l and b omitted for clarity.

5.1.4 Trailing Bits

The last step of the construction of the codebook $\mathcal{G}_{s,l,b,t}$ consists of taking into account the requirement about the maximum number of trailing bits at the same value, which shall be *no more than* t, with $t \leq b$. This is quite simple because $\mathcal{B}_{s,l,b}(k)$ satisfies all the requirements on s, l, and b and, moreover, contains strings with *exactly* k trailing bits at the same value.

We can therefore write

$$\mathcal{G}_{s,l,b,t} = \bigcup_{k=1}^{t} \mathcal{B}_{s,l,b}(k), \quad s \geq l + 1 \ . \tag{5.32}$$

By recalling again that all the sets $\mathcal{B}_{s,l,b}(k)$ are all disjoint for a fixed s, it is

$$G_{s,l,b,t} = \sum_{k=1}^{t} B_{s,l,b}(k) \ . \tag{5.33}$$

Moreover, it is easy to show that, in the corner case $s = l$, a suitable choice for $\mathcal{G}_{l,l,b,t}$, analogous to (5.32), is

$$\mathcal{G}_{l,l,b,t} = \bigcup_{k=1}^{\min(l-1,t)} \mathcal{V}_l(k) \ . \tag{5.34}$$

5.1.5 Codebook Size Reduction

Let $\mathcal{G}^*_{s-1,l,b,t}$ be the set of strings obtained from $\mathcal{G}_{s,l,b,t}$ by deleting the rightmost bit of its elements. By construction, all the elements of $\mathcal{G}^*_{s-1,l,b,t}$ are of length $s-1$. It can be proved that

$$\mathcal{G}_{s-1,l,b,t} \subseteq \mathcal{G}^*_{s-1,l,b,t}, \quad s > l+1 \ , \tag{5.35}$$

which, informally speaking, means that for fixed l, b, and t, the codebooks $\mathcal{G}_{s,l,b,,t}$ obtained by varying s are all *nested* into each other.

From the implementation point of view, this is very useful because, although all the codebooks $\mathcal{G}_{s,l,b,t}$ for $s = 2, \ldots, s_m$ are needed in a certain application, only $\mathcal{G}_{s_m,l,b,t}$ must be stored explicitly, because all the smaller codebooks can be obtained from a subset of the biggest one by deleting some of the rightmost bits of its elements.

From (5.32), since we assume that $t \leq b$, we can write

$$\mathcal{G}_{s-1,l,b,t} = \bigcup_{k=1}^{t} \mathcal{B}_{s-1,l,b}(k) \subseteq \bigcup_{k=1}^{b} \mathcal{B}_{s-1,l,b}(k) \ . \tag{5.36}$$

From (5.23), defining $\mathcal{B}^*_{s-1,l,b}(1)$ as the set of strings obtained from $\mathcal{B}_{s,l,b,t}(1)$ by removing the rightmost bit of its elements, we can also write

$$\mathcal{B}^*_{s-1,l,b}(1) = \{x, \quad x \in \mathcal{B}_{s-1,l,b}(k)\}, \ 1 \leq k \leq b \ , \tag{5.37}$$

and therefore

$$\mathcal{B}^*_{s-1,l,b}(1) = \bigcup_{k=1}^{b} \mathcal{B}_{s-1,l,b}(k) \ . \tag{5.38}$$

Substituting (5.38) back into (5.36) we obtain

$$\mathcal{G}_{s-1,l,b,t} \subseteq \mathcal{B}^*_{s-1,l,b}(1) \ . \tag{5.39}$$

From (5.32), being $t \geq 1$, it is

$$\mathcal{B}_{s,l,b}(1) \subseteq \mathcal{G}_{s,l,b,t} \ . \tag{5.40}$$

Remembering the definition of $\mathcal{G}^*_{s-1,l,b,t}$ and $\mathcal{B}^*_{s-1,l,b}(1)$, this implies

$$\mathcal{B}^*_{s-1,l,b}(1) \subseteq \mathcal{G}^*_{s-1,l,b,t} \ . \tag{5.41}$$

At this point, we obtain (5.35) by combining (5.39) and (5.41). The same nesting property can also be stated in a different way by observing that, due to (5.23) and (5.32), we can write

$$x \in \mathcal{G}_{s,l,b,t} \Rightarrow x \wr \sim \mathrm{rb}(x) \in \mathcal{G}_{s+1,l,b,t} \ . \tag{5.42}$$

Therefore, by induction,

$$x \in \mathcal{G}_{s,l,b,t} \Rightarrow x \wr \mathrm{abp}(\sim \mathrm{rb}(x), s' - s) \in \mathcal{G}_{s',l,b,t}, \quad s' > s \ , \tag{5.43}$$

where $\mathrm{abp}(b, l)$ is an alternating bit pattern of length l starting with bit b. This alternate formulation will be especially important to reduce the footprint of the reverse lookup table used by the decoder module, to be discussed in Section 5.2.2.

Further codebook storage savings can be achieved by observing that, due to how $\mathcal{G}_{s,l,b,t}$ has been defined, it is

$$g \in \mathcal{G}_{s,l,b,t} \Leftrightarrow \sim g \in \mathcal{G}_{s,l,b,t} \ . \tag{5.44}$$

Informally speaking, if g belongs to $\mathcal{G}_{s,l,b,t}$, then its one's complement $\sim g$ also belongs to the same codebook. It is therefore possible to define a *reduced* codebook $\mathcal{G}^{+}_{s,l,b,t}$ containing only half of the codewords of $\mathcal{G}_{s,l,b,t}$, namely, the ones beginning with zero. The amount of storage required for $\mathcal{G}^{+}_{s,l,b,t}$ with respect to $\mathcal{G}_{s,l,b,t}$, in bits, is considerably reduced because:

- it contains only half of the entries, and
- the leftmost bit of all entries is always zero, and hence, it is unnecessary to store it explicitly.

At the same time, it is still possible to calculate the elements of the full codebook from the reduced ones by means of simple and efficient binary operations.

Due to (5.6), (5.7), (5.21), (5.23), (5.24), and (5.32), the elements of $\mathcal{Q}_1$ become the leftmost bit of any codeword in $\mathcal{G}_{s,l,b,t}$. It is therefore possible to build $\mathcal{G}^{+}_{s,l,b,t}$ by using the same process (base case and induction) discussed before, but starting from $\mathcal{Q}^{+}_{1} = \{0_2\}$ instead of $\mathcal{Q}_1$.

5.2 Implementation and Analysis

The implementation activity concerns two complementary software modules, with very different goals and requirements, namely:

1. *Codebook generator.* This module is executed offline, once and for all. For this reason, the focus during software development shall be on *correctness*, rather than performance. It generates the codebooks $\mathcal{G}_{s,l,b,t}, 2 \leq s \leq s_m$, in a format suitable to be used by the other module and as compact as possible to reduce their memory footprint.

2. *Encoder and decoder.* The purpose of this module is to encode the data stream before transmitting it on the channel and, symmetrically, decode the data stream coming from the channel before handing it to the upper layers of the protocol stack. Since it resides in the critical path of the protocol stack, its *efficiency* is of paramount practical importance.

5.2.1 Codebook Generator

The codebook generator was implemented in ISO Prolog [47] for the SWI Prolog system [95]. The choice of a logic programming language instead of a more traditional one stems from the fact that logic languages in general, and Prolog in particular, directly support the inductive definition

```
1  in_qs(1, [0]).
2  in_qs(1, [1]).
3
4  in_qs(S, Y) :-
5    S >= 2, T is S-1, in_qs(T, X),
6    rb(X, R), app(X, R, Y).
7
8  in_rs_anyk(S, X) :-
9    in_rs_anyk_iter(S, 1, X).
10
11 in_rs_anyk_iter(S, K, X) :-
12   in_rsk(S, K, X).
13
14 in_rs_anyk_iter(S, K, X) :-
15   KK is K+1, KK < S,
16   in_rs_anyk_iter(S, KK, X).
17
18 in_rsk(S, 1, Y) :-
19   S >= 2, T is S-1, in_rs_anyk(T, X),
20   rb(X, R), neg(R, NR), app(X, NR, Y).
21
22 in_rsk(S, 1, Y) :-
23   S >= 2, T is S-1, in_qs(T, X),
24   rb(X, R), neg(R, NR), app(X, NR, Y).
25
26 in_rsk(S, K, Y) :-
27   S >= 2, K >= 2,
28   T is S-1, H is K-1, in_rsk(T, H, X),
29   rb(X, R), app(X, R, Y).
```

Figure 5.3. Prolog definition of $\mathcal{V}_s(k)$ and $\mathcal{Q}_s$.

of sets. Hence, the program is very concise and its statements closely resemble the mathematical definitions given in Section 5.1. In turn, this greatly reduces the likelihood of programming mistakes.

For instance, the generation of the sets $\mathcal{V}_s(k)$ and $\mathcal{Q}_s$ defined in Section 5.1.2 has been programmed as shown in Figure 5.3. The functors `rb(X, Y)`, `neg(X, Y)`, and `app(X, R, Y)` correspond to $y = \mathrm{rb}(x)$, $y = {\sim}x$, and $y = x \wr r$, respectively. Their definition consists of a few lines of Prolog code and is not further discussed for conciseness. Bit strings are modeled as lists.

The base case for the definition of $\mathcal{Q}_s$ is given at lines 1–2. The two facts concerning the functor `in_qs` correspond to (5.5) and assert the initial contents of $\mathcal{Q}_1$. Then, lines 4–6 give the inductive construction of $\mathcal{Q}_s$ for $s \geq 2$ in terms of $\mathcal{Q}_{s-1}$, as defined in (5.8). The inductive construction of $\mathcal{V}_s(1)$ is slightly more complex because, as defined in (5.6), it entails the enumeration of all elements belonging to $\mathcal{V}_{s-1}(k)$ for $1 \leq k \leq s-2$. This is performed by the helper functor `in_rs_anyk` and its iterator `in_rs_anyk_iter` at lines 8–16. The inductive construction itself is performed by the two induction rules found at lines 18–24. Finally, the inductive construction of $\mathcal{V}_s(k)$ for $2 \leq k \leq s-1$, which corresponds to (5.7) is given at lines 26—29.

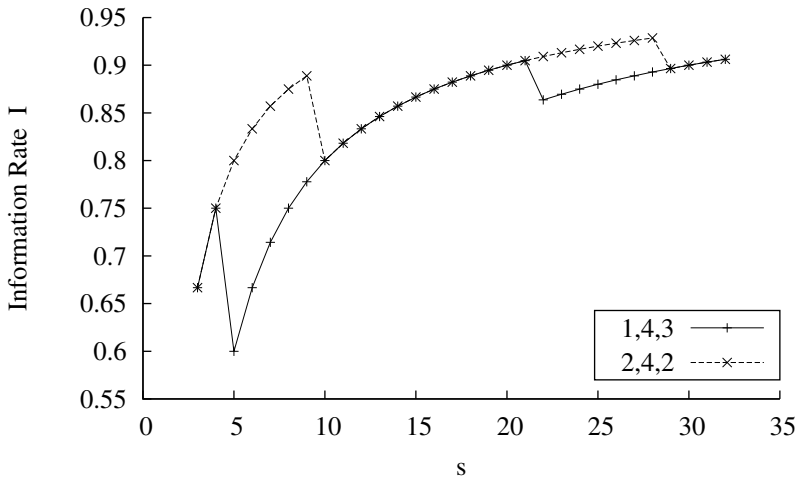

Figure 5.4. Information rate I of codebooks $\mathcal{G}_{s,1,4,3}$ and $\mathcal{G}_{s,2,4,2}$ as a function of $3 \leq s \leq 32$.

With those definition, the whole set of codewords belonging, for instance, to $\mathcal{V}_4(2)$ can easily be found with the query

```
bagof(CW, in_rsk(4, 2, CW), CB).
```

The result is bound to variable `CB`:

```
CB = [[0, 1, 0, 0], [1, 0, 1, 1],
      [0, 0, 1, 1], [1, 1, 0, 0]].
```

All the other sets presented in Sections 5.1.3 and 5.1.4 have been defined in a very similar way and are not shown here for conciseness. It should be noted that, as discussed in Section 5.1.5, the code shown in Figure 5.3 can be used to generate either the full codebooks or the reduced ones, by including or excluding line 2, respectively. No modifications to any other part of the code are necessary to this purpose.

The codebook generator output can be used for a variety of purposes. First of all, the quantities $G_{s,l,b,t}$ provide a mean to evaluate and compare the efficiency of different codebooks for different parameter values. In fact, a codebook composed of $G_{s,l,b,t}$ strings of length s is able to encode any string of length

$$s' = \lfloor \log_2 G_{s,l,b,t} \rfloor \ . \tag{5.45}$$

The efficiency of the codebook is then given by its *information rate* I, that is, the ratio

$$I = \frac{s'}{s} = \frac{1}{s} \lfloor \log_2 G_{s,l,b,t} \rfloor \ . \tag{5.46}$$

Figure 5.4 shows the information rate of the codebooks $\mathcal{G}_{s,1,4,3}$ and $\mathcal{G}_{s,2,4,2}$ as a function of s. Although they both satisfy the requirements needed to prevent CAN bit stuffing, $\mathcal{G}_{s,2,4,2}$ has a better efficiency for some values of s, namely, for $5 \leq s \leq 9$. Another interesting information that can be inferred from Figure 5.4 is that $\mathcal{G}_{9,2,4,2}$ has the best efficiency for any value of $s \leq 18$. The result highlights that for a memoryless block encoder for CAN, $\mathcal{G}_{9,2,4,2}$ is the best choice.

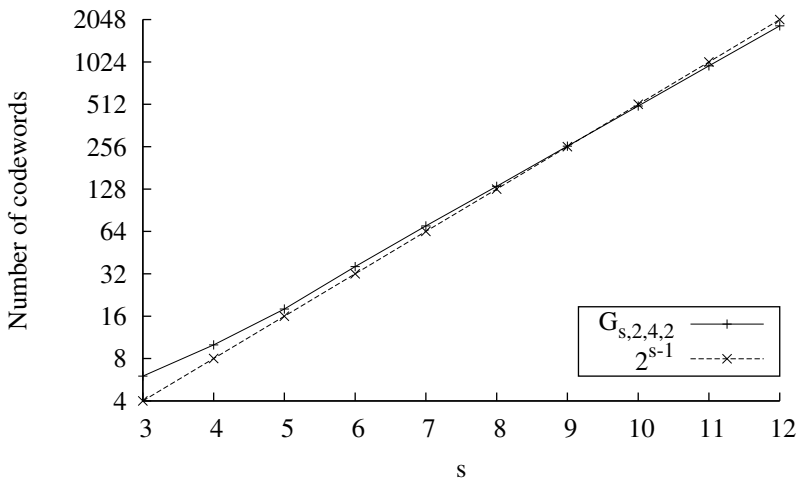

Figure 5.5. Ability to encode a binary string of length $s-1$ by means of $\mathcal{G}_{s,2,4,2}$ as a function of $3 \leq s \leq 12$.

In order to achieve a better efficiency, it would in fact be necessary to adopt a codebook with $s > 18$. However, in this case, the footprint of the encoding and decoding lookup tables—which is proportional to the codebook size—would be too large, at least for a small embedded system.

Moreover, the codebook generator output has also been used to check which codebooks of the form $\mathcal{G}_{s,2,4,2}$ satisfy property (5.2) and, as a consequence, determine the value of s_m discussed in Section 5.1.1. Figure 5.5 compares the two sides of (5.2) and shows that, in the case being considered, $s_m = 9$. In other words, any codebook $\mathcal{G}_{s,2,4,2}$ with $s \leq s_m$ has enough elements to encode any string of length $s-1$, but this property no longer holds for any $s > s_m$.

Last, but not least, the codebook generator has been used as the starting point to build the forward and reverse lookup tables for the encoder and decoder module, to be discussed in Section 5.2.2. As an example, Table 5.1 (next page) shows the codebooks $\mathcal{G}^{+}_{s,2,4,2}, 2 \leq s \leq 9$, highlighting the nesting property presented in Section 5.1.5. Defining the inference rules in the right order within the Prolog code—namely, assuring that the inference rule corresponding to (5.23) comes before the one corresponding to (5.24)—ensures that, when the largest codebook $\mathcal{G}^{+}_{9,2,4,2}$ is being generated, the codebook entries corresponding to the nested, smaller codebooks, are generated first, as shown in the table.

5.2.2 Encoder and Decoder

Besides dealing with payload encoding and decoding, with the help of the codebook presented in Section 5.2.1, the actual encoder and decoder software modules must also take care of other important low-level details—for instance, format the encoded message as a whole in a proper way and prevent other parts of the message, namely the header, from injecting part of a primer sequence into the payload. Both goals must be accomplished in an *efficient* way, where efficiency has got two different, and possibly conflicting, meanings:

Codebook	Codeword
$\mathcal{G}^+_{2,2,4,2}$	0 1 | 0 1 0 1 0 1 0
	0 0 1 0 1 0 1 0 1
$\mathcal{G}^+_{3,2,4,2}$	0 1 1 | 0 1 0 1 0 1
	0 1 0 0 1 0 1 0 1
$\mathcal{G}^+_{4,2,4,2}$	0 0 1 1 | 0 1 0 1 0
	0 1 1 1 0 1 0 1 0
	0 1 0 1 1 0 1 0 1
	0 0 1 0 0 1 0 1 0
$\mathcal{G}^+_{5,2,4,2}$	0 1 1 0 0 | 1 0 1 0
	0 1 0 0 0 1 0 1 0
	0 0 1 1 1 0 1 0 1
	0 1 1 1 1 0 1 0 1
	0 1 0 1 0 0 1 0 1
	0 0 1 0 1 1 0 1 0
	0 1 1 0 1 1 0 1 0
	0 1 0 0 1 1 0 1 0
	0 0 1 1 0 0 1 0 1
$\mathcal{G}^+_{6,2,4,2}$	0 1 1 1 0 0 | 1 0 1
	0 1 0 1 1 1 0 1 0
	0 0 1 0 0 0 1 0 1
	0 1 1 0 0 0 1 0 1
	0 1 0 0 0 0 1 0 1
	0 0 1 1 1 1 0 1 0
	0 1 0 1 0 1 1 0 1
	0 0 1 0 1 0 0 1 0
	0 1 1 0 1 0 0 1 0
	0 1 0 0 1 0 0 1 0
	0 0 1 1 0 1 1 0 1
	0 1 1 1 0 1 1 0 1
	0 1 0 1 1 0 0 1 0
	0 0 1 0 0 1 1 0 1
	0 1 1 0 0 1 1 0 1
	0 1 0 0 0 1 1 0 1
	0 0 1 1 1 0 0 1 0
$\mathcal{G}^+_{7,2,4,2}$	0 1 1 1 1 0 0 | 1 0

Codebook	Codeword
	0 1 0 1 0 0 0 1 0
	0 0 1 0 1 1 1 0 1
	0 1 1 0 1 1 1 0 1
	0 1 0 0 1 1 1 0 1
	0 0 1 1 0 0 0 1 0
	0 1 1 1 0 0 0 1 0
	0 1 0 1 1 1 1 0 1
	0 0 1 0 0 0 0 1 0
	0 1 1 0 0 0 0 1 0
	0 1 0 1 0 1 0 0 1
	0 0 1 0 1 0 1 1 0
	0 1 1 0 1 0 1 1 0
	0 1 0 0 1 0 1 1 0
	0 0 1 1 0 1 0 0 1
	0 1 1 1 0 1 0 0 1
	0 1 0 1 1 0 1 1 0
	0 0 1 0 0 1 0 0 1
	0 1 1 0 0 1 0 0 1
	0 1 0 0 0 1 0 0 1
	0 0 1 1 1 0 0 1 0
	0 1 1 1 1 0 1 1 0
	0 1 0 1 0 0 1 1 0
	0 0 1 0 1 1 0 0 1
	0 1 1 0 1 1 0 0 1
	0 1 0 0 1 1 0 0 1
	0 0 1 1 0 0 1 1 0
	0 1 1 1 0 0 1 1 0
	0 1 0 1 1 1 0 0 1
	0 0 1 0 0 0 1 1 0
	0 1 1 0 0 0 1 1 0
	0 1 0 0 0 0 1 1 0
$\mathcal{G}^+_{8,2,4,2}$	0 0 1 1 1 1 0 0 | 1
	0 1 0 1 0 1 1 1 0
	.
	...
	.....
	...
	.
$\mathcal{G}^+_{9,2,4,2}$	0 1 1 0 0 0 0 1 1

Table 5.1. Reduced codebooks $\mathcal{G}^+_{s,2,4,2}$, $2 \le s \le 9$, showing how they are nested into each other.

Original payload Maximum size *n.m* [B.b]	Encoded data field DLC [B]	DLC value	BB value	8B9B seq. size [b]	PAD size [b]
0.0	0	0000	—	0	0
0.7	1	0001	—	0	8
1.6	2	0010	—	9	7
2.4	3	0011	0	18	5
3.4	4	0100	—	27	5
4.3	5	0101	—	36	4
5.2	6	0110	—	45	3
6.1	7	0111	0	54	1
7.0	8	1000	1	63	0

Table 5.2. Size of the 8B9B-encoded data field vs. original DLC.

- *space* efficiency, to make the best possible use of the limited payload carrying capabilities of CAN;
- *time* efficiency, because the encoding and decoding processes must be as fast as possible.

For what concerns the break bit (BB), a simple optimization consists of including it only when necessary, that is, when the DLC in the transmitted frame is equal to 7, 8, or 3. In the former two cases it is easy to see that the corresponding DLC bit patterns (0111_2 and 1000_2) include 3 bits at the same value at their end. In the absence of BB, it would no longer be ensured that no stuff bit may appear when a codeword is appended. The latter case (0011_2) leads to the same issue when a stuff bit (at 1_2) is inserted just after the two initial bits at 0_2 because the part of header that precedes the DLC ends with 000_2.

Another drawback of 8B9B is that part of the data field is actually *wasted*. In fact, the encoded payload is typically not aligned on a byte boundary, because it is made up of an integral number of 9-bit codewords, and must be properly padded at the end. The VHCC encoding scheme overcomes this limitation. In fact, instead of filling the PAD field using a fixed alternating bit pattern—just to prevent bit stuffing—it is now possible to leverage it suitably in order to pack further *information*. If the PAD field is made up of s bits, the codebook $\mathcal{G}^{+}_{s,2,4,2}$ discussed in Section 5.1 can be used to efficiently store $s-1$ bits of additional data. For instance, when the PAD is 7-bit long (as happens when the original payload takes one whole byte and DLC equals 2), it is possible to accommodate a further piece of information comprising up to 6 data bits. This means, 14 bits can be encoded in this case with VHCC versus the 8 bits allowed in 8B9B.

In practice, an additional parameter PS (PAD selector) expressed on one byte is provided to the encoding function—besides the original payload—which is used for determining the content of the PAD field (the old name has been retained for simplicity, in spite of the fact its meaning has actually changed). Basically, after all the bytes of the original payload have been converted to the related codewords, one additional translation is carried out using the least significant part of PS. For obvious reasons, not every bit of this byte will be encoded (see Table 5.2 for details). In the case no additional information are available, the user has to provide a dummy value for PS nonetheless, which will never be used on the receiver side. The contents of PS will be encoded

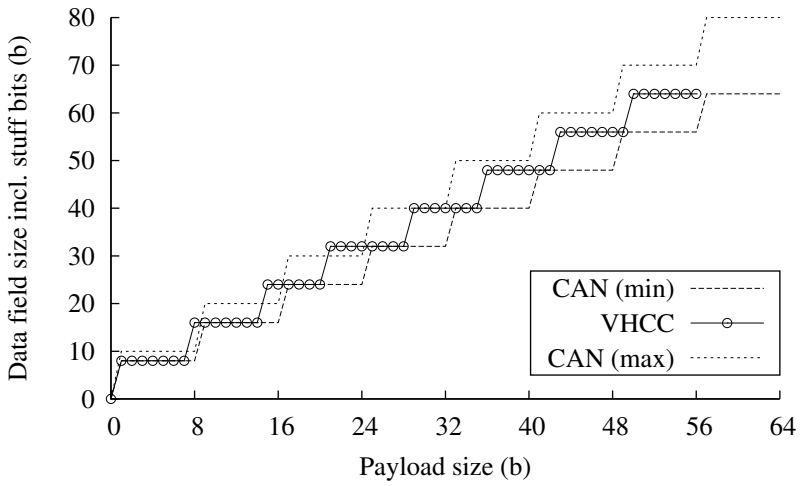

Figure 5.6. VHCC data field size, as a function of the payload size, with respect to plain CAN.

anyway, and hence, the PAD field will be set to a value that prevents primer sequences in the data field, as happened in 8B9B with the original, alternating bit pattern.

The additional byte PS introduced above can be seen as part of an "extended" payload and appended at byte position $n + 1$. In the following, the notation $n.m$ will be used to denote the size of such a payload, made up of n bytes (base size) and m bits (additional information). In Table 5.2, details about the enhancements achieved by the VHCC encoding scheme are shown. Particularly, for every DLC value, the maximum size allowed for the payload is reported (expressed as $n.m$), as well as the presence (and value) of BB. It is worth noting that, in the original 8B9B, encoded frames with a DLC value equal to 1 were forbidden. Instead, in VHCC, this case is permitted and can be used to achieve sub-byte information encoding (up to 7 bits).

The approach defined above could be profitably adopted, for instance, when the payload is obtained by collecting together several signals defined at the application level (as for PDO mapping in CANopen) or when information is encoded on more bits than strictly necessary (this happens, e.g., if a small set of enumerated values takes one whole byte, as in the case of the state information found in some NMT messages of CANopen). In these cases, the payload (as seen by the application) is best described as a sequence of bits, instead of bytes—indeed, the requirement that the payload takes an integral number of bytes depends basically on CAN.

In Figure 5.6, the duration (in bit times) of the data field in the frame sent over the bus is shown for plain CAN and VHCC for different sizes of the payload (also expressed in bits). The other fields of the CAN frame were not taken into account explicitly, because either BS cannot be prevented (CRC) with these methods, it does not apply (unstuffed trailer part), or it can be tackled by means of a careful selection of the message parameters (i.e., the identifier). It is worth noting that the actual duration in CAN is not fixed, because of the stuff bits possibly added by the CAN controller. In the worst case, up to 2 bits may be added to every original byte. As a consequence, two plots labeled "CAN (min)" and "CAN (max)" were added to Figure 5.6, which correspond to the best and worst cases, respectively. It can be seen that, overall, VHCC is often quite close

to the best case in CAN, and rarely it happens to be (slightly) worse. The major advantage with respect to plain CAN is the noticeably lower transmission jitter, especially when the payload is large. More specifically, the residual jitter only comes from the CRC field.

From the implementation point of view, Table 5.2 includes two borderline cases:

1. The first row of the table corresponds to a completely empty payload (payload size 0.0). To streamline the encoding software, it is assumed that this case—usually related to peculiar classes of messages—will be handled specially in the application layer, so that the encoder will never encounter it.

2. In the penultimate row of the table (payload size 6.1), there is only one PAD bit available. In order to ensure that the payload still ends with no more than 2 bits at the same value—and preserve one of the basic code properties—it would be necessary to set the PAD bit to the complement of the last bit of the last codeword.

   However, since the proposed scheme does not handle CRC jitter in any way, a more efficient implementation has been obtained by simply using that bit to transfer one more payload bit. It is, in fact, possible to design the encoding and decoding algorithms so that they gracefully fall back to this behavior when executed with a one-bit input and one-bit output. As a side effect, we accept that up to 3 bits at the same value may appear at the end of the payload. And this may increases the probability of having stuff bits in the CRC.

   This choice also explains the apparent anomaly of Table 5.2, in which the PAD size is always 1 bit more than m (the additional information part of the payload), whereas in this case it is not.

The most critical part of the implementation, regarding performance, is how to individually encode and decode each byte of the payload, as well as PS. To attain maximum efficiency, this is performed by means of two lookup tables:

1. *Forward lookup table.* The first 2^7 entries of the reduced codebook shown in Table 5.1 embody the forward lookup table of the encoder for $\mathcal{G}^{+}_{9,2,4,2}$. The 8-bit word P to be encoded shall first of all be reduced, by masking off its most significant bit, and then used as an index in the table. The contents of the table will then be either used directly as a codeword, or complemented, depending on whether the most significant bit of P was 0_2 or 1_2.

   The table will be only 8-bit wide instead of 9 when stored in memory, because the leftmost bit is zero in all its entries and it is therefore unnecessary to store it explicitly.

   To encode a word P according to the smaller codebooks $\mathcal{G}^{+}_{s,2,4,2}$, with $2 \leq s < 9$, due to the codebook nesting property, P itself can still be used as an index in the forward lookup table as discussed before. Then, the codeword derived from the table must be shifted/masked to extract its s most significant bits.

2. *Reverse lookup table.* Table 5.1 can be used to generate the reverse lookup table for $\mathcal{G}^{+}_{9,2,4,2}$, by inverting it. Namely, if the forward lookup table contains codeword g at index P, with $0 \leq P < 2^7$, the reverse lookup table will contain the value P at index g. Reduced codewords g are 8-bit wide, while P is 7-bit wide, and hence, the reverse lookup table must have 2^8 7-bit entries.

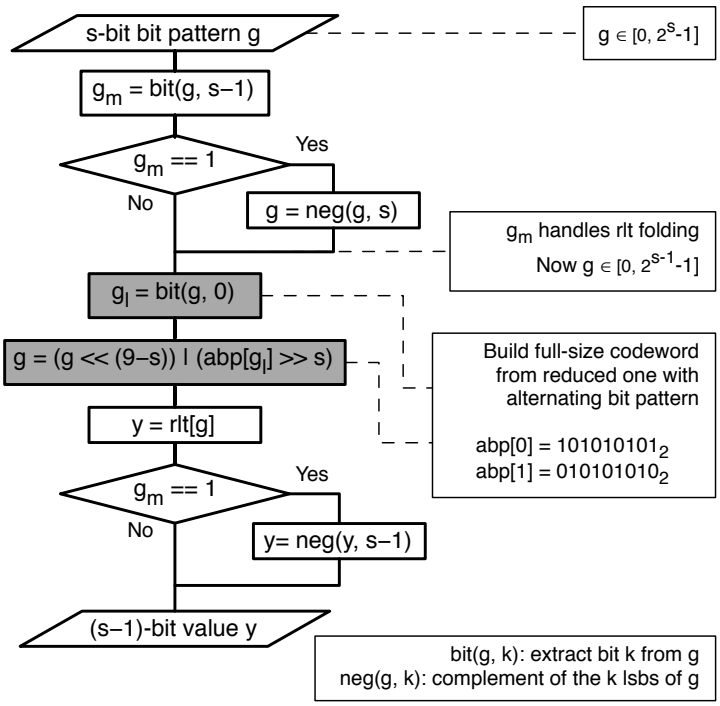

Figure 5.7. Access procedure to the reduced reverse lookup table for codewords of any size.

Since only 2^7 reduced codewords exist, not all entries of the reverse lookup table will be filled in this way. The remaining entries correspond to invalid bit patterns, which are not valid encoder outputs.

Due to property (5.43), the same reverse lookup table can also be used when smaller codebooks $\mathcal{G}_{s,2,4,2}$, $2 \leq s < 9$ are needed. In this case, the encoded bit pattern must be padded to the right with an alternating bit pattern, beginning with the complement of its least significant bit, before being used as an index in the reverse lookup table of $\mathcal{G}^{+}_{9,2,4,2}$. The length of the alternating bit pattern must be chosen to bring the total width of the index to 8 bits.

Figure 5.7 fully describes how the reverse lookup table is handled to decode a single codeword of any size, since this is the most complex operation to be performed. This procedure is used to handle both the 9-bit codewords coming from byte-by-byte payload encoding, as well as the smaller codeword coming form PS encoding.

In the flowchart, the function bit(g, k) extracts bit number k from word g. Bits are numbered from zero starting with the least significant one. Function neg(g, k) returns the complement of the k least significant bits of g. Both functions are straightforward to express in any high-level programming language and are amenable to a very efficient machine-language translation on most modern processor architectures.

- The procedure starts with an s-bit pattern g extracted from the CAN message payload. The value of g ranges from 0 to $2^s - 1$, inclusive.

- The most significant bit of g, called g_m, is used to reduce the range of g in preparation to the reverse lookup table access. In particular, if $g_m = 1_2$, g is complemented to make sure that its range is from 0 and $2^{s-1} - 1$, inclusive.

- If necessary, g is extended to the right with an alternating bit pattern, in order to make it 8-bit wide. The correct bit pattern to be used (out of the two possible ones) depends on the least significant bit of g before extension, called g_l. In the flowchart, both bit patterns are stored in the 2-element array abp[] and g_l is used as an index into it. However, for efficiency, in the actual implementation they are specified as constants in the code.

- The reduced reverse lookup table rlt[] is accessed to get the decoded value. Depending on the value of g_m, the decoded value is either the value found in the table or its complement.

- At the end of the procedure, y contains the decoded value in its $s - 1$ least significant bits.

The additional consistency checks involving invalid bit patterns are not shown in Figure 5.7 for clarity. In addition, when $s = 9$, the operations specified in the gray blocks can be skipped for efficiency.

5.3 VHCC Correctness, Performance and Footprint

A prototype implementation of VHCC has been developed for the NXP LPC 1768 microcontroller [72, 73] and then tested, focusing on *correctness*, *performance*, and *footprint* evaluation. The microcontroller was configured to run with a core clock frequency of 100 MHz and the software was developed by means of an open-source toolchain based on `binutils` [76], `gcc` [92], and `newlib` [13].

The correctness of a single encoding and decoding *step*—involving forward and reverse lookup table access—has been verified exhaustively, by encoding all the possible bit patterns of length between 1 and 8 bits inclusive, and then decoding the result. Lengths from 1 to 7 bits inclusive cover all the possible cases for PS encoding and decoding, while the length of 8 refers to how a whole byte of the payload is processed.

However, verifying the encoding and decoding modules *as a whole* in the same way would have been infeasible, due to the huge size of the sample space. Instead, a probabilistic approach has been adopted. Namely, correctness has been checked on a large number (10^7) of randomly-generated payloads for each length from 0.7 to 7.0 (see Table 5.2).

Similarly, performance was evaluated by measuring the encoding and decoding times of randomly-generated payloads of varying sizes by means of one LPC 1768 internal timer, with a resolution of 10 ns. Figure 5.8 (next page) shows the encoding and decoding times as a function of payload size.

By looking at the figure, it can be noted that the main properties of the 8B9B implementation are still valid for VHCC. The execution time *jitter* is in fact below the minimum measurable amount of 10 ns, that is, one core clock cycle. For this reason, the sample variance has not been shown in Figure 5.8. Moreover, the overall execution time of the encoder and decoder modules never exceeds a few bit times, even at the maximum CAN bit rate of 1 Mbps.

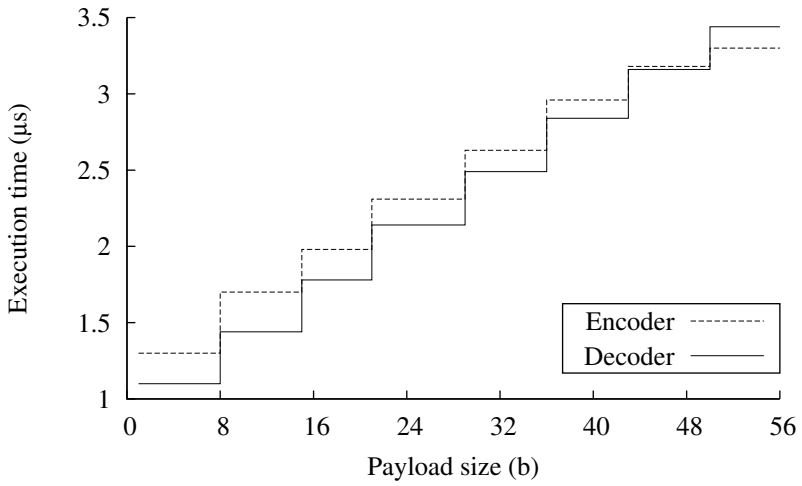

Figure 5.8. VHCC encoding and decoding time as a function of payload size, 10^7 random samples for each size.

For what concerns footprint, Table 5.3 details the size of the encoding and decoding modules in different memory *segments*. This breakdown is important because, in a typical embedded system, different segments are allocated to different kinds of memory, with their own features. For instance, the *constant data* (cdata) segment can be stored in a read-only memory, whereas the stack, being heavily used, requires a fast read-write memory. Furthermore, the code footprint has been broken down to shed more light on the relative size of the encoding and decoding loops, handling full payload bytes, with respect to loop setup as well as PS encoding and decoding.

In the first place, it is important to remark that the total footprint of the combined VHCC encoding and decoding modules is only 1052 bytes for the code and constant data segments, plus either 48 or 40 bytes of stack space, respectively. This figure, just above 1 kilobyte, is likely to be acceptable even in a very small and resource-limited embedded system.

Moreover, the footprint results indirectly confirm the consistency of the performance data shown in Figure 5.8. Although the exact relationship between the size of a code fragment and its execution time is not straightforward on current processors, it can qualitatively be seen that one very likely reason of the slowness of the encoder with respect to the decoder for small payload sizes is due to its bigger loop setup and PS encoding sections. Similarly, the execution time of the encoder grows slower than the decoder as the payload size increases, because its loop section is smaller.

5.4 Comparison with Competing Approaches

The problem of BS-induced jitters in CAN is well known, and a number of remedies have been proposed in the literature over the past years. Early solutions [68, 65], also known as XOR-based approaches, operate by scrambling the contents of the data field before transmission. Despite providing tangible improvements in several operating conditions, they are unable to ensure that

Component	Segment	Size [B]
Forward lookup table	cdata	128
Reverse lookup table	cdata	256
Encoder	stack	48
Encoder (total)	code	368
Encoder (loop setup)	code	168
Encoder (loop)	code	74
Encoder (PS)	code	126
Decoder	stack	40
Decoder (total)	code	300
Decoder (loop setup)	code	130
Decoder (loop)	code	76
Decoder (PS)	code	94

Table 5.3. Footprint of the VHCC encoding and decoding modules, LPC 1768 microcontroller and `gcc`-based toolchain.

stuff bits are completely prevented in the data field. The comparison with respect to XOR-based approach for what concerns overall encoding efficiency and jitter reduction capability has been discussed thoroughly in Chapter 4. As a consequence, they have not been considered in the following.

More recent solutions, instead, operate by encoding the data field in a suitable way, so that stuff bits can never occur in this part of the frame. This is the case of software bit stuffing (SBS) [66], eight-to-eleven modulation (EEM) [64] and 8B9B. A comparison is carried out in this section between VHCC and these approaches. Since EEM and 8B9B are very similar and the second one can be seen as an optimized version of the first, only 8B9B and SBS will be considered.

In principle, other solutions exist that are able to ensure jitter-free communication. In [91] a CAN-like protocol is described, with grounds on the overclocking technique first proposed in [12]. An enhancement of the basic technique is introduced in [91], which exploits a variable-size dummy field (an alternating bit pattern called *Data_Plus*, located after the CRC) so as to guarantee that the duration of frame transmissions does not depend on the content of the payload, but only on its size. This idea is quite appealing as, unlike the other solutions listed above, it permits to remove BS-jitters altogether (including those caused by stuff bits added to the CRC). More recently CAN-FD [83], which relies on similar concepts, has also been introduced. However, neither their implementation is possible on conventional CAN controllers, nor do these approaches ensure backward compatibility with legacy CAN networks. Thus, they will not be taken into account further.

The comparison among SBS, 8B9B, and VHCC includes maximum payload size, codec encoding efficiency, execution time, and memory footprint as performance metrics. As said above, the effectiveness of jitter reduction is reasonably the same for these approaches since, in all cases, stuff bits may be inserted only in the CRC. Because of the way the CRC is computed, the same amount of residual jitter could be expected on the average. For the same reason, only codec encoding efficiency is considered here.

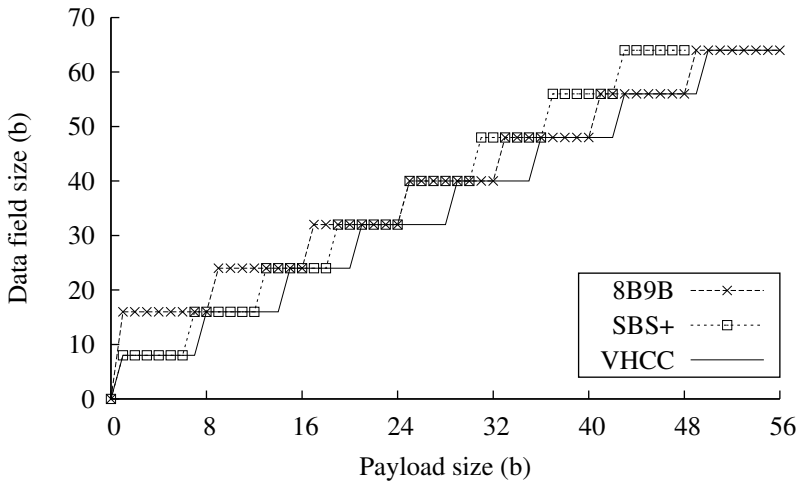

Figure 5.9. Data field size, as a function of the payload size, for diverse encoding techniques.

5.4.1 Maximum Payload Size and Codec Encoding Efficiency

When the payload (as seen at the application level) takes an integral number of bytes, 8B9B and VHCC have exactly the same encoding efficiency, whereas SBS behaves slightly worse because of the higher overhead (potentially, up to one software stuff bit may be added every three original bits).

Things change when the payload can take an arbitrary number of bits. Figure 5.9 shows the size of the data field in the frame sent over the bus for SBS, 8B9B, and VHCC for different sizes of the payload. In order to carry out a fair comparison, a slightly different version of SBS is considered here, called SBS+. This is because SBS was initially conceived for use with the shared-clock (S-C) [77] synchronization mechanism. SBS+ is almost the same as the original one, but, a) it does not encode any additional information in the data field (besides the original payload) and, b) it can also tackle the cases where the original payload is not aligned on a byte boundary.

As can be seen from the figure, VHCC never leads to a larger data field than 8B9B and SBS+. Thus, as expected, it is the optimal choice for achieving low-jitter communication in CAN. VHCC is particularly advantageous over 8B9B when the original payload size is small—which is also the case where the encoding efficiency of the latter was noticeably worse than plain CAN. In these conditions, both VHCC and SBS+ outperform 8B9B for what concerns encoding efficiency. This is because, 8B9B was not conceived to deal with sub-byte information: for instance, in order to encode a single bit, two bytes are needed. Conversely, 8B9B excels for large payloads, where it resembles VHCC closely.

5.4.2 Execution Time and Footprint

The execution time at maximum payload size and code/data footprint of 8B9B, SBS, and VHCC are shown in Table 5.4. The figures for SBS were taken from published literature [66] and results

Method/Component	**Exec. time [μs]**	**Footprint (C+D) [B]**
8B9B, LPC 1768 at 100 MHz		
Encoder	2.70	160 + 156
Decoder	2.54	112 + 276
SBS, LPC 2129 at 60 MHz		
Encoder	158.5	484 + 34
Decoder	160.8	320 + 18
VHCC, LPC 1768 at 100 MHz		
Encoder	3.30	368 + 176
Decoder	3.44	300 + 296

Table 5.4. Execution time at maximum payload size and code/data footprint of 8B9B, SBS, and VHCC.

of 8B9B were taken from Chapter 3, while they were derived from Figure 5.8 and Table 5.3 for VHCC. Regarding VHCC, footprint data have been aggregated to bring them down to the same level of detail as the other methods and simplify the comparison. In particular, the data footprint includes both the constant and variable data segments, as well as the stack space. For the sake of the comparison, it is also assumed that the execution time and footprint of SBS and SBS+ are the same, regardless of the differences between them.

First of all, it is important to remark that a direct comparison among the figures shown in Table 5.4 is not possible, because the processor core used for SBS is not the same as the other and the two cores differ significantly in terms of instruction set and clock speed. In fact, the LPC 2129 uses an ARM7 core, whereas the LPC 1768 adopts a Cortex-M3 core running at a higher clock speed.

It can however be noted that, despite the noticeable difference in flexibility between the two methods, the VHCC and 8B9B execution times are still comparable. The VHCC execution time is roughly equal to the 8B9B execution time with one more payload byte plus some additional overhead. This relationship can easily be justified by noticing that the complexity of PS encoding in VHCC is of the same order of magnitude as the encoding of one more full byte in 8B9B. The additional overhead is due to the more complex table lookup procedure used by VHCC with respect to 8B9B for PS encoding and decoding, as shown in Figure 5.7 (for the decoding part only).

By contrast, the SBS execution time is significantly higher. This is probably due to the different nature of the SBS algorithm with respect to the others. In fact, the software stuff bit insertion performed by SBS is basically a bit-oriented operation, whereas both 8B9B and VHCC are codeword-oriented algorithms. Moreover, the implementation of both 8B9B and VHCC has been highly optimized, thus imparting additional bias to the comparison.

A somewhat opposite consideration can be done about footprint, though. Albeit the code footprint of SBS and VHCC are close to each other, the data footprint of VHCC is noticeably higher. This is due to its forward and reverse lookup tables, which are not needed by the bit-oriented SBS algorithm. Even if they greatly contribute to the high performance of VHCC in terms of execution time, they put it at a disadvantage in terms of data footprint. However, especially on

modern microcontrollers, the extra 384 bytes for the lookup tables are likely an affordable price to be paid for the added performance and flexibility of VHCC.

Going into a more detailed comparison between 8B9B and VHCC—referring back to Table 5.3 and the additional data presented in Chapter 3—it should be noted that, although the total code size of both the encoder and decoder modules more than doubled when going from one method to the other, most additional instructions are located either in loop setup or in PS handling. Therefore, these instructions are executed just once for each encoding or decoding request and their impact on performance is limited. By contrast, the size of the encoding and decoding loops only increased by about 12% and 15%, respectively. This can easily be justified by considering that: the higher complexity of the VHCC algorithm forced the compiler to allocate more registers and stack space for local variables and temporary storage. For this reason, the quality of code generation and optimization related to the loops worsened.

Summary

The bit stuffing mechanism of CAN implies transmission jitters that may lead to an unwanted (and sometimes unacceptable) worsening in the timing accuracy of distributed real-time control applications. Approaches like 8B9B are aimed at lessening this problem, by preventing stuff bits in the data field.

In this chapter, a theoretical foundation has been developed that provides two important benefits: first, it demonstrates that the 8B9B approach is the best choice in its class for what concerns the achievable information rate; second, it allows the fully automatic construction of a family of nested codebooks that supports efficient sub-byte encoding and decoding.

The new codebooks, along with some additional improvements, have been used to bring further notable improvements to the basic encoding scheme. The new scheme, called VHCC, is able to make full use of the room available in the data field, yet preserving the ability of completely preventing the occurrence of stuff bits in this field.

Measurements carried out on a prototype implementation of the related software modules show that their execution speed and footprint are still comparable to those achieved in the previous version, although the new enhanced scheme has a better encoding efficiency.

Chapter 6

Zero Stuff-Bits CRC (ZSC)

As discussed in Chapter 3 and Chapter 5, in order to reduce the bit stuffing jitter introduced at the physical layer of CAN, several approaches were defined. In particular, XOR-based solutions [69, 68, 65] operate by scrambling the content of the data field in software before the frame is sent, trying to reduce the number of stuff bits. More recent solutions, such as SBS [66], EEM [64], 8B9B, and VHCC are able to prevent the occurrence of stuff bits in the data field completely by using suitable encoding schemes. We will refer to the latter class of approaches as *Zero Stuff-bit Data* (ZSD). In particular, 8B9B was shown to provide the best encoding efficiency in its class.

The main drawback of all the above approaches is that, the occurrence of stuff bits is prevented only in the payload of the message. Two significant parts of the frame are left out, namely the header and the cyclic redundancy check (CRC). As explained earlier, the number of stuff bits that are inserted in the header is fixed, known in advance, and does not lead to communication jitters. Instead, the CRC field is more problematic. This is because its value is computed in hardware by the CAN controller at run time. Apparently, the system designer has no means to prevent (or just reduce) the number of stuff bits inserted in this part of the frame. As pointed out in Section 4.1, this implies that a residual jitter—which can be as high as 4 bit times—seems to be unavoidable, even with ZSD approaches. This is clearly unacceptable when a timing accuracy in the order of one μs or less is needed.

In this chapter a new mechanism is introduced, known as *Zero Stuff-bit CRC* (ZSC), that enhances ZSD encoding schemes (like 8B9B) by preventing the occurrence of stuff bits in the CRC. The resulting *Zero Stuff-bit* (ZS) code, which combines ZSD and ZSC, is able to avoid the insertion of stuff bits in every variable part of the CAN frame, hence it leads to truly fixed transmission times over the bus. When coupled with a suitable mechanism able to prevent bus contentions (e.g., a master-slave approach), this makes CAN communication timings completely deterministic, without having to resort to time-triggered paradigms.

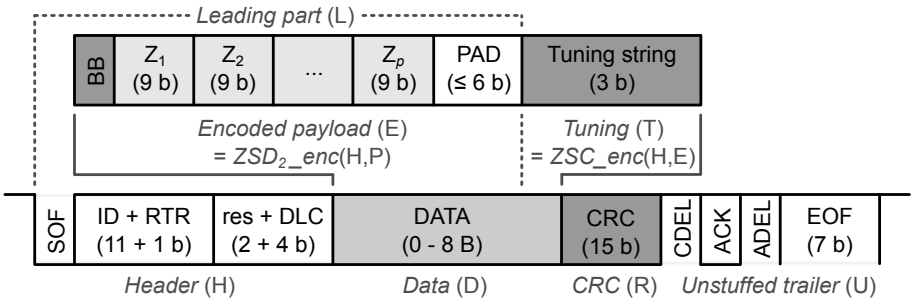

Figure 6.1. CAN frame format (11-bit id.) with 8B9B and ZSC encoding.

6.1 Formal Foundation

6.1.1 ZSD Properties

It is worth pointing out that, from a practical point of view, preventing stuff bits in the CRC field by means of the ZSC mechanism is mostly useless, unless countermeasures have been taken to remove all the sources of BS jitter from the other parts of the frame, too. In particular, ZSD encoding schemes can be adopted to this aim to cope with the data field. They satisfy the following two properties:

1. the encoded data field never includes primer sequences;
2. primer sequences are not found at the boundary between the frame header and the encoded data field.

To make jitter prevention easier in the CRC, a third property may be *optionally* taken into account, which requires that no more than a given number w_e of bits at the same value be found at the end of the encoded data field ($1 \leq w_e \leq 4$). The class of related encoding schemes will be denoted ZSD_{w_e}. For these classes, the property $w_1 < w_2 \Rightarrow \text{ZSD}_{w_1} \subseteq \text{ZSD}_{w_2}$ holds. As having more than 4 adjacent bits at the same value causes the insertion of a stuff bit, ZSD_4 actually coincides with ZSD.

In the following, ZSD_2 schemes to which 8B9B belongs will be considered explicitly. However, most results we obtained apply, with slight changes, to other approaches as well.

6.1.2 Definitions

The term *section* is used to refer to either a single field or a specific part of the frame comprising several adjacent fields. As shown in the lower part of Figure 6.1, every frame in CAN is made up of different sections: *Header* (H), *Data* field (D), *CRC* field (R) and *Unstuffed trailer* (U). In particular, *Data* field (D) is not interpreted by the CAN controller and includes information to be exchanged among devices over the network. It is built from the *original payload* (P), which is defined at the application level and is completely under control of the application itself. All mechanisms that prevent/lessen the effects of bit stuffing in CAN operate by encoding P in a suitable way before storing the result into D.

The nominal size in bits of a generic section Y is denoted n_Y: for instance, $n_H = 19$ (using standard 11-bit identifiers), $n_R = 15$, $n_U = 10$, while $n_D = 8 \cdot \mathrm{DLC}$ (where $0 \leq \mathrm{DLC} \leq 8$). Calligraphic letters refer to the sets of legal values (bit sequences) that sections can assume (before bit stuffing is applied). For convenience, section names are also used to denote a specific, legal value of that section. For instance, $\mathcal{Y}$ denotes the set of all legal values that section Y can assume and $Y \in \mathcal{Y}$ represents one of them. In general, $|\mathcal{Y}| \leq 2^{n_Y}$ and the strict inequality holds when some values of Y are illegal.

Let "≀" denote the *concatenation* operator between sections. For example, section $X \wr Y$ is made up of all the bits in X (in the original order) followed by all the bits in Y (in the original order). When sizes are concerned, $n_{X \wr Y} = n_X + n_Y$. By extension, the same operator will also be used to concatenate individual bits and bit sequences.

As proved in the following for specific conditions, part of the bits in D can be used to prevent the insertion of stuff bits in the CRC field. Such bits, located at the very end of the data field (as shown in Figure 6.1), are referred to as the *tuning* field (T) and are leveraged by the ZSC mechanism to "shape" the bit pattern in R properly. Quite obviously, this requires that D is not empty, that is, $\mathrm{DLC} > 0$. To this extent, it is worth remarking that neither ZSD nor ZSC apply to messages with no payload (i.e., data frames with $\mathrm{DLC} = 0$ and remote frames).

The part of D allotted to the user information—i.e., with the exclusion of the tuning field—is referred to as the *encoded payload* (E) and carries (in our case encoded through a ZSD scheme) the original payload P. This implies that $D = E \wr T$, which means that the number of bits in D that are actually available for E is $n_E = 8 \cdot \mathrm{DLC} - n_T$. Together, header and encoded payload build up the *leading part* of the frame (L), defined as $L = H \wr E$.

6.1.3 CRC Computation

The CRC computation is carried out in hardware by the CAN controller according to a predefined scheme. For this reason it is just impossible to rely on approaches similar to 8B9B (or any other proposed in the literature) in order to prevent the occurrence of stuff bits in the CRC field.

The portion of the frame that is covered by the CRC is referred to as *message* (M) and is given by

$$M = H \wr D = H \wr E \wr T = L \wr T \ , \tag{6.1}$$

while the whole CAN *frame* (F) can be expressed as $F = M \wr R \wr U$. Of course, we are interested only in the portion of the frame that is affected by bit stuffing, namely, $M \wr R$.

Concerning CRC computation, every frame section can be seen as a *polynomial*. The same symbol is used in the following to denote both a section and the corresponding polynomial, that is, polynomial $Y(x)$ corresponds to section Y. The degree of $Y(x)$ is $n_Y - 1$ while its coefficients y_i ($0 \leq i \leq n_Y - 1$) are the same as the bits in Y.

For instance, the CRC field can be seen as a bit sequence

$$R = r_{n_R-1} \wr r_{n_R-2} \wr \ldots \wr r_1 \wr r_0 \ , \tag{6.2}$$

while the corresponding polynomial is

$$R(x) = \sum_{i=0}^{n_R-1} r_i \cdot x^i \ . \tag{6.3}$$

From (6.1) the polynomial associated to M is

$$M(x) = L(x) \cdot x^{n_\mathrm{T}} + T(x) \ . \tag{6.4}$$

The CRC is defined as the remainder of the division between $M(x) \cdot x^{n_\mathrm{R}}$ and the generator polynomial $G(x)$

$$M(x) \cdot x^{n_\mathrm{R}} = Q(x) \cdot G(x) + R(x) \ , \tag{6.5}$$

or, using the mod operator (in modulo 2 arithmetic)

$$R(x) = M(x) \cdot x^{n_\mathrm{R}} \bmod G(x) \ . \tag{6.6}$$

From a practical viewpoint, the numerator is obtained by appending 15 bits at the "0" value to the end of M. Let $c(\cdot)$ denote the function that computes the CRC in CAN. Then,

$$\mathrm{R} = c(\mathrm{M}) = c(\mathrm{L} \wr \mathrm{T}) \ . \tag{6.7}$$

6.1.4 8B9B Encoding

Although ZSC can be profitably paired to any ZSD encoding scheme, in the following attention is focused specifically on 8B9B (or, more generally, on the class of codes it belongs to). In these cases, a codebook $\mathcal{G}$ is used for translating every single byte P_i of the original payload ($\mathrm{P} = \mathrm{P}_1 \wr \ldots \wr \mathrm{P}_p$, where $p \geq 1$ is the size in bytes of P) to a distinct 9 b pattern $\mathrm{Z}_i \in \mathcal{G}$. Other such codebooks exist, that feature different properties. For instance, the one used in VHCC satisfies a nesting property, which can be exploited to encode sub-byte values.

Such translation process can be modeled as a function $f(\cdot)$ so that $\mathrm{Z}_i = f(\mathrm{P}_i)$. From a practical point of view the codebook $\mathcal{G}$ in 8B9B corresponds to a forward lookup table (FLT), which includes 256 entries (memory optimizations are possible thanks to its symmetry, see Section 3.3.2). On the receive side, a reverse lookup table (RLT) is required to get back each single byte of the original payload, that is, $\mathrm{P}_i = f^{-1}(\mathrm{Z}_i)$.

Patterns Z_i obtained from translations are then concatenated, in the original order, and become part of E. As it was done in Chapter 5, let us denote with $\mathcal{G}_{n_\mathrm{Z},w_l,w_b,w_t}$ the set of all the binary strings of length n_Z, which have at most w_l leading bits at the same value, at most w_b consecutive inner bits at the same value, and at most w_t trailing bits at the same value. If $\mathcal{G} \subseteq \mathcal{G}_{9,2,4,2}$ (as in the case of the 8B9B family of encoding schemes) then codewords do not include more than 4 consecutive inner bits at the same level and no more than 2 bits at the same level at both ends. It can be easily shown that the same property holds for the bit sequence obtained by concatenating codewords Z_i (as long as this sequence is not empty), that is, $\mathrm{Z}_1 \wr \mathrm{Z}_2 \wr \ldots \wr \mathrm{Z}_p \in \mathcal{G}_{9 \cdot p,2,4,2}$ (p also corresponds to the number of 9 b patterns in E).

Depending on the value of DLC (that does not vary for a given message stream), a *break bit* (BB) may be included by 8B9B at the beginning of the data field, which is set at the opposite value than the last bit of the DLC field. Moreover, as shown in the upper part of Figure 6.1, n_D is a multiple of 8 b whereas each codeword is 9 b long. Hence, generally D is not completely filled by $\mathrm{BB} \wr \mathrm{Z}_1 \wr \mathrm{Z}_2 \wr \ldots \wr \mathrm{Z}_p$ and T. The remaining space, which lies between the last codeword Z_p and the tuning field T, is referred to as *padding* (PAD). Its value, which is effectively unused, has to

be chosen so that it does not cause the insertion of any stuff bit. To this extent, an alternating bit pattern is sufficient, as already discussed in Section 3.2.1.

The resulting bit sequence for the encoded payload, whose general structure is $\mathrm{E} = \mathrm{BB} \wr \mathrm{Z}_1 \wr \mathrm{Z}_2 \wr \ldots \wr \mathrm{Z}_p \wr \mathrm{PAD}$, satisfies the *first* ZSD property mentioned in Section 6.1.1. BB is used to prevents propagation of stuff bits from H to E when necessary. Hence, primer sequences cannot appear at the boundary between these two sections, and the *second* ZSD property holds as well. If the filling pattern is selected so that the first bit of PAD is at the opposite value than the last bit of Z_p, no more than two bits at the same value can ever be found at the end of L, even in the limit cases when $n_{\mathrm{PAD}} = 1$ or 0. Therefore, also the *third* property is true for $w_e = 2$. This implies that 8B9B fulfills all the requirements for ZSD_2 codes. Consequently, such an approach (and the like) can be effectively adopted for practical implementations of ZSC.

The condition that a specific encoded payload E, following a given header H, satisfies the ZSD_{w_e} properties, is generically expressed as $\mathrm{E} \in \mathcal{E}_{\mathrm{H}}^{(\mathrm{ZSD}_{w_e})}$ (only the last bit of H is actually relevant in the case of 8B9B). The resulting leading part L will be said to belong to $\mathcal{L}^{(\mathrm{ZSD}_{w_e})}$, defined as the set of all the legal values section L of the frame may assume when the payload is encoded with a ZSD_{w_e} scheme.

6.1.5 Counting Stuff Bits

In order to detect the occurrence of a primer sequence, also previously inserted stuff bits have to be taken into account. This may lead to a domino effect, which in the worst case results in the insertion of one stuff bit every time 4 bits at the same value are found in the original frame [68]. As a consequence, bit stuffing in any section of the frame is affected by all preceding fields. This means that, in theory, in order to evaluate the exact number of stuff bits that are added to the CRC, sections H, E, and T have to be considered.

Let $h(\mathrm{Y})$ be a function that returns the exact number of stuff bits that are added to the bit sequence Y when considered alone—that is, when the contribution of any field that (possibly) precedes Y is not taken into account. Moreover, let $h_{\mathrm{X}}(\mathrm{Y})$ be a function that returns the actual number of stuff bits that are specifically added to section Y when preceded by the bit sequence X, that is,

$$h_{\mathrm{X}}(\mathrm{Y}) \triangleq h(\mathrm{X} \wr \mathrm{Y}) - h(\mathrm{X}) \ . \tag{6.8}$$

It is worth noting that $h(\mathrm{X} \wr \mathrm{Y})$ is not necessarily the same as $h(\mathrm{X}) + h(\mathrm{Y})$, because the final bits in X may contribute to the creation of a primer sequence across the boundary with Y. The difference (if any) between $h(\mathrm{Y})$ and $h_{\mathrm{X}}(\mathrm{Y})$ is due to X, which represents the part of the frame that precedes Y.

$h(\mathrm{F})$ represents the overall number of stuff bits that are added to F by the CAN controller (in hardware). It can be expressed as

$$h(\mathrm{F}) = h(\mathrm{H}) + h_{\mathrm{H}}(\mathrm{E}) + h_{\mathrm{L}}(\mathrm{T} \wr \mathrm{R}) \ . \tag{6.9}$$

Obviously, no stuff bits are added to the unstuffed trailer. The header is the first field in the frame. Under the (reasonable) assumption that H does not change at run-time for a given message stream, the number of stuff bits added to this section, denoted h_{H}, is fixed ($h_{\mathrm{H}} \geq 0$). As

a consequence, no jitter on transmission times is due to H. By means of a careful selection of message identifiers (but not for every possible size of the payload) h_{H} can often be reduced to 0.

If a ZSD scheme is adopted, no stuff bits at all can be inserted in the encoded payload irrespective of the header, that is,

$$h_{\mathrm{H}}(\mathrm{E}) = h(\mathrm{E}) = 0, \quad \forall \mathrm{H} \in \mathcal{H}, \forall \mathrm{E} \in \mathcal{E}_{\mathrm{H}}^{(\mathrm{ZSD})} \quad . \tag{6.10}$$

Therefore, the overall jitter on frame reception due to BS depends only on stuff bits added to the fields that follow the encoded payload, that is, the tuning and CRC fields

$$h(\mathrm{F}) = h_{\mathrm{H}} + h_{\mathrm{L}}(\mathrm{T} \wr \mathrm{R}), \quad \forall \mathrm{L} \in \mathcal{L}^{(\mathrm{ZSD})} \quad . \tag{6.11}$$

In order to make sure that the algorithm used to select the value of T in ZSC can work properly, bit stuffing in T and R must be decoupled from the preceding fields in the frame (i.e., L). While a second break bit located right before T (irrespective of the actual encoding of E) may suffice, exploiting the peculiar properties of ZSD_2 codes is a better solution. Moreover, not every value of T is legal in order not to add any stuff bits.

Let $\mathcal{T}$ be a subset of the values T may assume ($n_{\mathrm{T}} > 1$), which satisfies the following properties

$$\mathcal{T} \subseteq \begin{cases} \{01, 10\}, & n_{\mathrm{T}} = 2 \\ \mathcal{G}_{n_{\mathrm{T}},2,4,4}, & n_{\mathrm{T}} > 2 \end{cases} \quad . \tag{6.12}$$

The first property means that, in the case T includes only two bits, they are not allowed to be at the same value. Instead, in the second property it is indicated that T must not include primer sequences and no more than two leading bits can be found at the same level.

It can be proved that, if $\mathrm{L} \in \mathcal{L}^{(\mathrm{ZSD}_2)}$ and $\mathrm{T} \in \mathcal{T}$, then the number of stuff bits added to the tuning and CRC fields equals $h(\mathrm{T} \wr \mathrm{R})$, that is,

$$h_{\mathrm{L}}(\mathrm{T} \wr \mathrm{R}) = h(\mathrm{T} \wr \mathrm{R}) \quad . \tag{6.13}$$

In fact, because of (6.12) and the properties of ZSD_2 codes, no more than 4 bits at the same level can be found at the boundary between L and T. Therefore, primer sequences are prevented there. This implies that the value $h_{\mathrm{L}}(\mathrm{T} \wr \mathrm{R})$ depends only on the values of T and R and not on the (preceding) leading part L of the frame. Under the above hypotheses, $h(\mathrm{F})$ can be obtained as

$$h(\mathrm{F}) = h_{\mathrm{H}} + h(\mathrm{T} \wr \mathrm{R}) \quad . \tag{6.14}$$

Basically, (6.13) implies that stuff bits are completely prevented in the part of the frame that follows L, provided that E is ZSD_2 encoded, T is (suitably) selected in $\mathcal{T}$, and no primer sequence is found in $\mathrm{T} \wr \mathrm{R}$. According to (6.7), R depends on L too. Hence, section L still affects (indirectly) the value of $h(\mathrm{T} \wr \mathrm{R})$ in (6.13) and hence $h(\mathrm{F})$ in (6.14).

6.2 Prevention of Bit Stuffing in the CRC

The CRC cannot be controlled directly, as it is calculated in hardware. Conversely, size and value of the tuning field can be selected in software. As conjectured above, T could be computed at

runtime—before the data field is copied into the CAN controller—so as to prevent primer sequences from occurring in the CRC.

The ZSC feasibility depends on the existence of a function $z(\cdot)$ that, starting from L, determines a value for T able to prevent stuff bits in the CRC (and T as well). That is to say

$$\forall \mathrm{L} \in \mathcal{L}^{(\mathrm{ZSD})}, \exists \mathrm{T} \in \mathcal{T} \;\mid\; \mathrm{T} = z(\mathrm{L}) \;\; \wedge \;\; h_{\mathrm{L}}(\mathrm{T} \wr \mathrm{R}) = 0 \;\; . \tag{6.15}$$

The cardinality of $\mathcal{T}$ (and, consequently, $\mathcal{E}$ and $\mathcal{L}$) is directly affected by n_{T}. However, this is not highlighted explicitly in the notation in order to keep it simple. It can be proved that, if (6.15) holds for a given value of n_{T}, then it certainly holds for any higher values (since $\mathcal{L}$ consequently shrinks).

Before trying to conceive an efficient algorithm for $z(\cdot)$, the minimum size of T that makes the above approach feasible (if any) has to be evaluated. In the following, it will be shown that 3 bits are sufficient to this purpose. In theory, this could be (trivially) demonstrated by considering all possible combinations of bits for the header, encoded payload, and tuning fields. In particular, for any size n_{T} of T and for any value $\mathrm{L} \in \mathcal{L}^{(\mathrm{ZSD})}$, a set $\mathcal{T}_{\mathrm{L}}^{\langle n_{\mathrm{T}} \rangle}$ is determined as

$$\mathcal{T}_{\mathrm{L}}^{\langle n_{\mathrm{T}} \rangle} \triangleq \{\mathrm{T} \;\mid\; h(\mathrm{L} \wr \mathrm{T} \wr c(\mathrm{L} \wr \mathrm{T})) = h_{\mathrm{H}}\} \;\; . \tag{6.16}$$

Each set $\mathcal{T}_{\mathrm{L}}^{\langle n_{\mathrm{T}} \rangle}$ may include zero or more elements. If the condition

$$\exists n_{\mathrm{T}} \;\mid\; \mathcal{T}_{\mathrm{L}}^{\langle n_{\mathrm{T}} \rangle} \neq \emptyset, \;\; \forall \mathrm{L} \in \mathcal{L}^{(\mathrm{ZSD})} \tag{6.17}$$

holds, then, generally speaking, the tuning field can be used to prevent stuff bits in the CRC completely. In this case, the minimum value of n_{T} that satisfies (6.17), denoted $\min(n_{\mathrm{T}})$, is the optimal choice and would demonstrate the claim.

Unfortunately, this approach cannot be adopted in practice. This is because, set $\mathcal{L}^{(\mathrm{ZSD})}$ is too large to carry out an exhaustive search. In fact, although $\left|\mathcal{L}^{(\mathrm{ZSD})}\right| < |\mathcal{M}| / 2^{n_{\mathrm{T}}}$, $\mathcal{M}$ includes more than 2^{11+64} patterns for standard identifiers and even more for extended ones. In the following, some techniques are described aimed at reducing dramatically the state space and, consequently, the search complexity.

6.2.1 Partial Remainders

The ZSC mechanism relies on the *linearity* of CRC codes. In particular, the value of the R field in the frame sent over the bus can be decoupled into two independent contributions, which depend on the L and T portions of message M, respectively. The following partial remainders can be defined by carrying out independent divisions for the sections that make up M:

$$\begin{aligned} R_{\mathrm{L}}(x) &\triangleq \left(L(x) \cdot x^{n_{\mathrm{T}}+n_{\mathrm{R}}}\right) \mod G(x) \;\; , \\ R_{\mathrm{T}}(x) &\triangleq \left(T(x) \cdot x^{n_{\mathrm{R}}}\right) \mod G(x) \;\; . \end{aligned} \tag{6.18}$$

It is possible to prove that the R field can be evaluated from the partial remainders related to the L and T sections, that is,

$$\mathrm{R} = c\left(\mathrm{L} \wr \mathrm{T}_0\right) \oplus c\left(\mathrm{T}\right) \;\; , \tag{6.19}$$

where "⊕" denotes bitwise EXOR and T_0 consists of a sequence of n_T bits at the 0 value.

To this extent, it should be noted that the following properties hold for $R_\mathrm{L}(x)$ and $R_\mathrm{T}(x)$

$$\begin{aligned} L(x) \cdot x^{n_\mathrm{T}+n_\mathrm{R}} &= Q_\mathrm{L}(x) \cdot G(x) + R_\mathrm{L}(x) \ , \\ T(x) \cdot x^{n_\mathrm{R}} &= Q_\mathrm{T}(x) \cdot G(x) + R_\mathrm{T}(x) \ . \end{aligned} \tag{6.20}$$

By summing these contributions, from (6.4) we find

$$M(x) \cdot x^{n_\mathrm{R}} = [Q_\mathrm{L}(x) + Q_\mathrm{T}(x)] \cdot G(x) + R_\mathrm{L}(x) + R_\mathrm{T}(x) \ . \tag{6.21}$$

As the degree of polynomials $R_\mathrm{L}(x)$ and $R_\mathrm{T}(x)$ is certainly lower than $G(x)$, the same property holds for their sum as well. This means that $R_\mathrm{L}(x) + R_\mathrm{T}(x)$ in (6.21) equals $R(x)$ in (6.5).

From (6.18), working on bit sequences and because of (6.6) and (6.7), $\mathrm{R_L} = c\,(\mathrm{L} \wr \mathrm{T}_0)$ and $\mathrm{R_T} = c\,(\mathrm{T})$. Equation (6.19) is proved by remembering that *additions* among polynomials in modulo 2 arithmetic correspond to bitwise EXOR operations on the corresponding bit patterns.

6.2.2 Zero Stuff-Bit CRC Mechanism

In this section a *sufficient condition* is determined for the feasibility of the ZSC mechanism. In particular, the minimum size of the T field, $\min(n_\mathrm{T})$, which ensures that no stuff bit will ever be inserted in $\mathrm{T} \wr \mathrm{R}$ when $\mathrm{L} \in \mathcal{L}^{(\mathrm{ZSD}_2)}$, is 3 bits.

In order to demonstrate the above claim, exhaustive space state analysis can be carried out on a proper subset of states, small enough to make the analysis computable in a finite (and short) time. The reasoning below is repeated for increasing values of n_T from 2 on.

Instead of considering separately every possible value for L, only its contribution $\mathrm{R_L}$ to the CRC is taken into account, thanks to (6.19). All possible values $\mathrm{R}_k \in \mathcal{R}$ have to be considered for $\mathrm{R_L}$, where $\mathcal{R}$ is the set of values R can assume.

Let $\mathcal{L}_{\mathrm{R}_k}$ be a set of L values defined as

$$\mathcal{L}_{\mathrm{R}_k} \triangleq \left\{ \mathrm{L} \in \mathcal{L}^{(\mathrm{ZSD}_2)} \ \middle| \ c(\mathrm{L} \wr \mathrm{T}_0) = \mathrm{R}_k \right\} \ . \tag{6.22}$$

Sets $\mathcal{L}_{\mathrm{R}_k}$ are disjoint, i.e., $\mathrm{R}_x \neq \mathrm{R}_y \Rightarrow \mathcal{L}_{\mathrm{R}_x} \cap \mathcal{L}_{\mathrm{R}_y} = \emptyset$, and their union equals the universe, $\bigcup_{\mathrm{R}_k \in \mathcal{R}} \mathcal{L}_{\mathrm{R}_k} = \mathcal{L}^{(\mathrm{ZSD}_2)}$.

For every pattern R_k, all possible values $\mathrm{T}_j \in \mathcal{T}$ of the tuning field have been considered. Because of (6.19),

$$c(\mathrm{L} \wr \mathrm{T}_j) = c(\mathrm{L} \wr \mathrm{T}_0) \oplus c(\mathrm{T}_j) = \mathrm{R}_k \oplus c(\mathrm{T}_j), \quad \forall \mathrm{L} \in \mathcal{L}_{\mathrm{R}_k} \ . \tag{6.23}$$

In order to meet the assumptions of (6.13), only patterns T_j that satisfy (6.12) have to be taken into account. This is clearly a limiting assumption. Since the CAN controller, when transmitting a frame, actually knows L, discarding in advance the values of T not included in $\mathcal{T}$ is not strictly necessary. Nevertheless, we are looking for a sufficient condition that ensures complete jitter prevention, and operating in this way is necessary to make exhaustive search feasible.

In Table 6.1, the value of $c(\mathrm{T}_j)$ is shown for every pattern T_j, in the specific case when $n_\mathrm{T} = 3$. The case $n_\mathrm{T} = 2$ can be dealt with easily, by using just the first two values of $c(\mathrm{T}_j)$ from the table—due to a known property of CRC calculation that leading zeros do not change the final result. For

j	T_j	$c(\mathrm{T}_j)$ (binary)	$c(\mathrm{T}_j)$ (hex)
1	001	100 0101 1001 1001	0x4599
2	010	100 1110 1010 1011	0x4eab
3	011	000 1011 0011 0010	0x0b32
4	100	101 1000 1100 1111	0x58cf
5	101	001 1101 0101 0110	0x1d56
6	110	001 0110 0110 0100	0x1664

Table 6.1. Contribution of the tuning field to the CRC.

every $\langle \mathrm{R}_k, \mathrm{T}_j \rangle$ pair, R is computed from (6.23) and the presence of stuff bits in the tuning and CRC fields is checked by evaluating $h_{\mathrm{L}}(\mathrm{T}_j \wr \mathrm{R})$. Because of (6.13), only $\mathrm{T}_j \wr \mathrm{R}$ is relevant to this extent.

The space of $\langle \mathrm{R}_k, \mathrm{T}_j \rangle$ pairs was explored exhaustively so as to determine whether or not the following property holds

$$\forall \mathrm{R}_k \in \mathcal{R}, \exists \mathrm{T}_j \in \mathcal{T} \;\mid\; h(\mathrm{T}_j \wr (\mathrm{R}_k \oplus c(\mathrm{T}_j))) = 0 \;. \tag{6.24}$$

The size of the space to be explored does not exceed $2^{n_{\mathrm{T}}+n_{\mathrm{R}}}$. When the tuning field includes 3 bits, this leads to $6 \cdot 2^{15}$ states, which can be easily dealt with in a very short time on any modern computer.

We found that, if $n_{\mathrm{T}} = 2$, some messages exist in which one or more stuff bits are still added to $\mathrm{T} \wr \mathrm{R}$, whatever the value chosen for T. Due to the way this result was obtained, also the case $n_{\mathrm{T}} = 1$ (that, indeed, needs to be dealt with in a slightly different manner) is ruled out for sure. On the contrary, when $n_{\mathrm{T}} = 3$, then at least one T_j always exists such that, irrespective of the specific content of H and E, no stuff bits at all are added to $\mathrm{T}_j \wr \mathrm{R}$. The above reasoning fully demonstrates the initial claim about the sufficient condition and, as a consequence, ZSC feasibility.

As a summary, it should be noted that:

1. the number of stuff bits in H is fixed and known;
2. ZSD schemes prevent stuff bits in E completely;
3. ZSC, when combined with ZSD, prevents stuff bits completely in $\mathrm{E} \wr \mathrm{T} \wr \mathrm{R}$.

Consequently, the overall jitter in CAN due to BS can be reduced to 0 by a suitable ZS algorithm that encodes P in D.

6.3 Implementation and Experimental Results

The practical implementation of ZSC had two main goals. The first intent was to verify by experiment that the CAN frames generated by a combination of a suitable ZSD_2 encoder plus ZSC incur no bit stuffing at the data-link level, as a way to double-check the theoretical proof given in Section 6.2.

Secondly, effort has been put in optimizing the prototype codec so as to reduce both the amount of jitter it injects into the communication path, and its processing time. Results show that the proposed approach is feasible also in practice, and its adoption neither disrupts CAN communication performance, nor impacts the memory requirement of the system in a significant way.

```
1)  ZS_enc(in H, in P) → out D
2)      E := ZSD2_enc(H, P);
3)      T := ZSC_enc(H, E);
4)      D := E ≀ T;
5)      return D;

6)  ZSD2_enc(in H, P) → out E
7)      E := BB; // break bit depends on DLC (i.e., H)
8)      for i = 1 to p // size in bytes of the payload
9)          E := E ≀ f(P_i); // translate one byte with FLT
10)     E := E ≀ PAD; // padding depends on E
11)     return E;

12) ZSC_enc(in H, E) → out T
13)     R_L := c(H ≀ E ≀ 000_2);
14)     for each T_j ∈ {001_2, 010_2, 011_2, 100_2, 101_2, 110_2}
15)         R_j := R_L ⊕ c(T_j);
16)         if h(T_j ≀ R_j) = 0 then T := T_j;
17)     return T;  // the last T value is taken as z(H ≀ E)
```

Figure 6.2. Operation of the ZS encoder.

As discussed in the following, performance evaluation experiments carried out on large sets of data revealed that the total communication jitter (1.08 μs in the worst case, but it could be further improved) is 40 times lower than in plain CAN (22 bit times, corresponding to 44 μs when the CAN bit rate is 500 kbps). Actually, this is entirely due to data processing jitter in the codec and residual jitter at the interface between the CPU and the CAN controller [9].

6.3.1 Implementation

As discussed before, operation of the ZS encoder, which includes both ZSD_2 and ZSC, is described by the algorithm shown in Figure 6.2. In practice, all experiments have been performed by means of the testbed shown in Figure 6.3, which faithfully reproduces the whole software stack needed for ZS CAN communication. The 8B9B codec discussed in Chapter 3, slightly modified as described in the following, has been chosen as ZSD_2. The main testbed components (white blocks in Figure 6.3) are the following:

- A *test harness* schedules the experiment, and then collects and stores test results for later analysis.
- A *test data generator* produces the application-level test data to be used for the experiments.
- The *8B9B* and *ZSC encoders* encode the test data and build the data field of the CAN frames to be transmitted conforming to the algorithm shown in Figure 6.2.

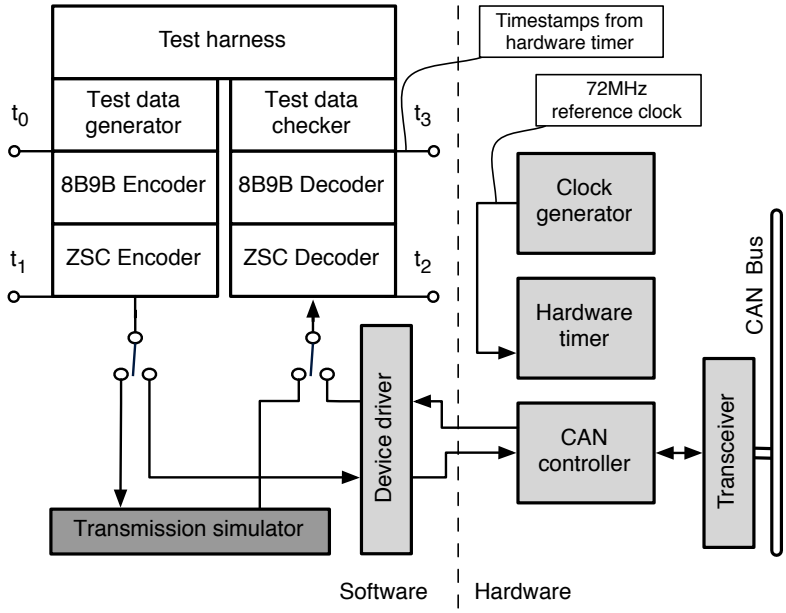

Figure 6.3. Experimental testbed.

- The *8B9B* and *ZSC decoders* recover the test data from the received CAN frames. The ZSC decoder is actually very thin and must just discard the tuning field before feeding the incoming data into the ZSD_2 decoder.
- The *test data checker* verifies that the encoding/decoding process did not modify the test data in any way.

The most important difference between the general algorithm shown in Figure 6.2 and its practical implementation is that, for the sake of efficiency, the boundary between partial and complete CRC calculation (corresponding to variables $\mathrm{R_L}$ and R_j in the algorithm) has been moved to the position between the last byte of D and its predecessor, instead of being placed between E and T_j. This optimization does not affect results because (6.19) is actually valid regardless of where the boundary is. By analogy with (6.19), when L is split at a byte boundary as $\mathrm{L} = \mathrm{L}_1 \wr \mathrm{L}_2$, it is

$$\mathrm{R} = c(\mathrm{L}_1 \wr 00000000_2) \oplus c(\mathrm{L}_2 \wr \mathrm{T}) \ , \tag{6.25}$$

and the width of $\mathrm{L}_2 \wr \mathrm{T}$ is one byte. To facilitate the CRC calculation, a predefined CRC lookup table with $2^8 = 256$ two-byte entries, which stores $c(x)$ for any one-byte value x, is used in the implementation.

6.3.2 Software- and Hardware-Based Correctness Checks

When the testbed is configured for *software-based* checks, the ZS modules exchange CAN frames through a *transmission simulator* (dark grey block in Figure 6.3). It implements CRC calculation, bit-stuffing and, in general, all the other parts of the CAN data-link protocol specification needed

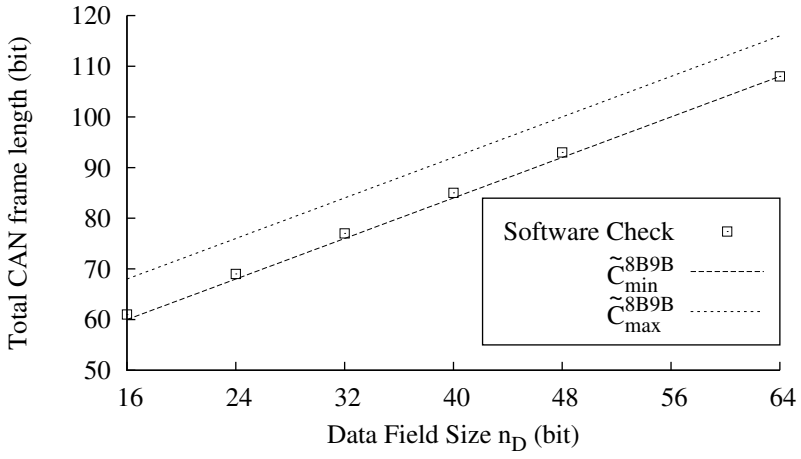

Figure 6.4. Total CAN frame length, as a function of n_D, measured in the software-based correctness check (10^9 samples for each experiment).

to compute the final length of a CAN frame as it is transmitted on wire. Those lengths are collected by the test harness.

The software-based checks have been carried out with the test data generator configured to produce 10^9 uniformly-distributed, random data items for each length from 1 to 6 bytes, corresponding to a data field size n_D from 16 to 64 bits after encoding. In fact, due to the 8B9B algorithm properties, n_D is generally one byte longer than the original data item length. The original payload length of 6 bytes is an exception, because it produces a sequence of 54 8B9B-encoded bits. If that sequence were stored in a 56-bit data field, there would not be enough room for the tuning field, because $\min(n_T) = 3$. As a consequence, extra room should be added to make tuning possible. Since n_D must be a multiple of 8 due to CAN frame format constraints, it must be $n_D = 64$ in this case.

Results of the software-based correctness check are shown in Figure 6.4 (□ diagram). The total CAN frame length measured by the transmission simulator during the experiments is plotted as a function of n_D. The sample variance is not shown, because it was zero in all experiments. In other words, the total CAN frame length was constant for any given n_D, regardless of the data being transmitted, thus confirming that the ZS encoder works as intended.

As a further reference, two quantities $\widetilde{C}^{8B9B}_{min}$ and $\widetilde{C}^{8B9B}_{max}$ have also been plotted in Figure 6.4. They are defined as the theoretical lower and upper bounds of the CAN frame length taking into account BS, when 8B9B is used alone. As 8B9B can completely prevent stuff bits in the data field, the difference $\widetilde{C}^{8B9B}_{max} - \widetilde{C}^{8B9B}_{min}$ represents the worst-case transmission time variation that may be introduced by the frame header and CRC fields. In the best case, no stuff bits are added, and hence, for CAN messages with standard 11-bit identifiers, it is

$$\widetilde{C}^{\,8B9B}_{min} = 19 + n_D + 15 + 10 = 44 + n_D \ , \tag{6.26}$$

where the value of 44 b represents the combined size of the frame header (19 b), the CRC (15 b), and the trailer (10 b).

On the other hand, the worst-case sequence concerning bit stuffing is made up of 5 b at `0`, followed by an alternating pattern made up of 4 b at `1` and 4 b at `0`. Therefore, as shown in Chapter 4, the maximum number $h_{\max}(w, l)$ of stuff bits that can be inserted in a message fragment of length l bits—when it is preceded by at most w consecutive bits ($0 \leq w \leq 4$) at the same value as the leading bit of the fragment—is

$$h_{\max}(w, l) = \left\lfloor \frac{w + l - 1}{4} \right\rfloor \ . \tag{6.27}$$

As a consequence, $\widetilde{C}_{\max}^{8B9B}$ can be calculated as

$$\widetilde{C}_{\max}^{8B9B} = \widetilde{C}_{\min}^{8B9B} + h_{\max}(0, 19) + h_{\max}(2, 15) = 44 + n_D + 4 + 4 = 52 + n_D \ . \tag{6.28}$$

In the previous formula, $h_{\max}(0, 19)$ represents the worst-case number of stuff bits added to the header (since the SOF bit at `0` is always preceded by a bit at `1`) and $h_{\max}(2, 15)$ gives the worst-case number of stuff bits added to the CRC (because, due to the 8B9B properties, the CRC can be preceded at worst by 2 consecutive bits at the same value as the leading bit of the CRC itself).

By comparing (6.26) and (6.28), the worst-case message length variation $\widetilde{C}_{\max}^{8B9B} - \widetilde{C}_{\min}^{8B9B}$ that may lead to jitter in transmission is therefore at most 8 b (4 b in the header and another 4 b in the CRC), when considering standard 11-bit identifiers.

The experimental results shown in Figure 6.4 reveal that this jitter has been completely removed by the ZS encoder. The only difference of 1 bit between the actual frame length and $\widetilde{C}_{\min}^{8B9B}$ for $n_D = 16, \ldots, 48$ is due to the *fixed* amount of bit stuffing (not jitter) introduced in the message header for the given combinations of the message identifier chosen for the experiments ($2AA_{16}$) and the DLC field. For the same reason, the 1 bit difference is not there when $n_D = 64$.

In order to further investigate the correctness of the codec and evaluate its performance, the testbed shown in Figure 6.3 can also be configured to perform *hardware-based* checks, by enabling the light gray components. With respect to the software-based configuration, two remarkable differences exist:

1. The transmission simulator is replaced by a *device driver* for a hardware CAN controller.

2. A *hardware timer* with a resolution of 13.9 ns (corresponding to a 72 MHz clock frequency) is used to timestamp each test message at points $t_0, \ldots, t_3$ as shown in the figure. t_1 and t_2 are taken just before (after) transmitting (retrieving) a CAN message to (from) the CAN controller, while t_0 and t_3 are placed before encoding starts and after decoding ends.

For hardware-based checks, the whole system has been implemented on an LPC 2468 [71, 70] single-chip microcontroller and the CAN controller has been configured to run at a line speed of 500 kbps in self-reception mode.

The timestamp points just introduced are suitable for a variety of measurements, concerning both correctness and performance. In the following, the discussion will be focused only on correctness, whereas performance-related measurements will be discussed in Section 6.3.3.

When correctness is considered, the difference $t_2 - t_1$ is very important, because it represents the actual end-to-end CAN frame transfer time. The results are shown in Figure 6.5 and Table 6.2 (both on the next page). In the figure, the minimum and maximum value of $t_2 - t_1$ measured for

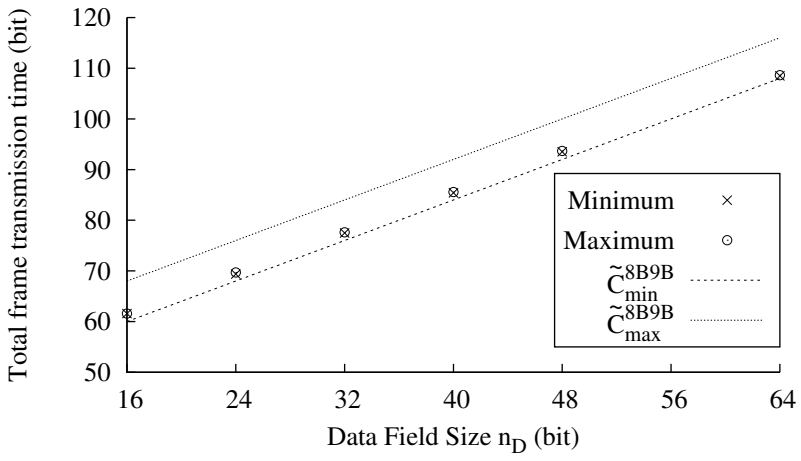

Figure 6.5. Min./max. CAN frame transfer time $t_2 - t_1$, as a function of n_D, from the hardware-based correctness check (10^7 samples for each experiment).

	Frame transmission time ($t_2 - t_1$)			
n_D	Average (μs)	Min (μs)	Max (μs)	Jitter (μs)
16	123.16	123.16	123.16	0.00
24	139.09	138.75	139.36	0.61
32	155.14	154.94	155.25	0.31
40	171.10	170.83	171.14	0.31
48	187.12	187.03	187.33	0.30
64	217.12	216.97	217.28	0.31

Table 6.2. Hardware-based CAN frame transfer time and residual jitter (10^7 samples for each value of n_D, CAN line rate of 500 kbps).

10^7 random test data items corresponding to a given value of n_D are plotted. They have been normalized to bit periods, for easier comparison with the results of software-based correctness check in Figure 6.4. The table contains more detailed information about the residual jitter, in numeric form.

Experimental results further confirm the correctness of the ZSC proposal. As it can be seen by comparing Figures 6.4 and 6.5, the actual end-to-end CAN frame transfer time is always the same as the theoretical value. At the same time—as listed in Table 6.2—the residual jitter is well below one bit time (that corresponds to 2 μs in the experiments being discussed). Hence, it cannot be caused by the bit stuffing mechanism, which would introduce an amount of jitter that is a multiple of the bit time itself. Rather, this value is totally compatible with what was shown to be the residual jitter at the interface between the CPU and CAN controller on the same kind of hardware [9] and using the same controller interface. It is worth remarking that, as mentioned in the same paper, this jitter is neither bit-rate dependent nor n_D dependent. It could probably be further reduced by upgrading the hardware.

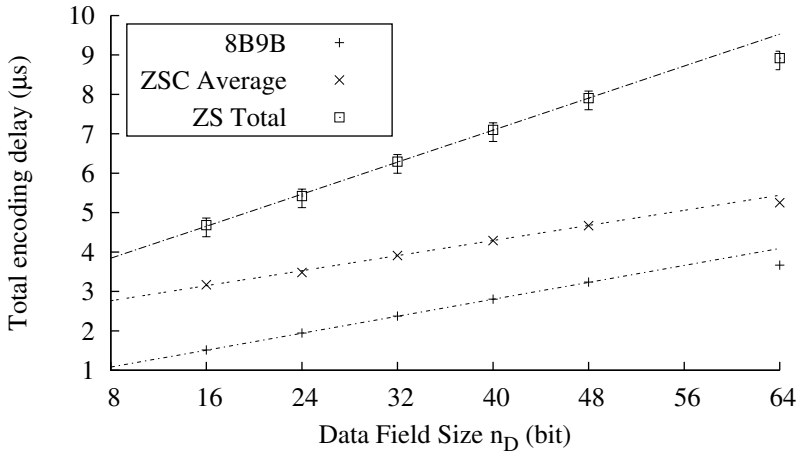

Figure 6.6. Encoding delay and jitter as a function of n_D (10^7 samples for each experiment).

6.3.3 Performance Evaluation

Performance-oriented measurements were focused on the ZS encoding and decoding time, as they are interesting from the application viewpoint. They correspond to $t_1 - t_0$ and $t_3 - t_2$ in Figure 6.3, respectively. Moreover, the contribution to them made by the 8B9B and ZSC modules was evaluated as well.

For what concerns decoding, as discussed before, the ZSC decoding layer is very thin and it simply discards the tuning field before feeding the incoming data to the 8B9B decoder. Consequently, the performance of the ZS decoder is very close to the 8B9B decoder alone, which has already been thoroughly evaluated in Chapter 3.4. Experimental data also shows that the extra delay introduced by ZSC decoding is always fixed and negligible (just a couple of CPU clock cycles). Overall, the ZS decoding time $t_3 - t_2$ is about 3.42 μs at the maximum DLC on the platform being considered in this chapter.

One crucial point of the evaluation in Chapter 3.4 was to show that the 8B9B decoder works in *constant* time for a given DLC, and hence, it does not introduce any communication jitter. As mentioned above, the ZSC decoding module does not introduce any other source of jitter either. Hence, the execution of ZS decoding module as a whole is *jitterless*.

Figure 6.6 and Table 6.3 (next page) show the experimental results concerning $t_1 - t_0$, that is, the total ZS encoding time. For better evaluation, $t_1 - t_0$ has also been broken down into the basic 8B9B encoding time plus the additional time needed for ZSC encoding. As shown in Chapter 3.4, the 8B9B encoder is completely jitterless, all the jitter encountered in overall ZS encoding is due to ZSC. For clarity, the error bar is not shown for ZSC encoding delay.

As it can be seen from Figure 6.6, both components of the encoding delay are linear with respect to n_D for $n_D \leq 48$ b. As a consequence, the ZS encoding time also preserves linearity in those cases, as shown in the figure. The difference in the slope between 8B9B and ZSC encoding can be easily justified by considering that in the first case, it depends on the 8B9B encoding loop, whereas in the second case it depends on the R_L calculation loop, which is simpler. In addition, as

n_D	8B9B Delay (μs)	ZSC Delay Average (μs)	ZS Delay ($t_1 - t_0$) Average (μs)	Min (μs)	Max (μs)	Jitter (μs)
16	1.51	3.17	4.68	4.39	4.86	0.47
24	1.94	3.48	5.42	5.13	5.60	0.47
32	2.38	3.91	6.29	6.00	6.47	0.47
40	2.81	4.29	7.10	6.81	7.28	0.47
48	3.24	4.67	7.91	7.61	8.08	0.47
64	3.67	5.25	8.92	8.63	9.10	0.47

Table 6.3. 8B9B, ZSC, and ZS encoding delay (10^7 samples for each value of n_D).

it can be seen from the figure, ZSC encoding is always slower than 8B9B encoding because ZSC also includes a *fixed* and n_D-independent calculation of the tuning field (T).

It is worth mentioning that, when $n_D \leq 48$ b, the data field after encoding is 1 byte longer than the original payload size. Instead, when $n_D = 64$, it is 2 bytes longer since one byte is used to give enough room for tuning in the ZS encoding as explained in Section 6.3.2. However, the number of iterations of the byte-by-byte 8B9B encoding loop just depends on the size of the original payload. That explains why 8B9B encoding time in ZS encoder is *faster* than linear when $n_D = 64$.

Similarly, when $n_D = 64$, ZSC encoding is also *faster* than linear, however, for a different reason. In the actual implementation, before calculating R_L and T, generally some checks on n_D need to be done. The compiler optimizer is able to omit this kind of check when $n_D = 64$. Since ZS is made up of 8B9B and ZSC, these observations also explain why ZS is *faster* than linear in the case $n_D = 64$.

The experimental data shown in Table 6.3 also indicates that the jitter encountered in software encoding is always less than one bit time. More importantly, it is not n_D dependent. In fact, further investigation revealed that this jitter comes from step 16) of the algorithm shown in Figure 6.2. This is because bit stuffing check is only required for $T_j \wr R_j$ instead of the whole frame as already explained in Section 6.1.5 and it is done in two steps in the real implementation, by first checking the existence of primer sequence(s) of all 1s and then all 0s.

However, the compiler optimized the code in a way that if a primer sequence of 1s is found, it will silently skip the check for 0s. Most likely, the programmer might not notice this optimization. In any case, this introduces uncertainty into software execution time. With little effort, this source of jitter can be dramatically reduced to 2 clock cycles (corresponding to 28 ns) with a delay penalty of 1.49 μs at most.

Without this extra jitter reduction, the ZS encoding time is about 8.92 μs at the maximum DLC. This means that the worst-case delay introduced into the communication by ZS is around 12.34 μs, among which 3.42 μs (28%) is for ZS decoding, 3.67 μs (30%) is for 8B9B encoding and 5.25 μs (42%) is for ZSC encoding. This is quite acceptable, as it is well below the minimum frame transmission time $t_2 - t_1$, which is about 123.16 μs as shown in Table 6.2. What's more, the ZS encoding/decoding time could probably be reduced by adopting either a better CPU or a higher clock frequency.

Segment	8B9B (B)	ZSC (B)	ZS (B)
text	398	898	1296
read-only data	384	512	896
bss	n/a	12	12
stack	60	8	68

Table 6.4. Memory requirement.

6.3.4 Memory Requirement

Memory requirement is an important design constraint in low-end embedded systems as generally they are short of on-chip memory. Resorting to external memory will probably increase the cost and complexity for development whereas the real-time performance may be impaired somehow.

Table 6.4 summarizes the memory footprint of the ZS codec and its two components, namely 8B9B and ZSC. For what concerns the ZSC modules, the most significant contributions to footprint are given by 898 B of code in the *text* segment, mainly for ZSC encoding, plus 512 B of *read-only data* used for the CRC lookup table mentioned in Section 6.3.1. On the other hand, the requirement of ZSC in the *bss* (uninitialized data) and *stack* segments amounts to only 20 B in total, and hence, is minimal.

Generally speaking, both the text and the read-only data segments are stored in (non-volatile) FLASH memory, while uninitialized data and stack take space in RAM memory. As a result, the ZS modules overall require 2192 B of FLASH memory and 80 B of RAM. Considering that 512 KB of FLASH memory and 96 KB of RAM are available on the microcontroller used in the experiments, this is quite acceptable. It should also be noted that the extra space needed for ZSC mostly concerns FLASH memory, which is generally more abundant than RAM on this kind of systems.

Summary

In this chapter a mechanism has been presented, known as ZSC, that allows to prevent stuff bits in the CRC of CAN frames. When coupled with a ZSD encoding scheme, like 8B9B, this prevents the insertion of stuff bits all over the frame except for the header. As mentioned in Section 6.1.5, stuff bits in the header do not lead to any jitter and, at the end, ZSC is indeed able to achieve truly jitterless communication (provided that CAN arbitration is not exploited). The advantage of this approach, when compared to other interesting solutions such as the one in [91], is that no modification is required to the hardware. On the contrary, conventional CAN controllers can be used, by placing over them a tiny software layer that carries out the ZSD and ZSC functions.

A ZS codec has been developed, which features small footprint and low processing times. More importantly, experimental results show that the residual jitter can be at least 40 times lower than in plain CAN. As a consequence, the proposed solution is proved to be able to support razor-sharp actuation in CAN systems even when implemented in software.

It is worth remarking that ZSC operates mostly on the transmitting side. Receivers must mainly be able to decode the leading part of the frame by means of the ZSD codec, because ZSC decoding just discards the tuning field before the data field is passed to the ZSD decoder. This means that

performance penalties are only found on transmitters. At the limit, simple nodes that only have to receive jitter-free messages, are allowed not to implement ZSC at all. In the case of master-slave systems, this permits lowering the implementation cost of slaves noticeably.

Chapter 7

Effect of Jitter-Reducing Encoders on CAN Error Detection

It is well known that bit stuffing in CAN may interfere with error detection and, in particular, with CRC-based approaches, so that the residual error probability can be noticeably higher than the theoretical value. Encoding techniques that prevent the insertion of stuff bits in the payload of the frame are a simple remedy, which improves integrity of data exchanged over the network noticeably.

A thorough experimental campaign was carried out in order to evaluate the benefits the adoption of such encoding schemes provides over plain CAN. Results confirm that, for realistic error models, the probability of having residual errors decreases by about two orders of magnitude at most payload lengths, which makes such an approach very effective. This implies that much higher reliability can be obtained from conventional CAN controllers, which resembles newer solutions like CAN FD.

7.1 CRC-Based Error Detection

Functional correctness is one of the main requirements of control systems, in particular when aspects related to either safety or dependability are involved. In the case of distributed control systems, where devices are interconnected by means of digital communication networks, the integrity of transmitted data has to be preserved irrespective of errors that may possibly take place on the communication support—due, for instance, to electromagnetic interferences. Although the error rate on wired transmission channels is usually low—and can be reduced further by adopting suitable solutions at the physical layer, like differential signaling, shielded wires, and so on—preventing errors completely is just impossible. Among the techniques implemented in the higher layers (data-link and above) to ensure data integrity, error detection (coupled with automatic retransmission) is certainly the most common solution. Error-correcting codes also exist, but they are seldom used in control networks.

The *cyclic redundancy check* (CRC) is one of the most effective ways to detect errors in the value domain. Basically, a small signature is calculated as the remainder of a polynomial division

and added to every transmitted frame. By carrying out the same computation, receivers can determine (with a high likelihood, which depends on the specific CRC chosen) whether or not the frame was altered during transmission. The CRC mechanism can be applied easily to any kind of network. For this reason the vast majority of the communication protocols (including those used in control systems) rely on such an approach.

While improving robustness noticeably, CRCs are not perfect [75]. Every error pattern that turns a valid frame into another valid frame goes *undetected*. The *residual error probability* π_{res} is defined as the likelihood of this event occurring. In many applications found in factory automation this is not felt as a real problem. In fact, most fieldbuses and real-time Ethernet solutions rely uniquely on CRC to ensure data integrity. Things change when safety-critical applications are taken into account. In the case of safety communication protocols (PROFIsafe, SafetyNET p, etc.), nested CRCs are typically used to reduce the residual error probability to values that are deemed acceptably low [89]. For high-dependability in-vehicle networks (FlexRay, TTCAN), error detection in the time domain is used along with CRC. Such an approach requires time-triggered communications and permits dealing with other kinds of error as well.

Controller Area Network (CAN) lies in between. Since this protocol was conceived to be as simple as possible, it does not rely on either nested CRCs or time-triggered approaches. Nevertheless, CAN reliability is undoubtedly higher (by design) than other networks used in automation. For this reason it became the "de facto" standard solution in the automotive industry. One of the reasons of the robustness of CAN is bit monitoring: while transmitting, every node senses the level on the bus and compares it with the value of the bit it is sending out. Should a mismatch occur, an error is detected. In turn, this forces the transmission of an error frame, aimed at making all the other nodes aware of the failure, and the subsequent retransmission of the original frame. Bit monitoring permits CAN to deal with all *global* errors—i.e., channel errors that affect in the same way every node in the network.

Unfortunately, *local* errors may be experienced as well on the CAN bus, which only involve a subset of nodes. They are due to a number of reasons, including the non-perfect alignment of sample points in different receivers or signal attenuation. Whenever a local error does not affect directly the transmitter(s), the CRC, form, and stuff error detection mechanisms are available to receivers to discover it. This leads to a small (but not null) probability of residual error, which can be estimated precisely through known results about CRCs. In theory, the CRC in CAN permits detecting up to 5 erroneous bits, irrespective of where they occur in the frame, as well as error bursts up to 15 bits in length. In practice, error detection capabilities are worse. In fact, the CRC is evaluated in CAN before *bit stuffing* (BS) is applied [44]. As pointed out in several papers [14, 94] doing so leads to unwanted effects sometimes, which may compromise CRC effectiveness and increase the residual error probability noticeably.

The actual impact of this peculiar issue on the correct behavior of the overall system is not easy to assess: as far as we know, no figures have been made publicly available by carmakers about the number of real accidents whose causes can be ascribed to the interference of bit stuffing on error detection mechanisms. What can be said for sure is that a new version of CAN—called CAN with Flexible Data Rate (CAN FD) [83]—has been introduced in 2012 by Bosch, which was pushed by the automotive industry in order to bring two major improvements. The first is about performance, which can be increased noticeably by means of overclocking/oversizing. The second, instead, has to do with error detection. In particular, the original CAN protocol was modified so that the CRC

is now computed starting from the stuffed bit sequence (a longer CRC and a different encoding for the related field are also envisaged in CAN FD). This change is clearly aimed at preventing the interference described above. Therefore, one is probably not wrong when it says that such a problem is not negligible.

CAN FD is a promising solution and chances are that it will eventually replace CAN in the automotive scenario. However, as a matter of fact it is not (completely) backward compatible with previous controllers, which are unable to decode frames with the new format and thus flag them as transmission errors. This means that devices based on CAN FD controllers can be used together with existing CAN devices only provided that new protocol features (including improved error detection) are not exploited at all. For this reason some interim solution would be welcome in the meanwhile, which applies to existing CAN controllers and ensures the same benefits, in terms of increased reliability, as CAN FD.

To this extent, suitable encoding techniques which prevent the insertion of stuff bits in the data field of CAN frames, for instance 8B9B introduced in Chapter 3, can be adopted on top of conventional CAN controllers. It is interesting to note that the very same approach (and codec) can also bring noticeable advantages concerning the residual error probability. In fact, removing (most of) the stuff bits in CAN frames reasonably decreases the likelihood that they may interfere with error detection and, in particular, the CRC.

In this chapter the analysis about the dependence between CRC and bit stuffing is first extended with respect to the works available in literature [14, 94]. Taking the effect of local errors into account is not that simple, mainly because every node may be affected, in theory, by a different error pattern. Because of state space explosion, using formal verification techniques is practically unfeasible, unless precise (yet realistic) simplifying assumptions are made on error models. Then, the benefits on data integrity achieved by including 8B9B encoding in CAN transmission are evaluated through a simulation study aimed at estimating the residual error probability for specific error models that may realistically occur in practical cases.

7.2 CAN Channel and Error Models

A proper definition of the CAN channel and error models is crucial to define and understand the effect of errors in a sound way. We start from the models proposed in [14, 94] to analyze error effects by means of a combination of theoretical derivations and simulation. Despite its simplicity, the proposed model is able to provide lower and upper bounds on the probability of residual error.

7.2.1 CAN Channel Model

As proposed by [14], the CAN channel has been modeled as a two-state binary symmetric channel. During the transmission of a frame, the channel can be in either a *bad* or *good* state, with probabilities χ_{bad} and $1-\chi_{\text{bad}}$, and it stays in that state for the whole frame. A bit error probability, $\pi_{E_{\text{bad}}}$ and $\pi_{E_{\text{good}}}$ respectively, is associated with each state. Being the channel symmetric, the probability of a bit to change from dominant ⓓ to recessive ⓡ, due to an error, is assumed to be the same as a change from ⓡ to ⓓ. The same model was kept in later works like [94] because, to the best of our knowledge, no specific models have been published for recessive/dominant channels.

Since the channel model is exactly the same in the *bad* and *good* states—except for the bit error probability—the residual error probability in a certain state is a function $\pi_{\text{res}}(\pi)$ of the associated bit error probability π only, where $\pi \in \{\pi_{E_{\text{good}}}, \pi_{E_{\text{bad}}}\}$. Therefore, we can write

$$\pi_{\text{res}} = \chi_{\text{bad}} \cdot \pi_{\text{res}}(\pi_{E_{\text{bad}}}) + (1 - \chi_{\text{bad}}) \cdot \pi_{\text{res}}(\pi_{E_{\text{good}}}) \ . \tag{7.1}$$

In most practical cases $\pi_{E_{\text{good}}} \ll \pi_{E_{\text{bad}}} \ll 1$, and hence the analysis can be simplified by considering only the *bad* channel state, that is $\pi_{\text{res}} \simeq \chi_{\text{bad}} \cdot \pi_{\text{res}}(\pi_{E_{\text{bad}}})$. In the following, regardless of this simplification, π_{res} will be determined as a function of a generic bit error probability π_E, so that the results are valid for both channel states.

Let K be a random variable that represents the number of channel errors occurring in a bit stream of length s. Assuming that errors are uncorrelated, K is Binomial and the probability to have k channel errors in the bit stream is

$$\mathbb{P}[K = k] = \binom{s}{k} {\pi_E}^k (1 - \pi_E)^{s-k}, \quad 0 \leq k \leq s \ . \tag{7.2}$$

The rightmost two factors of (7.2) express the probability that a given error pattern, comprising k errors in s bits, occurs. Instead, the leftmost factor represents how many such error patterns are to be considered. The value given in (7.2) corresponds to the case in which *no constraints* are posed on error locations, that is, errors are allowed to appear anywhere in the frame.

For what concerns error *effectiveness*, that is, which CAN nodes are affected by a given error, $\pi_{\text{eff}}^{(\mathsf{T})}$ and $\pi_{\text{eff}}^{(\mathsf{R})}$ denote the conditional probabilities that the error affects the transmitting node or a given receiver, respectively.

7.2.2 Distributed Error Detection

The distributed error detection and reporting capabilities of CAN must be carefully considered because the ability of a receiver to detect an error in the incoming frame can be heavily affected by other nodes in the network, which may detect the error themselves. This is especially important because it is unlikely that CAN is configured as a point-to-point link between only two nodes.

In fact, any CAN node detecting an error is required to signal it by transmitting an *error flag* (*globalization*). When the node is in the *error active* state, the (active) error flag consists of a sequence of 6 ⓓ bits that, when detected by other nodes, will trigger the transmission of another error flag and make them discard the frame being received. All nodes are assumed to be in the error active state in the following. Moreover, as in [14], it is assumed that error frames are always detectable by all nodes in the network.

It must also be noted that not always the transmission of an error flag is helpful to other nodes. In fact, a given receiver will be made aware of the error by the error flag and discard the incoming frame *only if*, according to its own logic, the frame has not yet been accepted (and passed on to the application) when it receives the error flag.

In the following sections, distributed error detection will be discussed in increasing level of detail (and complexity). The discussion starts by considering error effects on the *transmitter* (T) and a single *receiver* (R), each considered in isolation, and proceeds by considering the interactions between them. Last, the role played by *additional receivers*, referred to collectively as $\overline{\mathsf{R}}$, is analyzed.

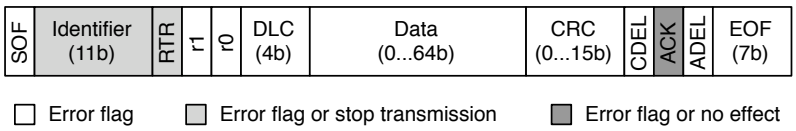

Figure 7.1. Effect of errors on the transmitter.

7.2.3 Errors Affecting the Transmitter

Due to its *bus monitor* function [44], the transmitter is in a more favorable position than receivers to detect and signal errors. During most of the transmission, T monitors the bus level. If the actual bus level differs from what is being transmitted, T detects a *bit error* (according to the standard nomenclature), stops transmitting the frame, and starts sending an error frame at the next bit. This means that, as long as T is involved, any channel error is always detected (and globalized), irrespective of whether it was global or only affected part of the nodes. Two notable exceptions to this general rule apply during arbitration and in the Acknowledge (ACK) slot.

More specifically, as summarized in Figure 7.1, three distinct cases are possible when a *local* error affects only T:

1. If the error occurs in the Start Of Frame (SOF), r1, r0, Data Length Code (DLC), data, Cyclic Redundancy Check (CRC), CRC delimiter (CDEL), ACK delimiter (ADEL), or the End Of Frame (EOF) field, the general rule discussed above applies.

2. If the error occurs in the arbitration field—consisting of the Identifier field plus the Remote Transmission Request (RTR) bit—and it transforms a ⓓ bit into ⓡ, T reacts as in case 1. Else, if the error transformed a ⓡ bit into ⓓ, T believes that it lost the arbitration due to a competing, higher-priority transmitter. It will still stop transmitting, but without sending any error flag. When no other transmitters are active on the bus, the bus will go into the idle ⓡ state. Due to its position within the frame, this will be eventually interpreted by the other nodes as a sequence of ⓡ bits that violates the bit stuffing rules. Accordingly, they will react by sending an error flag.

   It should be remarked that assuming that the sequence of ⓡ bits can reliably be detected by the other nodes is similar to the assumption that the active error flag (a sequence of ⓓ bits) can be detected as well, assuming a symmetric channel. Instead, if other nodes are concurrently transmitting, this error may possibly lead to *priority inversion*. This is because the bit value transmitted on the bus may still be ⓡ since this is a local error just affecting T. This means that other transmitters are sending the same bit as T. In this case, it is possible that T has a higher priority than the others, depending on the following bits in its arbitration field. However, this error makes T believe that its priority is lower and then it decides to withdraw from arbitration.

3. If the error occurs in the ACK slot, and changes a ⓓ bit into ⓡ, T will assume that the frame has not been acknowledged. This is considered as an error and triggers the transmission of an error flag. Else, if the error changed a ⓡ bit into ⓓ, T observes a “fake” acknowledge and assumes that the frame has been successfully received, even though it was not. In this

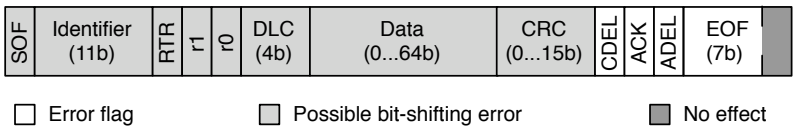

Figure 7.2. Effect of errors on receivers.

case, T does not send any error flag and does not affect the receivers' decision about the validity of the frame. This reduces the effectiveness of the error detection scheme based on acknowledgments, but only in the (unlikely) case of a network where there are no error-active nodes.

7.2.4 Errors Affecting a Receiver

This section analyzes the effects of *local* errors when they affect only R, without taking into account its interactions with T and $\overline{\text{R}}$ (which will be addressed in the next section). The main receivers' error detection capabilities involve *form error* checks—in which fixed-form fields (like CDEL) are compared against their expected value—*stuff error* checks and the CRC check. As it happens for T, the effect of an error depends on its position within the CAN frame, as shown in Figure 7.2.

Errors between SOF and CDEL

When an error affects the bit-stuffed portion of the frame (before CDEL), excluding SOF and DLC, four distinct outcomes, also discussed in [14, 10] in a narrower context, are possible. In the following examples, "⟶" indicates transmission over a channel with errors. Bits affected by errors are overstriked, whereas stuff bits are shown within parentheses (on the left side stuff bits inserted by the transmitter, while on the right side bits that are interpreted as stuff bits by receivers—in the presence of errors they might not coincide).

1. The error may affect a data bit without introducing a stuff error or any other side effects. This happens, for example, when 10~~0~~10 ⟶ 10110.

2. The error may affect either a data bit or a stuff bit and be detected as a stuff error by R. For example, 011~~0~~1110 ⟶ 011111(1)0 (error affects a data bit) or 011111(~~0~~)0 ⟶ 011111(1)0 (error affects a stuff bit). In both cases, 6 consecutive ⓡ bits appear in the corrupted bit stream.

3. The error may affect a data bit and transform a subsequent data bit into a stuff bit, in the opinion of R. For instance, 1000~~1~~010 ⟶ 100000(1)0.

4. The opposite is also possible, that is, an error may transform a stuff bit into a data bit. For instance, 10~~0~~000(1)0 ⟶ 10100010.

Case 2 is the most favourable from the point of view of error detection, because R detects the error directly during bit destuffing. On the contrary, case 1 is the most difficult one because R

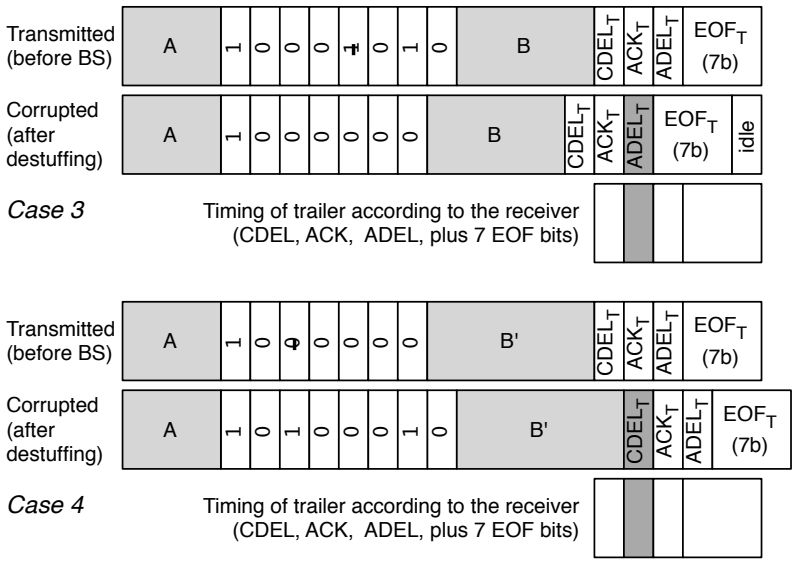

Figure 7.3. Effect of bit-shifting errors.

must rely exclusively on the CRC check to detect it. This is because, even though some bits (for instance, r0) must be sent at a fixed value by the transmitter (ⓓ for r0), the standard stipulates that receivers must accept either ⓓ or ⓡ *without* flagging a form error.

Cases 3 and 4 (called *bit-shifting* errors in the following) either remove or insert a bit into the destuffed message, and hence, they lead R to interpret the message bits not like T intended. For this reason, the impact of this kind of error is more difficult to predict theoretically. For instance, as shown in the upper part of Figure 7.3, after a case-3 error R suppresses a data bit from the message, because it believes it is a stuff bit. Therefore, part B of the transmitted message will be shifted by one bit to the left for R. R also believes that the message trailer (from CDEL on) begins at the ACK slot of T and ends one bit after the end of the transmitted EOF sequence. In the figure, field names listed according to the opinion of T have got a T suffix. Stuff bits are not shown. A case-4 error leads instead to a shift of B' in the opposite direction, as shown in the bottom part of Figure 7.3.

It should also be noted that when more than one bit-shifting error occurs, their effects on the message length may cancel. For instance, if one case-3 and one case-4 error take place, the opinion of R about the message trailer position will nevertheless agree with T. This defeats any kind of error check that R carries out on the message trailer and makes the pair of errors impossible to detect in this way. The case of an even number of bit-shifting errors that cancel each other is referred to in the following as *eliding-shifts* error. Their effect is to misalign one or more bit subsequences in the stuffed part of the frame. As pointed out in [14], eliding-shifts errors are especially interesting because a small number of channel errors may lead to a corrupted destuffed message, which passes the form error checks of R and yet contains a number of bit-level errors beyond CRC detection capabilities.

Errors in the DLC

A different kind of frame length discrepancy between T and R happens when, due to errors, the DLC seen by R differs from that sent by T so that, from the point of view of R, the frame ends "earlier" or "later" by a multiple of byte. As discussed before for case-3 and case-4 errors, the form error checks performed by R may be of limited usefulness because, due to additional errors, R may observe a valid trailer where it is expected.

Errors in the message trailer

Any error in the unstuffed message trailer that affects R—from CDEL until the last but one bit of EOF—leads R to discard the frame. Instead, when the error occurs in the 7$^{\text{th}}$ EOF bit, it is "too late" for R to react in time, because it already accepted the frame at the end of the 6$^{\text{th}}$ EOF bit. In both cases, the trailer position must be calculated according to the understanding of R. As discussed before, this may not coincide with what T intended due to earlier bit-shifting errors.

SOF corruption

This kind of error leads R to synchronize with the next ⓓ bit in the bit stream and consider it as the SOF. In most cases, the effect is the same as a bit-shifting error because the message fields position according to R changes accordingly. An exception occurs when the message begins with `0̶0000(1)` and due to error it becomes `100001`. In this case, the second bit at ⓓ is taken as SOF by the receiver. What's more, SOF corruption also transforms the stuff bit `(1)` into a data bit and the two shifts immediately cancel each other.

7.2.5 Interaction Between Transmitter and Receivers

We agree with [14, 94] on the fact that neither T nor R are able to detect any bit or form errors, respectively, when the frame incurs in eliding-shifts errors, because the message trailer, in the opinion of R, is still aligned with the transmitted trailer. However, the ability to detect bit-shifting errors when their effects do *not* cancel each other should not be taken for granted.

For instance, continuing with the case-3 example in Figure 7.3, let us assume that bit-shifting errors affect only R and the bus is idle after EOF. If the message passes the CRC check R acknowledges it in what it believes to be the ACK slot (provided no form errors were detected up to that point), but its ACK actually overlaps with the ADEL of T. This gives T two distinct opportunities to detect the error and make R aware of it by means of an error flag:

1. If no other receivers are present, or they are passive and affected by detectable errors, they will not acknowledge the frame in the ACK slot of T, leading T to flag an ACK error.
2. T will flag an ADEL bit error on the ACK of R.

However:

1. The other receivers may be unaffected by errors and provide a valid acknowledgment. This defeats the first error detection opportunity on the transmitter side and, as discussed later, it does not necessarily trigger any error on the erroneous receiver side.

2. The ADEL bit error is detected only if T is *not* affected by an error at that bit position. Since the CAN bus is inherently symmetric, there is no reason to accept the occurrence of errors when data flow from T to R and neglect them in the opposite direction.

More specifically, the following sequence of events may occur:

- T sends a frame with a CRC ending in `01111`.
- A group of receivers receives the frame correctly and acknowledges it, setting the frame trailer (from CDEL up to the 6th EOF bit) to `101111111`.
- Due to a bit-shifting error, another group of receivers considers a data bit as a stuff bit. From their point of view, the CRC ends in `011111` (the rightmost bit comes from the CDEL of T) and the frame trailer is `111111111`, because the `0` bit in the ACK slot is considered a stuff bit pertaining to the last part of the CRC and the bus is assumed to be idle after EOF. These receivers do not detect any error and acknowledge the frame at the first EOF bit of T.
- Due to a second bit error, neither T nor the first group of receivers is affected by the erroneous acknowledgment, and hence, they do not detect any error.

In summary, it is possible that a receiver accepts a message affected by a double error (one bit-shifting error plus one error in the message trailer) in spite of a one-bit length discrepancy between the transmitter and the receiver. The existence of other error scenarios similar to the one just discussed cannot be excluded—although their probability of occurrence may be lower than the ones discussed in [14, 94]. Therefore, the CAN error model to be discussed in the next section was designed to consider the best and worst cases from this point of view.

7.3 Simulation Framework

Considering every possible interaction between the transmitter, the receiver under analysis, and other receivers on the bus quickly becomes very demanding from both the modeling and simulation performance points of view, because it requires to take all those nodes into account. This is even more important given that the events being considered have an extremely low probability of occurrence, thus requiring a very high number of sample points to be analyzed in a statistically significant way. For this reason, a simplified error model has been adopted for the simulations.

7.3.1 Simplified CAN Error Model

The first simplifying assumption is that the simulation, by itself, considers only two CAN nodes explicitly, that is one transmitter and one receiver. The behavior of R is simulated completely whereas T just generates the bit stream as required, without performing any error checks. This is the same as assuming that any error injected during the simulation affects only R, so that T never detects and flags an error on its own. Additional receivers $\overline{\mathsf{R}}$ unaffected by errors are simulated in a simplified way, by forcing an ACK to appear in the position expected by T.

The second assumption is that, in the simulation, the peculiar interactions among CAN nodes highlighted in Section 7.2.5, which may occur in the presence of errors both in the bit-stuffed part and in the trailer of the frame, are simplified as follows:

- If R acknowledges the frame in the "right" position, that is, no bit-level length discrepancy occurred, T (and $\overline{\mathsf{R}}$) are certainly *never* able to detect that R was affected by local error(s).
- If R acknowledges the frame somewhere else, T (and $\overline{\mathsf{R}}$) *may* be able to recognize that R was in error, depending on message contents, network configuration, and the occurrence of additional errors in the trailer.

  Doing so permits to derive a best and worst-case bound on the residual error probability. Assuming that the error detection mechanisms on T (and $\overline{\mathsf{R}}$) are able to discover all the errors that lead to a bit-level length discrepancy produces the lower bound. On the contrary, relying exclusively on the error checks carried out by R, without considering length discrepancies, yields the upper bound.

  Concerning the case which gives the upper bound, it should be said that the probability that one (or more) bit-shifting errors on R are not detected by T (and $\overline{\mathsf{R}}$), is practically low since it requires yet another error to occur as discussed in Section 7.2.5. Therefore, the lower bound determined in this way is closer to reality than the upper bound—which justifies our simplified approach.

7.3.2 Upper-Layers Error Detection

The upper layers of the communication protocol stack may offer further opportunities of error detection, even after a frame has been accepted by the CAN controller. Namely, if a payload encoding technique involving some kinds of redundancy is employed, like 8B9B, the decoding process may be able to detect that the CAN data field has been corrupted because it contains an invalid codeword. It should be noted that the effectiveness of this technique (that resembles a *presentation layer*) does not depend at all on the application that makes use of the payload.

Another opportunity for a device to detect an error in a received frame resides at the *application layer* where, for instance, a message may be rejected because its identifier is unexpected, its length is invalid, or its payload contains illegal data. However, the effectiveness of this kind of check heavily depends on the structure and complexity of the application itself. For example, the probability of detecting that a message has been corrupted because its identifier is unknown to the application depends on how many distinct identifiers the application itself is supposed to handle. For this reason, in order to be conservative, the simulator does not consider this opportunity, except for a trivial check to discard remote frames (which makes all errors involving RTR detectable).

7.3.3 Simulator

The general architecture of the software is shown in Figure 7.4. It has been designed as a general-purpose simulation framework, based on the C programming language, to experiment with the statistical behavior of CAN employing various payload encoding techniques in presence of communication errors. Test data (depicted as white blocks in the figure) are generated on the transmit side—on the left of the figure—and pass through a set of software modules (gray blocks) that completely simulate a transmitting CAN controller (T), except error detection.

The resulting bit stream is corrupted by an *Error Injector* and the result is sent to the receive side—on the right of the figure. There, the bit stream is decoded and the frame is reconstructed

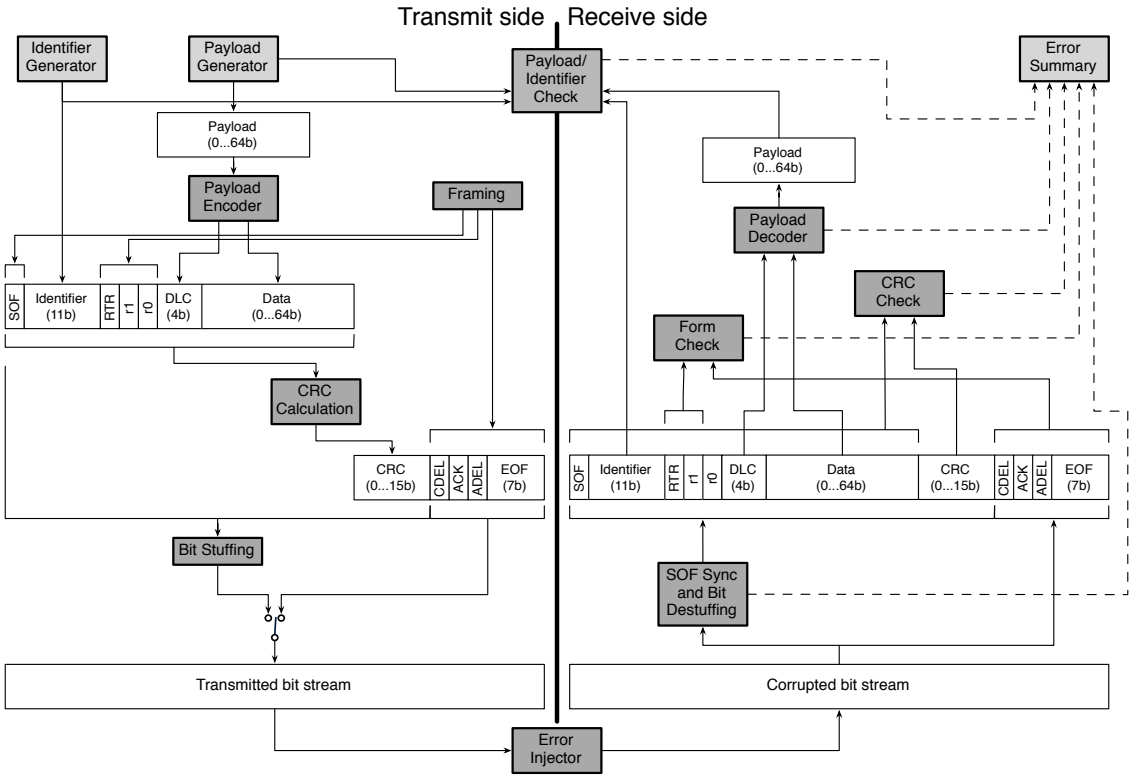

Figure 7.4. Simplified architecture of the simulator.

by means of another set of software modules that implement a receiving CAN controller (R), including error checks. The error injection law is *programmable*. Finally, a *Payload/Identifier Check* module compares the identifier and payload generated by T and what has been received on R, to spot errors that went undetected by the receiving controller checks.

Moreover, the simulator is able to force the correct acknowledgment of a frame (to account for the presence of receivers $\overline{\mathsf{R}}$ unaffected by errors), and it is also possible to enable or disable the length check discussed in Section 7.3.1 (to obtain the lower and upper bounds, respectively).

The main components along the transmit side are discussed in the following.

- The *Identifier Generator* and *Payload Generator* modules generate the identifier and payload according to a programmable law, e.g., as uniformly-distributed random values. The payload length is also programmable. In this chapter, the term *payload* is used for application-level data and *data* is used for the contents of the CAN frame-level data field.

- In order to evaluate the effect of jitter-reducing encoding techniques on CAN error detection mechanisms, the payload is not put directly into the CAN frame. Instead, it passes through a *Payload Encoder*, which outputs the DLC and data field to be used in the transmitted frame. The encoder may be null, in which case the payload and the data field coincide.

- A *Framing* module adds to the frame the SOF bit, the RTR bit, reserved bits r1 and r0, as well as the trailing part. Since the simulator supports neither RTR frames nor extended-format CAN 2.0B frames, the RTR and r1 bits are always transmitted as ⓓ. Otherwise, the receiver is able to detect these kinds of frame and report them as errors.
- A *CRC Calculation* module calculates the CRC of the message from the SOF until the end of the data field. The 15-bit result is put into the CRC field of the frame to be transmitted.
- The *Bit Stuffing* module applies the bit stuffing algorithm to the relevant portion of the frame. The result is combined with the (unstuffed) message trailer to build the complete bit stream to be transmitted.

On the receive side, after error injection, the following modules process the incoming bit stream. Most of them are also able to detect various kinds of error. Those are forwarded to the *Error Summary* module that collects and summarizes them.

- The *SOF Sync and Bit Destuffing* module scans the incoming bit stream, looking for a valid SOF—the one that has been sent by T may have been corrupted. After a successful synchronization, it removes stuff bits from the bit-stuffed portion of the frame. In the process, the expected length of the frame is also calculated, based on the received DLC. The frame trailer (from CDEL to EOF) is taken from the bit stream as is, without bit destuffing.
- A *Form Check* module verifies that the bits within the frame that shall have a fixed value (like CDEL) are correct. It also confirms that the RTR and r1 bits have the expected value, given the fact that R is able to handle only data frames in CAN 2.0A format.
- The *CRC Check* module compares the received CRC against the expected CRC, according to the other parts of the received message.
- The *Payload Decoder*, given the DLC and data fields found in the received message, outputs the application-level payload.

The main modules presented above are complemented by several auxiliary components, not shown in Figure 7.4 for clarity. The most important one among them is the *Random Number Generator* (RNG), used by the Error Injector, as well as the Identifier and Payload Generators.

The standard C library RNG *rand48* (based on a linear congruential generation algorithm) was deemed inadequate to drive the simulation, especially because extremely low-probability events are of interest and the simulation needs long vectors of random values [32]. These remarks were corroborated by early experience with the simulator, whose results exhibited large statistical variations, even considering a large number of samples. For this reason, a better-performing RNG was incorporated into the simulator, namely the *mt19937* generator based on the Mersenne Twister algorithm [57].

7.4 Simulation Results and Discussion

The residual error probability was evaluated for the simplified error model described above in two specific cases, when either a single or a double channel error is experienced during frame

Simulation parameters				$f_{\text{res}}^{(1)}$
Encoder	**Len (B)**	**L. Chk**	**Ack**	**($\cdot 10^{-6}$)**
None	7	✓	n/a	0
None	8	✓	n/a	0
8B9B	7	✓	n/a	0
None	7	—	—	3.42
None	8	—	—	2.55
8B9B	7	—	—	0
None	7	—	✓	0.050
None	8	—	✓	0.045
8B9B	7	—	✓	0

Table 7.1. Single errors in the frame leader.

transmission. Because of (7.2) and the typical values for the bit error probability π_E reported in [27], considering a larger number of errors is not worth the additional effort in realistic CAN setups.

7.4.1 Single Errors

The first set of simulations, whose results are shown in Table 7.1, was aimed at evaluating the effect of single errors in the leading part of the frame, that is, from SOF to the end of CRC included. To this purpose, the simulator was instructed to generate $n = 2 \cdot 10^9$ uniformly-distributed, random identifiers and payloads, injecting *one error* into them. Each simulation run consumed about two hours of CPU time on an Intel Core 2 Duo processor running at 2 GHz. In different runs, the simulator was configured to use or not use the 8B9B payload encoder (first column of Table 7.1), work with different payload lengths (second column), perform or skip the length checks discussed in Section 7.3.1 (third column), and simulate or not simulate the presence of one (or more) additional receivers ($\overline{\mathsf{R}}$) unaffected by errors (fourth column). The frequency of residual errors $f_{\text{res}}^{(1)}$, relative to the number of samples n, is listed in the fifth column of the table. For 8B9B, the maximum payload length was chosen, that is, 7 B. The simulation without encoding was carried out with a payload length of 7 B (to make the comparison sensible from the application point of view) and 8 B (to obtain approximately the same frame length on the CAN bus).

The probability of each sample point (that is, to have exactly one error affecting only R), $\pi_{\text{sam}}^{(1)}$, is given by

$$\pi_{\text{sam}}^{(1)} = s\pi_E(1-\pi_E)^{s-1}(1-\pi_{\text{eff}}^{(\mathsf{T})})\pi_{\text{eff}}^{(\mathsf{R})} \quad . \tag{7.3}$$

The leftmost three factors specialize (7.2) when $k = 1$ while the rightmost two factors express the error effectiveness as defined at the end of Section 7.2.1. The factor $\prod_{r\in\overline{\mathsf{R}}}(1-\pi_{\text{eff}}^{(r)})$, related to other receivers $\overline{\mathsf{R}}$, was omitted for simplicity. The residual error probability $\pi_{\text{res}}^{(1)}$ estimated from $f_{\text{res}}^{(1)}$ obtained from the simulations must be considered as a *conditional* probability with respect to $\pi_{\text{sam}}^{(1)}$. The expression of $\pi_{\text{sam}}^{(1)}$ depends on π_E, $\pi_{\text{eff}}^{(\mathsf{T})}$, and $\pi_{\text{eff}}^{(\mathsf{R})}$, which must be derived from the electrical characteristics of the CAN channel [27], and also on s, the frame length. This value is difficult to calculate analytically because it depends on the amount of bit stuffing and varies

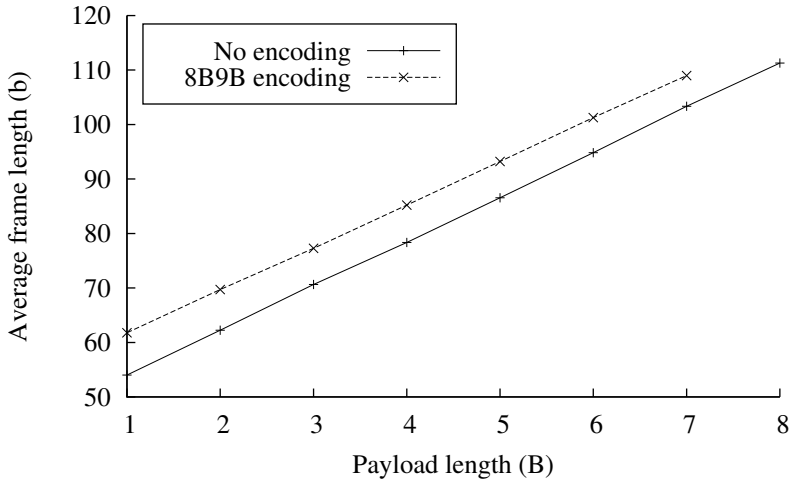

Figure 7.5. Average frame length.

sample by sample (it might even differ among different nodes due to bit-shifting errors.) A useful estimation of s, the average length of the frames considered during the experiment, comes from the simulator itself and is shown in Figure 7.5. Two different curves are plotted, with and without the 8B9B encoder.

As shown in the top three lines of the table, if it is assumed that bit-length discrepancies can reliably be detected (length checks enabled), the presence of other receivers is irrelevant for the experiment, because the frame trailers according to R and all $\overline{\mathsf{R}}$ are always aligned with the frame trailer of T upon undetectable errors. In this case, $f_{\text{res}}^{(1)}$ upon single errors is consistently zero. This does not necessarily imply that the *probability* of residual errors $\pi_{\text{res}}^{(1)}$ is also zero.

However, if we let F_{res} be the random variable that describes f_{res} from the simulation, the fact that $F_{\text{res}} = 0$ allows us to assert, with a certain confidence level, that π_{res} is below a threshold. Under the assumption that residual errors are statistically independent in the simulation space, consisting of n simulation samples, and they occur with constant probability π_{res}, F_{res} is Binomial, which implies $\mathbb{P}[F_{\text{res}} = 0] = (1 - \pi_{\text{res}})^n$. Let us set, as null hypothesis, that $\pi_{\text{res}} \geq \pi_0$. The conditional probability of observing $F_{\text{res}} = 0$ under the null hypothesis is:

$$\mathbb{P}[F_{\text{res}} = 0 \mid \pi_{\text{res}} \geq \pi_0] \leq (1 - \pi_0)^n \quad . \tag{7.4}$$

Given a confidence level of γ, solving the inequation

$$\mathbb{P}[F_{\text{res}} = 0 \mid \pi_{\text{res}} \geq \pi_0] \leq (1 - \pi_0)^n \leq 1 - \gamma \tag{7.5}$$

with respect to π_0 gives a threshold on π_{res} that allows us to reject the null hypothesis, and hence state that $\pi_{\text{res}} < \pi_0$, with a confidence level of γ.

When $\gamma = 0.95$ (which represents a confidence level of 95%), solving (7.5) with respect to π_0 gives $\pi_0 = 1.5 \cdot 10^{-9}$. In other words, this is to say that when we observe $f_{\text{res}} = 0$ we can assert that $\pi_{\text{res}} < 1.5 \cdot 10^{-9}$ with 95% confidence.

Simulation parameters				$f_{\text{res}}^{(2)}$
Encoder	**Len (B)**	**L. Chk**	**Ack**	**($\cdot 10^{-6}$)**
None	7	✓	n/a	0.204
None	8	✓	n/a	0.241
8B9B	7	✓	n/a	0
None	7	—	—	3.70
None	8	—	—	3.94
8B9B	7	—	—	0.191
None	7	—	✓	0.393
None	8	—	✓	0.419
8B9B	7	—	✓	0.001

Table 7.2. Double errors in the frame leader.

On the other hand, when length checks are disabled, the simulation summarily considers the additional error modes presented in Section 7.2.5, provided the sample point probability $\pi_{\text{sam}}^{(1)}$ is adjusted to consider the occurrence of (at least) a second error in the frame trailer (which does not affect R, but T and $\overline{\text{R}}$) that prevents T and $\overline{\text{R}}$ from detecting the error. In this case, $f_{\text{res}}^{(1)}$ without 8B9B encoding is no longer zero.

8B9B brings at least a 30-times improvement over CAN. The worst case is obtained by comparing 8-byte unencoded frames in the presence of other receivers, for which $f_{\text{res}}^{(1)} = 45 \cdot 10^{-9}$ (penultimate row of Table 7.1), and the threshold $\pi_0 = 1.5 \cdot 10^{-9}$ of $\pi_{\text{res}}^{(1)}$ with 8B9B encoding. The improvement increases to at least 3 orders of magnitude when there are no other receivers on the network.

7.4.2 Double Errors

A second set of simulations was run with the same parameters, but injecting *two distinct errors* in the frame, thus considering the eliding-shifts error modes described in [14, 94]. In this case, the sample point probability is:

$$\pi_{\text{sam}}^{(2)} = \binom{s}{2} {\pi_E}^2 (1-\pi_E)^{s-2} (1-\pi_{\text{eff}}^{(\text{T})})^2 {\pi_{\text{eff}}^{(\text{R})}}^2 \quad . \tag{7.6}$$

From the results shown in the top three lines of Table 7.2, it is again evident that 8B9B can reduce $f_{\text{res}}^{(2)}$ in excess of two orders of magnitude when length checks are enabled. Indeed, no residual errors when using 8B9B have been detected in the simulations. When length checks are disabled (as shown in the second part of the table) the improvement is not as remarkable although an about 20-fold improvement is still evident, even in the worst case. However, it should be noted that the simulation sample points in the last case (when length check is disabled) have a much lower probability of occurrence than in the first (when length check is enabled), because the presence of (at least) a third error (which does not affect R, but T and $\overline{\text{R}}$) in the message trailer is mandatory for the error to be undetectable.

Simulation parameters			Non-det. freq. ($\cdot 10^{-6}$)	
Encoder	**L. Chk**	**Ack**	**CAN**	**Decoder**
None	✓	n/a	0.241	—
8B9B	✓	n/a	0.057	0
None	—	—	3.94	—
8B9B	—	—	2.87	0.191
None	—	✓	0.419	—
8B9B	—	✓	0.289	0.001

Table 7.3. Modality of error detection.

7.4.3 Modality of Error Detection

When the 8B9B codec is in use, two distinct mechanisms of error detection are available on R:

1. the CAN controller may be able to detect the error on its own, like it does when no codec is used;
2. the 8B9B decoder itself may detect an invalid codeword in its input data (provided the frame passed the first check).

The use of 8B9B obviously improves the likelihood of error detection through the second mechanism because the occurrence of errors in the data field may transform a valid 8B9B codeword into an invalid one, even though those errors are otherwise undetectable by the CAN controller. However, it should be noted that 8B9B enhances the first error detection mechanism, too. In fact, as remarked in Section 7.2.4, the occurrence of an error pattern that the CAN controller cannot detect is tied to the presence of eliding-shifts errors. In turn, a case-4 bit-shifting error requires the presence of a stuff bit (that is transformed into a data bit). The 8B9B encoder completely prevents stuff bits from appearing in the data field of the frame, and hence, reduces their probability in the frame as a whole.

To evaluate the relative importance of the two mechanisms, further simulations were performed to measure their error *non-detection* frequency. The results are shown in Table 7.3 for double errors and comparing 8B plain-CAN frames with 7B 8B9B-encoded frames. This choice makes the frame length as close as possible in the two cases being considered.

When length checks are enabled (top two lines of Table 7.3), 8B9B improves the error detection capabilities of the CAN controller alone by a factor of about 4, while all the other errors are detected by the 8B9B decoder. When length checks are disabled (bottom of the table) the improvement is much less noticeable and most of the errors escape CAN controller detection—although the vast majority of them is still trapped by the 8B9B decoder. However, as already remarked previously, this corresponds to very low-probability error scenarios.

7.4.4 Consistency Across Payload Length

The final set of simulations was aimed at verifying that the residual error probability improvement brought by 8B9B was consistent across the whole range of application-level payload lengths. As before, attention was focused on single errors without length checks (bottom of Table 7.1) and

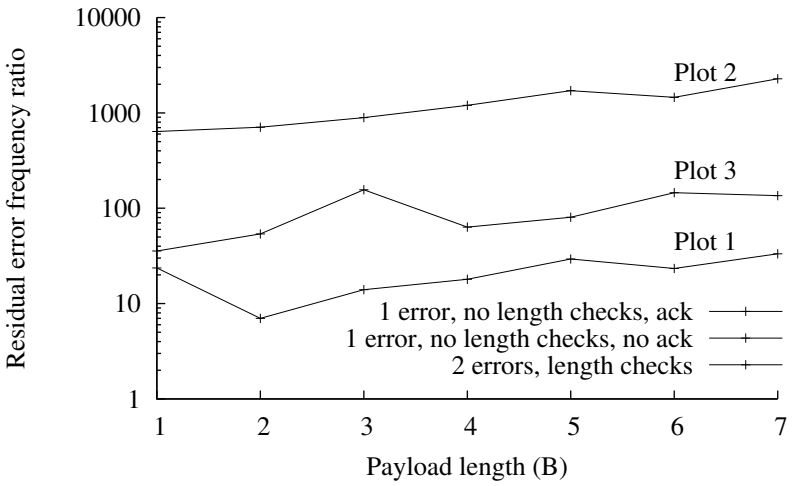

Figure 7.6. Residual error frequency ratio, 8B9B vs. plain CAN.

double errors with length checks (top of Table 7.2) because these are the highest-likelihood cases considered so far, in which the CAN controller by itself has a non-zero residual error probability. The results are plotted in Figure 7.6, which shows the f_{res} ratio between plain CAN and 8B9B-encoded data, across the range of payload lengths supported by both. This ratio estimates the improvement in residual error probability introduced by 8B9B. Due to the very small number of errors undetected by the CAN controller at smaller payload sizes, the number of samples for payload lengths 1 and 2 has been increased to $2 \cdot 10^{10}$ to preserve statistical significance.

For what concerns single errors without length checks and without $\overline{\mathsf{R}}$ (plot 2 in the figure) the improvement is consistently between two and three orders of magnitude. With other $\overline{\mathsf{R}}$, the improvement is only one order of magnitude (plot 1) but it must be remarked that, from Table 7.1, the residual error probability of plain CAN is already much lower in this case. Regarding double errors with length checks (plot 3), the improvement is consistently around two orders of magnitude (unaffected by the presence of additional $\overline{\mathsf{R}}$) except at payload lengths 1 and 2, where it is between 30 and 60 times.

A likely explanation of the reduced performance at these lengths lies in the presence of an unavoidable stuff bit across the RTR, r1, r0 sequence (three ⓓ bits) and the first two bits of the DLC (two more ⓓ bits), regardless of the identifier. In turn, this increases the probability of eliding-shifts errors. The intuition is corroborated by the simulation transcript at length 2, where all undetectable error scenarios share the pattern of one bit error that transforms the above-mentioned stuff bit into a data bit, plus one bit error that transforms a CRC data bit into a stuff bit.

Summary

Bit stuffing in CAN may interfere with error detection mechanisms to the point that the residual error probability increases in a non-negligible way. In turn, this means that data integrity, and as a consequence the correct behavior of the system, can be tampered significantly.

A simple yet effective way to reduce drastically this problem is to adopt a suitable encoding scheme able to prevent the insertion of stuff bits in the data field of CAN frames. Several solutions are available to this extent, which were conceived mainly to reduce transmission jitters. Among them 8B9B is probably one of the most effective techniques. Besides, high performance codecs have been developed, which prove that such an approach can be effectively adopted in existing embedded platforms characterized by limited system resources.

In this chapter, a thorough simulation campaign has been carried out in order to evaluate the real benefits brought by 8B9B encoding on data integrity. Results show that, in typical operating conditions, the residual error probability can be lowered by about two orders of magnitude. Quite unexpectedly, this improvement is mostly due to the intrinsic error detection capabilities of the 8B9B coding and not to the removal of stuff bits. This result is very encouraging, as it permits obtaining a noticeably higher degree of reliability on conventional CAN equipment.

Part III

Flexible Communication in CAN

Chapter 8

General Purpose Protocol Support

Controller Area Network (CAN) is very popular in both in-vehicle and networked embedded systems. On the other hand, intranets are now ubiquitous in office, home, and factory environments. In particular, the Internet Protocol (IP) is the glue that permits any kind of information to be exchanged between devices in heterogeneous systems. In this chapter a network architecture is described that permits CAN buses to be integrated *seamlessly* in intranets. *Flexibility* and *scalability* were the key design requirements, in order to provide users with a comprehensive solution that suits both inexpensive and very complex applications. A sample prototype implementation has been developed and tested in order to assess the feasibility of such an architecture and the performance it can achieve on real embedded platforms.

8.1 IP-Based Communication in CAN

CAN is a real-time communication protocol that was purposely conceived for use in the automotive field. Not only it succeeded to become the "de facto" standard for connecting electronic control units (ECU) in cars, trucks, and other vehicles, but also it achieved good diffusion in factory automation and networked embedded systems. CAN is very stable and quite inexpensive. Many recent microcontrollers embed one or more CAN controllers, which makes this communication technology very cost-effective. Unfortunately, CAN struggles to keep the pace with newer solutions, because of the limitations affecting its network extension and transmission speed (1 Mbps at best for buses shorter than 40 m). Conversely, real-time Ethernet (RTE) networks run at 100 Mbps, which is (at least) two orders of magnitude higher than CAN. For these reasons, CAN is now deemed mostly suitable for application fields where good reliability is required along with very low implementation and deployment costs. For instance, it is increasingly popular in networked embedded systems.

In the past decade, the world of automation has witnessed the progressive move of industrial communications from fieldbuses to RTE solutions [28]. Besides a noticeable increase in network throughput, RTE enables seamless (non real-time) connectivity on field devices at both the Ethernet and IP levels. This permits an unprecedented degree of integration between real-time control systems located at the shop-floor and the applications found in the upper levels of the automation pyramid—e.g., enterprise resource planning (ERP). A similar move is currently taking place

in other applications fields, e.g., power distribution [39] and vehicular systems [80]. Instead, CAN has not been affected noticeably by this trend yet, mainly because it relies on a very simple protocol, targeted at connecting inexpensive equipment in networked embedded systems. Nevertheless, for the reasons previously mentioned, a uniform, well-agreed approach is more and more needed to achieve deeper integration between CAN buses and factory/building communication infrastructures—which typically rely on intranet technologies.

The idea of enabling IP-based communication in CAN is not new. The Internet Draft [6] defines a protocol for transporting IP datagrams over CAN, including addressing aspects. In [22] a gateway between Internet and CAN is described along with a prototype implementation on Linux-based PCs. Two different implementations of IP over CAN on real embedded devices, targeted at vehicular systems, are presented in [53] and [49].

All these approaches mainly deal with the protocol used to transmit datagrams over CAN. Conversely, our goal is to primarily focus on the overall network architecture, which permits CAN buses (and the related devices) to be seamlessly connected to intranets (e.g., factory backbones) and the Internet as well.

- The first requirement that drove its design is *flexibility*. Besides supporting IP communication between end nodes, transmission of Ethernet frames is also foreseen, which enables switchports to be set up. In particular, a multi-point tunnelling is defined that permits Ethernet (*payload protocol*) to be transparently layered on CAN (*delivery protocol*). While doing so is typically pointless, nevertheless it is sometimes required so as to support protocols that do not rely on IP, e.g., the Link Layer Discovery Protocol (LLDP) [41].

- The second key requirement is *scalability*. In the simplest cases, CAN nodes are connected to the intranet through simple routers. Both nodes and routers can be implemented inexpensively by using available open-source software (e.g., lwIP [23]). Conversely, for more complex systems suitable switches are defined, which permit bridged networks to be deployed by seamlessly interconnecting CAN buses, Ethernet links, and wireless extensions based on Wi-Fi.

8.2 CAN/Intranets Integration Approaches

Integration is a key requirement in every modern automation system [87]. To this extent, information must flow not only among devices in the same subsystem (horizontal integration), but also between systems located at different levels of the automation pyramid (vertical integration) [86]. In particular, the systems devoted to real-time control found at the shop-floor and based on fieldbus technology must be able to interact with office-level IP-based networks. The resulting interconnected system may be quite complex and include—besides real-time networks—the plant backbone, wireless LANs, and also public data networks to support remote (non real-time) access. This is the case, for instance, of the virtual automation networks (VAN) project described in [4]. Very similar considerations apply to CAN-based systems. In this case, besides factory automation, networked embedded systems are involved as well.

8.2.1 State of the Art

Different approaches exist for integrating CAN into larger systems—typically based on Ethernet—depending on the protocol layer at which interconnection is carried out:

- The first class of solutions relies on suitable *gateways*, which operate at the application layer. They are mostly commercial solutions, not backed by standard specifications. For instance, EtherCAN/2 foresees specific gateways for accessing CAN devices via Ethernet. To this purpose, the NTCAN [25] and EtherCAN Low Level Socket Interface (ELLSI) [26] application program interfaces are available for, e.g., Windows and Linux. Anybus [34] foresees network equipment aimed at connecting CAN devices to either other fieldbuses or industrial Ethernet. In this case, a specific kind of gateway is required for every host network, which means that they are not universal solutions. In many cases, mapping of functionality and behavior is not complete and not even perfect.

- The second class carries out integration by means of *routers*, located below the application layer but above the data-link (DL) layer. For instance, the framework defined by the Open DeviceNet Vendor Association (ODVA) foresees routers for interconnecting distinct networks, possibly based on different transmission technologies (CAN, Ethernet, etc.), provided that they rely on the Common Industrial Protocol (CIP). In this way, DeviceNet segments can be easily integrated in EtherNet/IP. Similarly, type-2 Ethernet POWERLINK (EPL) routers can be used to couple EPL and CANopen networks, enabling remote access to the object dictionary (OD) of nodes. These solutions offer seamless communication between devices, but they are limited to a specific application layer.

- The last class relies on specific *bridges* that work at the DL layer according to the store and forward principle. They are used to couple distinct CAN buses, possibly with different bit rates. By decoupling MAC operations, bridges permit increasing the network extension beyond the theoretical limits. Some of them enable selective forwarding based on message filtering (so as to reduce local traffic), possibly permitting identifier translation to achieve modular design. These solutions are mostly adopted for in-vehicle applications and are not suitable for interconnecting CAN to high-level networks. However, special bridges can be envisaged that link distinct CAN buses through Ethernet. They work by embedding CAN messages (either one or more) into Ethernet frames. A discussion on the subject is reported in [88], while [63] deals with the ways to improve the related response times.

The main advantage of the above solutions is that, no modifications are required to the existing CAN devices. Coupling devices (gateways, routers, bridges) make conventional CAN nodes communicating with either devices connected to a distinct CAN bus or devices that are not directly provided with a CAN interface (e.g., those having Ethernet ports only). Although the basic timing properties of CAN are not retained completely (nor they could, because of the intermediate relay equipment and networks), such solutions may still ensure real-time behavior. Worst-case analysis of delays in bridged heterogeneous networks comprising CAN and switched Ethernet is reported in [52], whereas the case of CAN and Wi-Fi is dealt with in [79].

In this chapter, a different approach is followed. In particular, integration is carried out at either the Ethernet or IP level. While requiring modifications to the firmware of CAN devices, this

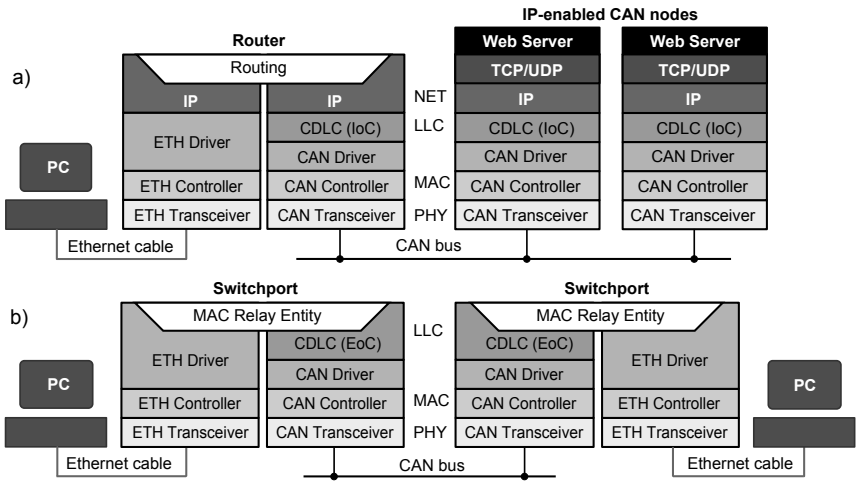

Figure 8.1. Sample systems exploiting interconnection of CAN and intranets.

enables them to communicate using potentially *every* kind of protocol defined for Ethernet and IP, respectively. Moreover, the behavior of the coupling devices can be standardized under almost every point of view—besides addressing, as will be explained in the following—irrespective of the specific CAN-based network and high-level protocol.

It is worth remarking that such an approach was *not* conceived bearing real-time communications in mind—including process data exchanges and asynchronous event notifications. To this aim, existing CAN-based solutions can be exploited (*coexistence* with them is a key point that has to be carefully considered). Instead, its purpose is to achieve seamless connectivity between CAN devices and intranets. It is mostly targeted at operations like remote configuration, monitoring, and diagnostics, which can be layered on top of IP and carried out using, e.g., a conventional web browser. While making CAN devices a bit more complex, doing so removes the need for dedicated management tools. For this reason, it should be regarded as a *long term* solution, meant to be adopted as a universal replacement for non real-time operations, which can be carried out according to a unified approach.

8.2.2 Proposed Interconnection Framework

Intranets are non-global computer networks that rely on IP and are deployed in the context of a specific organization (enterprise, plant, campus, etc.). Each such network consists of a number of subnetworks, referred to as *IP subnets*, which are interconnected through *routers*. The IP model refers generically to the underlying transmission networks as *links*, denoted DL-networks in the following. Each IP subnet may either coincide with a single DL-network or include a number of them—possibly based on different (but uniform) DL protocols—connected through bridges. Ethernet and Wi-Fi are two technologies typically employed in intranets, which rely on switches and access points, respectively. A bridged DL-network is seen by routers as a single link that, as a whole, behaves as a broadcast domain.

The sample network architecture depicted in Figure 8.1a can be used to enable IP communication over CAN, whereas Figure 8.1b concerns CAN switchports. In both cases a number of generic modules are included: hardware components (controllers/transceivers for CAN and Ethernet), the related device drivers, software modules related to the upper layers of the protocol stack (TCP/UDP/IP), and application programs implementing high-level network protocols (e.g., a web server).

Enabling Ethernet and IP communication over CAN requires that a specific *DL transport protocol* is defined—not to be confused with TCP and UDP, which belong to the transport layer—whose purpose is to overcome the limitations of CAN. In the following, it will be referred to as *CAN-based Data Link Control* (CDLC). Basically, CDLC is located immediately atop CAN drivers—i.e., it relies on the logical link control (LLC) services of CAN—and offers the upper layers a suitable interface, which permits the related *protocol data units* (PDU) to be transferred properly. CDLC must deal with two peculiarities of CAN, which make it unsuitable for direct transmission of Ethernet frames and IP datagrams, namely the limited payload size and the object-based addressing scheme.

The entity that implements CDLC in any CAN node is referred to in the following as *DL-entity*. This is because it offers the typical services that the upper layers (e.g., IP) expect from the data-link layer. From a conceptual point of view, CDLC operates by exchanging DLPDUs between DL-entities, where each DLPDU conveys exactly one PDU of the payload protocol (Ethernet frame, IP datagram, etc.) and is exchanged through one or more messages of the delivery protocol (CAN).

8.2.3 Two Different Approaches for Interconnection

Two different approaches can be devised in order to provide CAN devices with IP connectivity. Either IP is stacked directly above CDLC or the latter has to mimic Ethernet communication. In both cases CDLC permits DLPDUs to be exchanged on the CAN bus. The main difference between these approaches concerns addressing.

Although it may seem counter-intuitive, stacking IP directly on CAN is the most straightforward solution. In the following it will be referred to as *IP over CAN* (IoC). CDLC transfers whole IP datagrams in IoC. In particular, the source and destination IP addresses (IP-SA and IP-DA) are used by IoC nodes and routers to accomplish DLPDU routing. A second solution is to equip CAN nodes with an interface to the upper layers that resembles Ethernet, so that existing IP implementations can be used with minor modifications. Every DL-entity, in this case, must be assigned a globally unique MAC address. This is referred to as *Ethernet over CAN* (EoC). Working at the DL layer is more complex but achieves greater flexibility. Several cases can be distinguished, depending on the number and type of network interfaces: EoC nodes have only one interface (used to connect to CAN), whereas *C-switches* (C-SW) must have two or more (either CAN or Ethernet). More specifically, a C-SW having only one CAN interface is known as *switchport*, while those provided with CAN interfaces only (at least two) are called *CAN bridges*.

Supporting more than one payload protocol implies that every DLPDU must be tagged with the related protocol. Reasonably, Ethernet, IPv4, and ARP have to be considered, along with a set of network control messages used for coordinating DL-entities over CAN. However, it should be possible to include other protocols as well, e.g., IPv6. In this chapter, IPv4 was explicitly

considered, as overheads in IPv6 are noticeably higher and hardly fit the CAN bandwidth. Thus, encapsulating IPv4 in CDLC is probably the most reasonable option at the moment. Nevertheless, this is not a limiting choice for three reasons: first of all, the very same EoC/IoC approach can be easily applied to IPv6; secondly, IPv4 and IPv6 are perfectly interoperable; last but not the least, the recently introduced CAN with flexible data rate (CAN FD) [83], given its higher throughput, may be fast enough for IPv6. In the last case, our solution can be taken as the basis for a "wired" version of 6LoWPANs, which achieves very low cost, enables the Internet of Things (IoT) in automation environments [21], and supports uninterruptible operation by powering devices through the cable.

8.3 CAN-Based Data Link Control

CDLC has to tackle aspects related to both payload size and addressing. Its exact definition is, under many respects, uncorrelated with the network architecture. For instance, a suitable solution is defined in [6]. In this chapter a simpler protocol has been developed, which proved to be quite stable and efficient.

8.3.1 Payload Size

One of the main problems of transferring Ethernet frames or IP datagrams over CAN is the very different size of the *maximum transmission unit* (MTU) for the two protocols. For example, an Ethernet frame can be up to 1500 bytes in length (including those encapsulating an IP datagram [37]), whereas the data field in CAN is able to accommodate 8 bytes at most. A similar problem affects the direct transmission of IP datagrams. Although a fragmentation protocol is defined in IPv4 [78], which permits dealing with the different MTU size of the underlying networks, the minimum fragment size is by far too large for a single CAN message.

A straightforward solution to the MTU problem is to exploit fragmentation at the CDLC level. This means, that every DLPDU is split into a number of smaller fragments, which are sent in sequence over the CAN bus and reassembled at the target node(s). Fragmentation is customarily adopted in many application-layer protocols based on CAN. A popular example are service data objects (SDO), used to transfer object dictionary entries in CANopen [7]. DeviceNet [42] defines its own fragmentation protocol, too. Unlike SDOs, it permits peer-to-peer data transfers, but connections have to be created explicitly in advance.

When transferring Ethernet frames over CAN, frame preamble and interframe gap are not taken into account. Including the frame check sequence (FCS) would permit receivers to check the frame exchange on the whole. However, thanks to its error detection and globalization mechanisms, CAN is deemed reliable enough to avoid using this FCS, as witnessed by the fact that neither CANopen nor DeviceNet use it. In the case of IP the entire datagram is transported, including the header field (20 bytes plus options). Doing so permits existing software modules that implement IP to be used directly.

8.3.2 Addressing

CDLC must support true multi-peer communications through a *node-based* addressing scheme, required by the IP protocol [78] and directly supported, for instance, by Ethernet [40]. That is, every DL-entity must be able to explicitly address any other DL-entity connected to the same CAN bus and send it its own DLPDUs. This is in contrast with the *object-based* addressing scheme, specified in the CAN data link layer [44]. Before any messages can be exchanged over CAN, CDLC must therefore assign them a suitable message *identifier*, under the constraint that exactly one producer is allowed for any given identifier, so that arbitration always operates correctly. This implies that each DL-entity should be assigned one or more CAN identifiers for its exclusive use, which implicitly permit distinguishing it on the CAN bus (kind of a source addressing).

Concerning information about the destination DL-entity on the CAN bus, it can be encoded either in the CAN identifier or inside the CAN data field. In the first case the number of allowed DL-entities on each CAN bus is quite low, because of the limited size of standard 11-bit identifiers. This limitation can be loosened noticeably by using extended 29-bit identifiers, but throughput decreases by about 15%. In the second approach frame filtering cannot be exploited on the receiver side so as to reduce the rate at which interrupts are generated—which is usually deemed unacceptable in embedded systems. For this reason, it is strongly suggested that *all* addressing information are stored in the identifier. To this extent two sub-fields are defined in the identifier, whose purpose is to encode the source and destination *CAN node addresses*, denoted CAN-SA and CAN-DA, respectively.

CAN node addresses are much smaller than MAC addresses (6 bytes) and IP addresses (4 bytes in IPv4), which implies that the maximum number of DL-entities allowed on the same bus is typically low. For instance, by reserving half of the standard identifiers in the network to CDLC, 5 bits at most can be devoted to each CAN address. This is not a severe drawback, given what the CAN specification recommends [45], that is, to connect a maximum of 30 nodes at a bus rate of 1 Mbps and maximum bus length (40 m), when using an unshielded twisted pair (UTP) cable and ordinary transceivers. A higher number of nodes can be supported at the physical layer by using high input impedance transceivers but, as suggested above, extended identifiers are required for addressing. Else, distinct CAN buses can be interconnected through bridges.

Whatever the addressing scheme, at least one address must be reserved to broadcast communication for the proper operation of CDLC. Possibly, half of the address space can be reserved to multicast addresses, which means that only 16 CAN nodes are allowed. An individual/global (I/G) bit is used to distinguish between unicast and multicast addresses. In any case, a single pattern, typically 11111_2, is devoted to broadcasts. Assignment of addresses to CAN nodes can be carried out statically, by using dip switches or some other local means (non-volatile memory). Alternatively, CAN addresses can be assigned dynamically (on demand) by using either a centralized or a distributed approach. A thorough description of these aspects is outside the scope of this chapter.

8.3.3 Coexistence with Other CAN-Based Solutions

Most high-level CAN-based protocols (CANopen, DeviceNet, etc.) were not defined taking coexistence with other competing solutions into account. Having EoC/IoC coexist with these protocols on the same bus mostly depends on identifier assignment. So that the new communication

paradigm can be supported in existing systems, clashes among identifiers shall be avoided. For the above reason, a universal assignment scheme likely does not exist.

In embedded networked systems, where assignment is carried out by the system designer with no particular restrictions besides those mandated by schedulability analysis [19] to meet timing constraints, the simple solution described in this chapter may suffice. It reserves the most significant bit in the CAN identifier field to discriminate between CDLC (recessive) and other messages (dominant). This leaves half of the identifiers available for real-time exchanges and also ensures that the worsening CDLC causes on their latency is bounded.

Guaranteeing coexistence is a bit harder when CDLC has to be implemented in existing CAN-based networks. For instance, the number of identifiers left unused in the predefined connection set of CANopen is not sufficient to provide all nodes (up to 127 per network, according to the basic CANopen specification) with IP connectivity. In this case, switching to the extended identifier format (29 bits) is probably the best option. A satisfactory solution is to reserve the 7 least significant bits in the base identifier to encode CAN-DA, while the remaining 4 bits are set to the function code 1111_2 (not defined in the predefined connection set). In this way, regular CANopen messages are given precedence over CDLC, and the same filtering mask used for them can be exploited. CAN-SA, on which no filtering is carried out, takes 7 bits in the extended identifier, while the remaining 11 bits can be used for other purposes.

8.3.4 Preserving Real-Time Performance

When different kinds of traffic coexist on the same CAN bus, it is generally impossible to completely avoid mutual blocking without resorting to some forms of coordination, e.g., the time-triggered communication paradigm [46]. Then, it becomes important to understand which effect IoC or EoC communication has on real-time (RT) traffic taking place on the same bus. In the following, it is assumed that CAN identifiers are assigned in such a way that RT messages always have *higher* priority than CDLC messages. In other words, real-time and (non real-time) CDLC messages belong to two disjoint sets and any message belonging to the first set has strict precedence over those belonging to the second. As said above, this is the case of the prototype CDLC implementation presented in Section 8.6.1. In general, two sources of blocking can be identified:

1. *Bus-related* blocking, caused by multiple nodes trying to transmit concurrently on the same CAN bus.

2. *Queuing-related* blocking, which may be introduced when the same node generates different kinds of traffic through the same CAN controller.

For what concerns bus-related blocking, it can easily be proved that, if multiple frames compete for transmission in the same arbitration round, the highest-priority message will not suffer from any blocking. Thus, if a RT message competes with any number of CDLC fragments, it will not be delayed at all. Instead, if a RT message is queued for transmission when another message is already being sent on the bus, blocking will occur. The mutual blocking among multiple RT messages was thoroughly analyzed in [19]. Hence, here we focus on the effect of non-RT traffic. When a RT message is queued for transmission while a CDLC fragment is being sent on the bus, the former will be blocked until the next arbitration round, which takes place after the transmission

of the fragment has finished. At that point, the RT message will surely win the arbitration against any other CDLC fragment. As a consequence, the worst-case effect of the whole set of messages used by CDLC can be modeled as a single additional RT stream, whose priority is lower than any RT message and whose messages are queued for transmission back to back. Its effects, including chain blocking, can be analyzed as in [19].

Queuing-related blocking may occur when either the CAN controller or the software device driver only support a First-In, First-Out (FIFO) transmission queue rather than a priority-based queue or multiple queues with different relative priorities. In this case, a RT message entering the queue will suffer blocking from any other messages already buffered in that node at that time, regardless of its priority. Scheduling analysis is still possible in this scenario, as shown in [20].

8.4 Ethernet over CAN (EoC)

The aim of EoC is to make an application programming interface (API) available on CAN nodes that resembles closely the one provided by IEEE 802 protocols. Whole Ethernet frames are encoded by CDLC in EoC, including destination and source MAC addresses (DA and SA). Having CDLC sending on the bus the fragments of every Ethernet frame using the broadcast CAN-DA ensures, in theory, proper EoC communication. In fact, receivers can reassemble all incoming DLPDUs and then decide whether or not they are interested in the related frame by inspecting its (MAC) DA field. However, doing so would cause an excessive interrupt rate, which is typically not acceptable since most CAN nodes have limited processing capabilities. Therefore, some mechanism is needed by nodes in order to discard CDLC fragments related to irrelevant frames, which exploits the frame acceptance filtering function provided by CAN controllers.

In order to carry out the translation between MAC and CAN addresses a *transparent address resolution process* (TARP) is used that resembles the learning and forwarding mechanism in Ethernet bridges. To this extent, an *extended filtering database* (EFDB) is required in C-switches, which consists of a number of entries. Each entry is related to a remote node and includes several information: its MAC address (MAC_ADDR), the port on which it was last heard (PORT_ID), and, implicitly, the related type (PORT_TYPE—either Ethernet or CAN). In the case of CAN ports, the CAN address (CAN_ADDR) of the node bound to MAC_ADDR is also stored. A simplified version of EFDB is maintained by EoC nodes, which only includes information about MAC_ADDR and CAN_ADDR.

TARP operates on each Ethernet frame separately and consists of a forwarding and a learning subprocess. Frames can be either generated by the upper protocol layers (IP) or received from the network (Ethernet or CAN/CDLC).

Forwarding subprocess

Whenever an Ethernet frame has to be dealt with, the EFDB is searched for an entry whose MAC_ADDR field matches DA. Four outcomes are possible:

a) A match is found with the MAC address assigned to the local DL-entity: the frame is delivered to the upper protocol layers.

b) A match is found in the EFDB for a CAN port: a point-to-point fragmented transfer is carried out through CDLC to the specific destination node on the CAN bus using the CAN_ADDR information in the entry as CAN-DA.

c) A match is found in the EFDB for an Ethernet port: in this case, the frame is relayed directly on that port.

d) No match is found or DA specifies a multicast address: *flooding* takes place, i.e., the frame is relayed on all ports (Ethernet and CAN) with the exception of the one on which it was received; the broadcast CAN-DA is used by CDLC.

In order to support flooding, CDLC must be able to operate according to an unacknowledged scheme.

Learning subprocess

This process is used to populate the EFDB and resembles what happens in Ethernet switches. The SA field is inspected in every received Ethernet frame:

a) An entry already exists in the EFDB whose MAC_ADDR equals SA, but it is associated to a different port: the information in the entry are updated.

b) SA is unknown: a new entry is added to the EFDB where MAC_ADDR=SA; PORT_ID is set properly (and PORT_TYPE as well).

If the frame came from a CAN port, CAN_ADDR in the entry is also set. In any case the aging time of the entry is reset.

In order to keep EFDB replicas in different DL-entities on the same CAN bus in a coherent state, whenever a frame is relayed by CDLC to a specific destination (unicast CAN-DA) it is preceded by the transmission of a short *NC-MAC_Source_Announce* (*NC-MSA*) network control message, sent with the broadcast CAN-DA. Its purpose is to notify all the other DL-entities that an entry was either created or updated in the local EFDB, so that the change is propagated to all the other EFDBs. The only information this message has to include—besides CAN-SA—is the SA field of the subsequent Ethernet frame (6 bytes), which fits in the payload of a single CAN message. Upon reception of *NC-MSA*, every DL-entity updates its EFDB using the same rules above concerning reception of Ethernet frames. Doing so is unnecessary when Ethernet frames are broadcast by CDLC on the CAN bus.

In theory, information in EFDB replicas for DL-entities on the same CAN bus might become incoherent following a transmission error that involves a frame or a NC message (because of CAN error globalization, this is unlikely). At any rate, this is not a serious problem: missing an entry creation simply disables selective forwarding (and causes one more flooding) while missing an update increases the chance that the entry is removed because of aging.

As for every Ethernet bridge that complies to IEEE 802.1D, the Spanning Tree Protocol (STP and RSTP) must be included in C-switches so as to avoid loops in the network topology.

8.5 IP over CAN (IoC)

As a matter of fact IP is layered above the transmission network (Ethernet, WiFi, public data networks, etc.) in almost every high-level distributed application. For this reason, a second solution can be devised specifically for this kind of systems, namely IoC, where DLPDUs encode IP datagrams directly. This permits saving 14 bytes with respect to EoC (DA, SA, and ET fields). Most important, unlike EoC MAC addressing is not required: IoC nodes are identified on the whole network using IP addresses. This means that a mechanism is required to translate IP and CAN addresses directly.

Basic IoC implementation is straightforward: IP was conceived to run on different kinds of network and CDLC is just one of them. IoC has to be regarded as a simpler approach that permits CAN nodes to be easily connected to intranets.

8.5.1 Mapping IP and CAN Addresses

The address space for CAN nodes complies to the Classless Inter-Domain Routing (CIDR) scheme [81]. A router provided with both CAN and Ethernet interfaces is typically used to interconnect a CAN bus to the intranet, whose internal architecture resembles IoC devices. This means that it can be implemented on inexpensive embedded platforms.

Either static or dynamic mapping can be used for translating IP and CAN addresses. Assuming that standard CAN identifiers are in use, as discussed in Section 8.3.2, *static mapping* supports IoC networks with a `/27` CIDR prefix, which means that 27 of the 32 bits of the IP address represent the *network number* while the 5 least significant bits encode the *host number*. The host number is mapped one-to-one into the layer-2 CAN address. According to the requirements for Internet hosts [5], a host number of all ones, 11111_2 is reserved in this case for directed broadcasts and directly corresponds to the broadcast address specified in Section 8.3.2.

Due to the presence of a class of hosts that use a non-standard host number of all zeros for directed broadcast, [5] also recommends that Internet hosts recognize and accept both. Following this recommendation has the side effect of making address 00000_2 unavailable for unicast communication when static mapping is in use. This brings the total number of supported nodes to 30 (if no addresses besides the broadcast address are reserved for multicast communication) or 15 (if they are, according to the approach proposed in Section 8.3.2). The IP address of an IoC node is obtained concatenating the routing prefix of the CAN bus and the CAN address of the node. The reverse translation is based on the same principle.

Instead, with *dynamic mapping* IP and CAN addresses are uncorrelated, which means that they are assigned separately—and, possibly, in different stages of system design/deployment. A suitable mechanism is required for address translation. In particular, a streamlined version of ARP may suffice, referred to as *ARP over CAN* (AoC). As in ARP, AoC messages (request and reply, including gratuitous ones) include 4 fields, namely Sender/Target Hardware Addresses (SHA and THA) for encoding MAC addresses, and Sender/Target Protocol Addresses (SPA and TPA) for IP addresses. A cache is also defined whose entries relate IP and hardware addresses.

To retain compatibility, both IoC and EoC must use the same format for AoC messages. SPA and SHA are used by EoC nodes to populate a conventional ARP cache (IP↔MAC) that is layered above TARP and the related EFDB (MAC↔CAN). Unfortunately, IoC nodes do not have

a MAC address. To achieve interoperability with EoC, non-globally unique *forged MAC addresses* (FMA) must be defined for them by concatenating a specific (reserved) 3-byte *IoC-prefix* (FORGED_OUI), a 16-bit CAN bus identifier (CANBUS_ID) and the 8-bit CAN address of the IoC node (CAN_ADDR). Thanks to the prefix, FMAs can always be recognized to belong to IoC nodes. The reserved CANBUS_ID value `0x0000` denotes the *local* CAN bus. The related FMAs are referred to as *locally-unique FMAs* (L-FMA). Their uniqueness is ensured only on the bus to which the nodes are attached.

Each IoC node sets the SHA and THA fields in transmitted AoC messages to the related L-FMAs. Concerning received AoC messages, the CAN address of the sender node is determined by IoC nodes as the CAN-SA field in CAN messages (the values of SHA and THA in the message are ignored). Instead, EoC nodes use SHA and replace THA with their own MAC address before processing. From a practical point of view, a thin software layer can be added between ARP and CDLC, which carries out format translation between AoC and ARP messages—including conversions between CAN addresses (CAN-SA and CAN-DA) and the related L-FMAs (stored in SHA and THA). The above approach permits reusing (most of) the existing ARP implementations.

8.5.2 Layer-3 Enabled Switches

Besides exploiting routers, transmission of IP datagrams inside bridged DL networks that include both Ethernet links and CAN buses can be accomplished also by means of C-switches operating "above" layer 2. Thanks to the learning and forwarding approach, relaying DLPDUs at the data-link layer is more flexible than at the network layer. The problem is that the L-FMAs for IoC nodes are not unique across the whole IP subnet, and hence they cannot be used to forward IoC datagrams out of a CAN bus.

A possible solution is to extend C-SW behavior so that the IP address (IP_ADDR) is used in the place of MAC_ADDR to identify nodes uniquely in the EFDB and carry out forwarding. Such scheme is referred to as *uniform network addressing and traversal* (UNAT) and the related devices *layer-3-enabled C-switches* (C3-SW). UNAT unifies IP and MAC addressing by combining the learning and forwarding approach of TARP and the request-reply paradigm of ARP/AoC.

As depicted in Figure 8.2, 3 kinds of entries are foreseen in the EFDB of C3-SWs, i.e., Ethernet, EoC, and IoC. The last two are implicitly distinguished through the OUI in MAC_ADDR (IoC-prefix). Besides the port they refer to, the following information are stored in each entry, depending on its type:

1. Eth: ⟨IP_ADDR, MAC_ADDR⟩;
2. EoC: ⟨IP_ADDR, MAC_ADDR, CAN_ADDR⟩;
3. IoC: ⟨IP_ADDR, (forged) MAC_ADDR, CAN_ADDR⟩.

Forwarding subprocess

C3-SWs act as proxies. When an IoC-encoded datagram is received, IP-DA is searched in the EFDB. If a match is found the information in the related entry are used. Before forwarding takes place on PORT_ID the following actions are carried out, depending on its type:

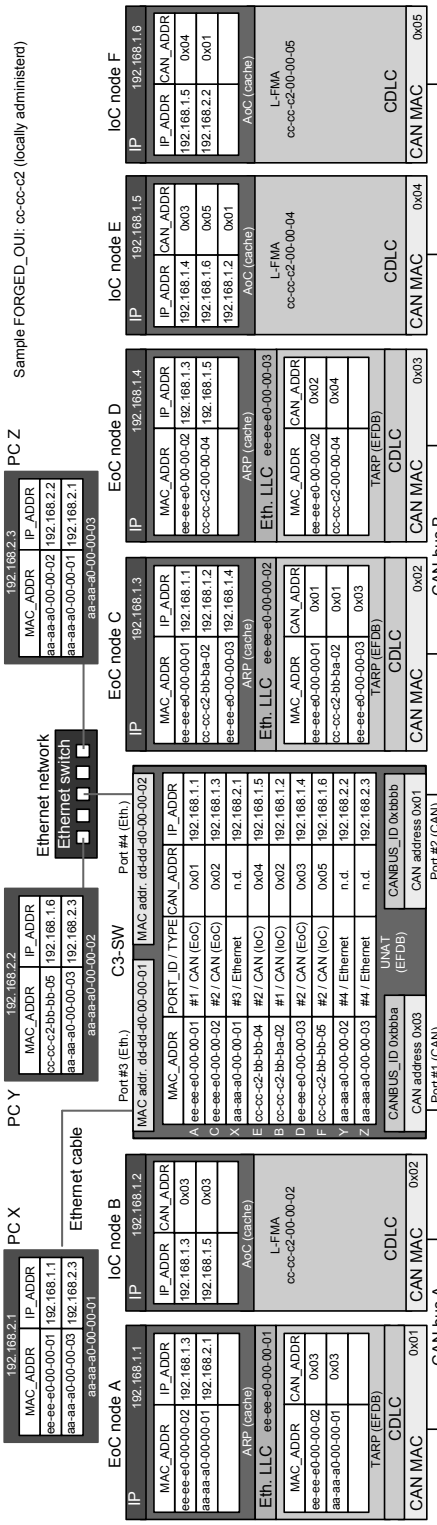

Figure 8.2. Sample network architecture including IoC, EoC, and Ethernet nodes, along with extended filtering databases and ARP/AoC caches.

- Ethernet: the datagram is embedded in an Ethernet frame where DA is set to MAC_ADDR. Since SA is not available, it can be set to either the globally-unique MAC address of the C3-SW or a MAC address purposely forged for CAN-SA so as to be unique on the whole network. The former approach is simpler but it may be weak against IP spoofing attacks [74]. In the latter approach a *network-wide-unique* forged MAC address (N-FMA) is created. To this extent, distinct CANBUS_IDs must be assigned to the CAN buses in the bridged DL network. It is worth noting that only C3-SWs (and not IoC nodes) are aware of CANBUS_IDs. This reduces implementation and network configuration efforts noticeably.

- IoC: the IP datagram is relayed on the target CAN bus directly using CDLC. The CAN_ADDR field of the entry is used as CAN-DA while CAN-SA is set to the CAN address of the C3-SW on the target bus.

- EoC: MAC addresses are set in the same way as for Ethernet and CAN addresses as for IoC. Transmission complies to EoC.

Instead, when an IP datagram is received encapsulated in an Ethernet frame—possibly on a CAN port through EoC—the EFDB is searched for either an Ethernet or EoC entry whose MAC_ADDR matches DA. If a match is found forwarding is carried out using Ethernet/EoC rules. Otherwise the EFDB is searched again for an IoC entry whose IP_ADDR matches IP-DA. If it is found, the datagram is relayed on CAN (using CDLC) in the same way as for IoC-encoded datagrams.

In both cases, when a match is not found in the EFDB the datagram is relayed on all ports of the C3-SW (flooding). On CAN ports transmission takes place as IoC using the broadcast CAN-DA—this is because EoC nodes can be easily enabled to decode IoC datagrams while the opposite is not true. Flooding of IP datagrams is indeed a rare event, since they are preceded by ARP messages, and is likely due to the misalignment of the EFDB and ARP caches on end nodes. To take care of this, an AoC request is broadcast by the C3-SW on all its CAN ports after a datagram flooding, asking for the related IP-DA. If the original datagram was encoded as IoC, a similar ARP request is sent on Ethernet ports as well. When the node addressed by IP-DA replies the EFDB in the C3-SW is coherent again.

Learning subprocess

UNAT ensures complete interoperability between Ethernet, EoC, and IoC in bridged DL networks. The related operations are carried out entirely by C3-SWs, so that implementation of IoC nodes remains as simple as possible. Generally speaking, the EFDB in C3-SWs acts as both filtering database and ARP cache. Unlike TARP in EoC, the learning process used by UNAT to update the EFDB relies only on the ARP/AoC messages relayed by the C3-SW:

- an AoC message is received from CAN: if SHA is a L-FMA the originator is an IoC node. The C3-SW forges a N-FMA for it and replaces SHA in the message body. At the same time an IoC entry is created/updated in the EFDB as ⟨IP_ADDR=SPA, CAN_ADDR=CAN-SA, MAC_ADDR=N-FMA⟩. Otherwise, the originator is an EoC node and the entry is set to ⟨IP_ADDR=SPA, CAN_ADDR=CAN-SA, MAC_ADDR=SHA⟩.

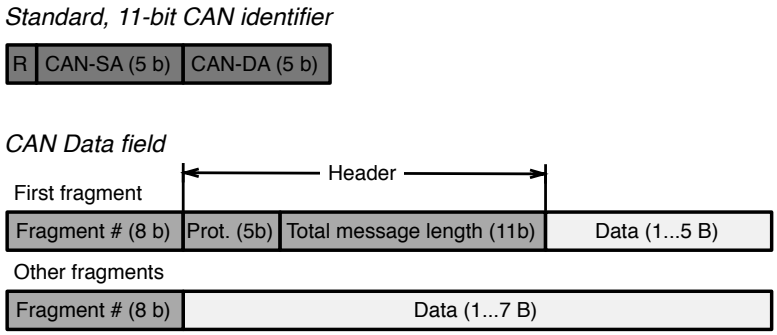

Figure 8.3. CAN identifiers and frame format in the CDLC protocol.

- an ARP message is received from Ethernet: an Ethernet entry is created/updated in the EFDB as ⟨IP_ADDR=SPA, MAC_ADDR=SHA⟩.

Transmission of ARP requests is carried out in broadcast on the bridged DL network (ARP messages are blocked by routers). In the case of AoC requests the broadcast CAN-DA is used. Instead, ARP/AoC replies are delivered as unicast messages. The EFDB in the C3-SW is used to forward them properly, as happens to IoC datagrams. While forwarding ARP messages from Ethernet to CAN, they are translated by the C3-SW to AoC (and vice-versa). When an AoC message generated by an IoC node is relayed (as ARP) on Ethernet, the related N-FMA is used for DA (the same as SHA). Instead, in AoC messages sent/relayed on CAN the CAN address of the port of the C3-SW connected to that bus is used as CAN-SA.

The effect of AoC messages on end nodes depends on their type. IoC nodes that receive an AoC message simply update their AoC cache with the entry ⟨IP_ADDR=SPA, CAN_ADDR=CAN-SA⟩. Conversely, EoC nodes update their ARP cache as ⟨IP_ADDR=SPA, MAC_ADDR=SHA⟩ and, through TARP, their EFDB as ⟨CAN_ADDR=CAN-SA, MAC_ADDR=SHA⟩. Besides ARP caches in end nodes, each ARP request/response pair sets all the relevant EFDB entries in the traversed C3-SWs in both directions, so that all the subsequent IP datagrams can be forwarded correctly to the intended destination.

8.6 IoC Prototype Design and Implementation

This section discusses how a prototype system has been designed and implemented. First of all, the CDLC protocol implementation, common to both IoC and EoC, is presented in Section 8.6.1. Due to protocol overhead and implementation complexity considerations, as shown in Section 8.6.2, the choice for the first prototype fell on IoC, which has then been integrated in the open-source lwIP protocol stack [23], as a network interface driver module, presented in Section 8.6.3.

8.6.1 CDLC Protocol Implementation

As shown at the top of Figure 8.3, the implementation makes use of standard CAN identifiers to hold the 5-bit CAN-SA and CAN-DA plus one reserved bit, denoted as R in the figure. CAN-SA

(the *source* address) comes before CAN-DA, so that in arbitration it implicitly sets the relative priority of frames transmitted by different CAN nodes in case of bus contention. Hence, an appropriate choice of CAN-SA provides a simple way of assigning different, fixed priorities to IP traffic originating from distinct nodes.

Since the prototype implementation's goals were mainly *feasibility* and *performance* evaluation, the static address mapping approach described in Section 8.5.1 was adopted. There is in fact no doubt that an ARP-like protocol can be implemented with reasonable effort within modern protocol stacks (that already provide ARP themselves). Moreover, address mapping is used only in the very first stage of the communication between peers, and hence, it marginally affects communication performance.

Referring back to Figure 8.3, the CDLC protocol transfers an IP datagram as a sequence of *fragments*, as specified in Section 8.3. All fragments start with a 1-byte *fragment number* at the beginning of the CAN data field, which indicates that up to 256 fragments can be supported. The first fragment has an additional 2-byte *header* containing a 5-bit *protocol identifier* and an 11-bit field containing the *total length* of the IP datagram to be transferred. The protocol identifier is set to 00001_2 for IoC while the other values are reserved for other protocols, namely EoC and network control, supported by CDLC. It is assumed that at most one pending transmission may exist between any two nodes at any given time. For this reason, the fragments bear no explicit information to determine whether or not they pertain to the same datagram.

In all fragments, the rest of the data field holds part of the IP datagram, up to 5 bytes in the first fragment and up to 7 bytes in the others. This corresponds to a maximum IP datagram size of $255 \cdot 7 + 5 = 1790$ bytes, a value bigger than the Ethernet upper limit [37], that is, 1500 bytes. However, the value has been capped to 1500 bytes to prevent IP-level fragmentation and reassembly, and hence, multiple fragmentation and reassembly points in the protocol stack when a datagram is routed from a CAN-based network to an Ethernet-based one.

In the design of CDLC, it has been assumed that hosts are able to reliably receive frames from a CAN bus working at full utilization. This assumption is reasonable given the computing power nowadays available even in very low-cost microcontrollers, like the one adopted for the implementation [73], and it simplifies protocol design. Namely, the protocol does not comprise any flow control mechanism because the CAN bus caps the maximum incoming fragment rate by itself.

Furthermore, neither end-to-end acknowledgment nor retransmission mechanisms have been included in CDLC, in order to simplify broadcast communication. A fragment loss, due to a bus error or transient overload, leads the receiving side to discard the whole IP datagram it belongs to. In this situation, some transport protocols, like TCP, will retransmit the datagram while others, like UDP, leave this duty to the application layer. From this point of view, CDLC behaves like an Ethernet-based link. Actually, the CAN bus (upon which CDLC is layered) does provide extra features with respect to Ethernet, for instance automatic fragment retransmission upon detectable bus errors, which are able to lower the probability of fragment loss. As a result, in CDLC, the frame transfer sequence becomes extremely straightforward:

- On the transmitting side, the fragments belonging to a certain IP datagram are transmitted in sequence by means of a single transmit queue, so that the CAN controller does not reorder them.

- On the receiving side, the reassembly automaton assumes that all fragments coming from the same source node shall be received in order. Any mismatch between the expected and actual fragment numbers leads the receiver to discard the current datagram and wait for the first fragment of the next one.

It must also be noted that, when considering the IP protocol, the absolute minimum datagram size that will ever be encountered is 20 bytes, that is, the minimum size of the IP header. Given the frame format shown in Figure 8.3, the number of fragments needed to transfer such a datagram using IoC is 4. Therefore, it is easy to prove that the minimum number of consecutive fragment losses that leads CDLC to erroneously reassemble together fragments belonging to different datagrams is 4, too. Given the extremely low probability of this error scenario, the protocol does not include any countermeasure against it.

8.6.2 IoC and EoC Protocol Overhead

The overhead introduced by IoC and EoC approaches can be evaluated by considering the number of bytes m'_{CAN}, which is needed to transfer a TCP segment of m bytes through the CAN interface:

$$m'_{\text{CAN}} \simeq (m + b) \cdot \frac{8}{7} \cdot \frac{114}{64} = 2.04\,(m + b) \quad . \tag{8.1}$$

In the equation above, b represents the total size of the protocol headers needed by TCP, IP, and possibly Ethernet (if EoC is used). More specifically, $b = 40$ for IoC (considering 20 B for the IP header plus 20 B for the TCP header). For EoC, $b = 54$ because DLPDUs transferred on the CAN bus also include a 14 B Ethernet header. The additional overhead implied by EoC (through a higher value of b) is the same for other transport protocols and this justifies the choice of IoC for the prototype implementation.

In Equation (8.1), 8/7 represents the approximate overhead introduced by CDLC through the presence of a 1-byte fragment number for every 7 data bytes, except in the first and last fragments. Namely, the first fragment also contains a 2 B CDLC header besides the fragment number while the last fragment may contain less than 7 data bytes. It should be remarked that the approximation is still accurate because the minimum $m + b$ is 40 B (namely, for an empty TCP segment in IoC), which corresponds to 6 fragments.

Last, 114/64 approximates the average overhead of CAN framing, CRC, IFS, and bit stuffing for 8-byte random payloads. As it is shown in Chapter 4, Table 4.2, the mean transmission time $\overline{C}$ of messages with 8-byte random payload and without applying any encoding method (depicted as $W8_{\text{CAN}}$ in the table) is 110.8 bit time. It is worth noting that, the 3-bit IFS was not taken into account there. This gives an overall average transmission time of 114, when rounded up to the next integer number of bits. The worst case can be modeled in a similar way, considering that the maximum number of bits needed to transfer a 64-bit payload is 135, of which 3 bits are for the IFS and 132 is calculated as shown in Section 4.1.3, Equation (4.3).

As mentioned in Section 8.2.3, another reason to choose IoC for the prototype is that several open-source protocol stacks directly support the integration of a network interface driver for IP datagrams, and this is the only additional component required. On the contrary, the implementation of EoC would be more complex because, as discussed in Section 8.4, it requires the implementation of additional protocols, typical of an Ethernet bridge.

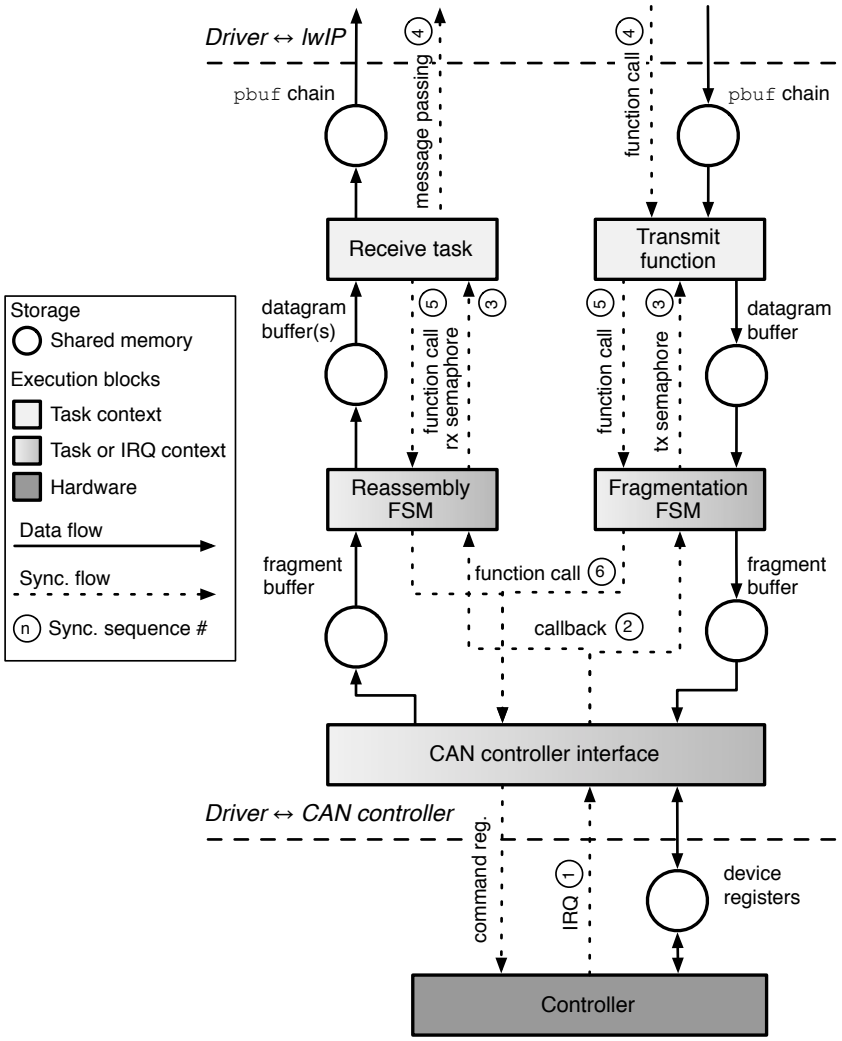

Figure 8.4. Internal structure of the lwIP CAN interface.

8.6.3 Integration of IoC in a TCP/IP Protocol Stack

The internal structure of the IoC driver for the lwIP protocol stack is shown in Figure 8.4. The driver ↔ lwIP interface is specified by lwIP. On the transmit side (right part of the figure), a function call, ④ in the figure, is used to request a datagram transmission. If the transmit path is idle, the transmit function immediately starts the transmission by means of function call ⑤ to the lower layer. Otherwise, synchronization between the two layers is achieved by means of semaphore primitive ③.

On the receive side (left part of the figure), a message passing interface ④ allows the driver to "push" incoming datagrams into the protocol stack upon notification that a datagram has been completely received ③. After that, a downward function call ⑤ synchronizes with the receive path and enables it to reuse the datagram buffer for reception. Since lwIP interactions must be

performed from a task context, a separate *receive task* is in charge of them.

Datagrams are held in a shared `pbuf` chain (a linked list of dynamically-allocated memory buffers defined by lwIP) on each side. On the transmit side, `pbuf` chains are allocated by lwIP itself and passed to the interface driver. On the other hand, the receive task is responsible of pre-allocating `pbuf` chains and filling them with incoming IP datagrams.

The driver ↔ CAN controller interface is defined by hardware. Downward synchronization and data transfer are implemented by device register access (the CAN controller in use is not DMA-capable), while upward synchronization is provided by interrupt requests ①. A *CAN controller interface* module provides device-independent access to these hardware features.

CDLC, by itself, is implemented by means of two Finite State Machines (FSMs) shown in the middle of Figure 8.4. They can be invoked by the transmit function (from a task context) and the receive task, as well as the CAN controller interface through a callback ② (from an IRQ context, when a fragment has been received or the controller is ready to accept a new fragment for transmission). Synchronization with the transmit function and the receive task takes place by means of two semaphores ③, located at the boundary between the IRQ and task contexts. Function calls ⑥ to the CAN controller interface are used to interact and synchronize with hardware.

The preliminary implementation leaves room for further optimizations for what concerns memory management. As shown in Figure 8.4, IP datagrams being transmitted are currently copied multiple times along the transmit path, and the same happens on the receive path. For this reason, the performance results described in Section 8.7 can likely be improved further.

For what concerns code complexity, the preliminary implementation consists of about 5500 lines of C code (of which 3000 are architecture-independent and 2500 are architecture-dependent).

8.7 IoC Performance Evaluation

8.7.1 Experimental Setup

In the prototype, IoC has been implemented on a hardware development board based on the NXP LPC1768 microcontroller [73]. The combination of the lwIP protocol stack and this microcontroller is convenient for performance comparison because it is in widespread use and its performance, at least for what concerns Ethernet-based networks, has been investigated in previous work [17].

In the experiments, two test programs set up a unidirectional data exchange, using the TCP protocol. This kind of exchange is representative of application-layer protocols in which data predominantly flows in one direction (like HTTP and FTP). Since the effect of IoC traffic on real-time performance has already been analyzed previously, in the following we will just focus on the measurement of the *average data transfer rate*, which is a main performance index for non real-time traffic. Data transfer takes place through a single connection. The maximum number of connections the system may support has not been evaluated because it mainly depends on the available memory in lwIP and the way it is configured rather than the kind of link. In any case, all connections to the same node share the available bandwidth.

The total amount of data to be exchanged is fixed and set to 1 MB in all experiments. Data is passed to the protocol stack in chunks of d bytes each time. The value of d is configurable and it directly affects the interaction overheads with the protocol stack, that is, the number of calls

needed to transfer a given total amount of data. As shown in the following, d may affect TCP segmentation indirectly, in particular the possible values of m, which represents the TCP segment size in byte. However, this is an internal choice of the protocol stack, for example lwIP or any other protocol stack in use.

The first round of experiments was aimed at evaluating how effective IoC is, with respect to a standard Ethernet-based connection. To this purpose, two LPC1768-based boards, hosting the test programs, are directly connected by means of either a CAN or an Ethernet link and the average data transfer rates through them are measured.

A second set of experiments was aimed at determining communication performance when the IP *forwarding* mechanism is in use between a CAN-based and an Ethernet-based network segment. To this purpose, the same test programs were used to exchange data between an Ethernet-equipped Linux PC and an LPC1768 board implementing IoC, with the interposition of another LPC1768 board working as a *router* based on lwIP, as shown in Figure 8.1a. Quite intuitively, IP forwarding is carried out by the CDLC-enabled router. This test configuration also provided the opportunity to confirm the interoperability of the router implemented on the embedded board with the Linux protocol stack and assess the impact of the data-originating protocol stack on communication performance.

8.7.2 Link-Level Performance

For what concerns the first round of experiments, the *measured average* TCP data transfer rates through an IoC (r_{CAN}) or a standard Ethernet link (r_{ETH}), are listed in Table 8.1 and plotted in Figure 8.5 as a function of the chunk size d. To make the comparison easier, the data transfer rate is plotted with reference to two calculated values, namely the *raw speed* of the physical link v and the *approximate maximum* transfer rate u. More specifically, $v_{\text{CAN}} = 500\,\text{kbps}$ for CAN and $v_{\text{ETH}} = 100\,\text{Mbps}$ for Ethernet, corresponding to $62.5\,\text{kB/s}$ and $12500\,\text{kB/s}$ respectively, as shown in the figure. u is calculated by taking into account data-link, network, and transport-layer protocol overheads as

$$u_{\text{CAN}} = v_{\text{CAN}} \cdot \frac{m}{m'_{\text{CAN}}} \quad \text{and} \quad u_{\text{ETH}} = v_{\text{ETH}} \cdot \frac{m}{m'_{\text{ETH}}} \ , \tag{8.2}$$

where m'_{CAN} and m'_{ETH} represent the total number of bytes to be transmitted on the link in order to transfer a TCP segment of m bytes through IoC and Ethernet, respectively. Neglecting non-piggybacked ACK traffic in overhead calculations, m'_{CAN} can be approximated by using (8.1) with $b = 40$ as

$$m'_{\text{CAN}} \simeq (m + 40) \cdot \frac{8}{7} \cdot \frac{114}{64} = 2.04\,m + 81.43 \ . \tag{8.3}$$

Similarly, the number of bytes m'_{ETH} needed to transfer the same segment through Ethernet is

$$m'_{\text{ETH}} = \begin{cases} 84 & m < 6 \\ m + 40 + 38 = m + 78 & m \geq 6 \end{cases} \ . \tag{8.4}$$

In both cases, the overhead of 40 B is due to IP and TCP headers. Instead, in (8.4) the additional overhead of 38 B takes into account the combined size of the Ethernet header, FCS, Preamble, and Interframe gap [40]. Finally, (8.4) also considers the minimum total Ethernet frame length constraint of 84 B.

d (B)	1	4	16	64	256	1024
r_{CAN} (kB/s)	0.78	2.91	8.90	18.02	24.76	26.08
r_{ETH} (kB/s)	5.75	22.75	88.45	332.61	1088.03	1284.71

Table 8.1. Measured link-level performance, TCP traffic.

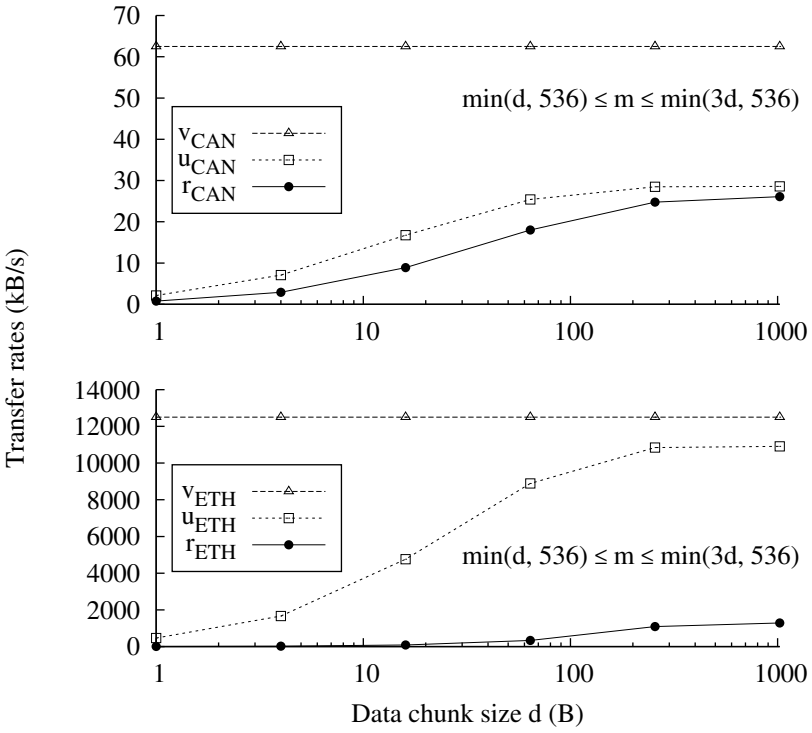

Figure 8.5. Link-level performance, IoC vs. IP over Ethernet, TCP traffic.

For a certain d, the range of the TCP segment size m observed experimentally is $\min(d, 536) \leq m \leq \min(3d, 536)$. As already mentioned in Section 8.7.1, the variability of m mainly depends on the segmentation algorithms of the lwIP protocol stack. Regardless of d, lwIP has been configured to limit m to 536 B to save memory. This upper limit is also the minimum legal value set forth in the IP protocol specification. For a certain d, the calculation of u_{CAN} and u_{ETH} has been approximated by excess using the largest value of m observed.

The results show that the two kinds of link behave in two very different ways. On one hand, the gap between u and v is bigger for the CAN-based link than the Ethernet-based link. This is because the network overhead plays a more important role in the former one, as can also be seen from (8.3) and (8.4). On the other hand, as shown in Figure 8.5, the CAN-based link is able to achieve nearly fully utilization as the average data transfer rate r_{CAN} is quite close to the approximate maximum transfer rate u_{CAN}. Instead, this is not the case for the Ethernet-based link because software overheads prevail. Those results are due to two different phenomena:

1. The value of d affects performance in two ways. First of all, smaller values of d correspond to smaller values of m used by lwIP. In turn, this lowers the performance according to (8.2)–(8.4). Secondly, d is inversely proportional to the number of lwIP calls needed to transfer a given total amount of data. Hence, smaller values of d introduce a higher software overhead due to the inter-task communication between the test programs and the protocol stacks.

2. Across the two kinds of link, the same value of d corresponds to two very different ratios between software overheads and communication times. In fact, software overheads depend on d (and also on the CPU speed) but are independent on the link type, at least above the data-link layer. Instead, communication times mainly depend on two link-dependent properties, namely, link speed and data-link overheads (including CDLC overheads in the case of CAN).

Their combined effect is rather counterintuitive because the speed advantage of Ethernet-based versus CAN-based communication, namely $r_{\mathrm{ETH}}/r_{\mathrm{CAN}}$, when carrying TCP traffic is much lower than the raw ratio between their link speeds, that is, $v_{\mathrm{ETH}}/v_{\mathrm{CAN}} = 200$. In fact, as can be derived from Table 8.1, the ratio between data transfer rates only goes from about 7.4 times (when $d = 1$) up to about 49 times (when $d = 1024$).

8.7.3 Forwarding Performance

The results of the second round of experiments, aimed at determining the performance of an integrated network involving the IP *forwarding* mechanism between a CAN-based and an Ethernet-based network segment, are shown in Table 8.2 and Figure 8.6. The first row of the table pertains to the experiments in which data originate from the lwIP-based IoC end node and flow from the CAN-based to the Ethernet-based segment, while the second corresponds to data that originate from the Linux protocol stack and flow in the opposite direction. In Figure 8.6, the values of v and u have been calculated with respect to CAN, which is the slower segment.

With respect to the first set of experiments, part of the TCP processing has been moved onto the PC where much higher computing power is available. For instance, when data flows from the Ethernet-based to the CAN-based segment, transmit-side TCP processing is moved onto the PC. In addition, due to the unidirectional data flow, the receive-side TCP processing still performed by lwIP is limited to passing incoming data to the application and sending acknowledgments back. Hence, overall TCP processing is mainly performed by the Linux protocol stack rather than lwIP. On the other hand, when data flows in the opposite direction, receive-side TCP processing is moved onto the PC. Even if IP forwarding is involved in both cases regarding the first set of experiments, it affects communication performance lightly since it doesn't involve the transport layer and *no* memory-to-memory copies are needed.

When lwIP is sending data, the dependency of $r_{\mathrm{C}\to\mathrm{E}}$ on d (also through m) is pronounced because it still includes both phenomena discussed in Section 8.7.2. On the contrary, when the Linux protocol stack is sending data, the dependency is less pronounced as the average transfer rate $r_{\mathrm{E}\to\mathrm{C}}$ is always close to $u_{\mathrm{E}\to\mathrm{C}}$. This is because the Linux protocol stack keeps the TCP segment size constantly at $m = 536$. However, as shown in Table 8.2, for small values of d, there is still a slight performance loss. This is because the protocol and link level overheads are now constant as m is fixed, while the dependency of inter-task communication overhead on d is still the same

d (B)	1	4	16	64	256	1024
$r_{C\to E}$ (kB/s)	0.65	2.49	7.97	17.80	24.07	26.43
$r_{E\to C}$ (kB/s)	25.18	26.12	26.36	26.50	26.46	26.44

Table 8.2. Forwarding performance, TCP traffic.

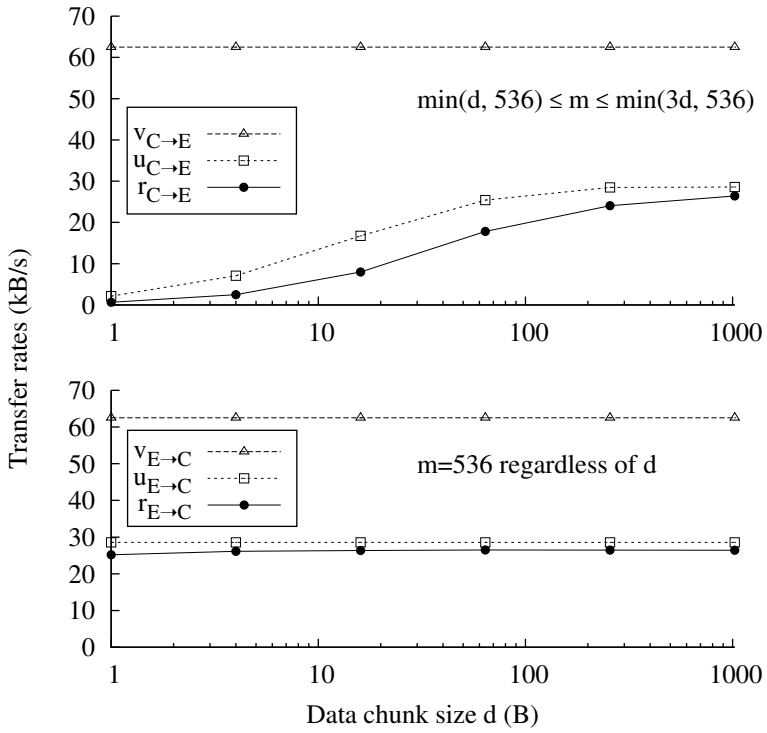

Figure 8.6. Forwarding performance, TCP traffic.

as before. What's more, since Linux aggressively gathers user data to build larger TCP segment, especially for small values of d, this introduces an additional source of overhead. But the last two kinds of overhead do not lead to dramatic performance loss because of the PC CPU speed advantage.

With respect to the previous set of experiments (Table 8.1), the data transfer rate $r_{E\to C}$ is almost always greater than the CAN-only transfer rate r_{CAN}. This is because transmit TCP processing on Linux plus lwIP forwarding is more efficient than transmit TCP processing on lwIP, taking into account that lwIP forwarding is lightweight while the PC CPU is at least one order of magnitude faster than the LPC1768.

Instead, the situation when data flows from the CAN-based segment to the Ethernet-based segment is more complex. The data transfer rate $r_{C\to E}$ is mainly bounded by TCP processing on the transmit side (still performed by lwIP) and communication overheads that depend on d through m. Hence, the general behavior of $r_{C\to E}$ is still the same as r_{CAN}. The slight speed advantage of the router configuration when $d = 1024$, and its disadvantage for smaller values of d, can be justified

by the balance between the overhead incurred in receive-side TCP processing and the forwarding overhead. In fact, with respect to r_{CAN}, the receive-side TCP processing is much more efficient on the PC rather than on the LPC1768, while the forwarding overhead is inversely proportional to m and becomes higher for smaller values of d.

It is worth mentioning that the integration of the CAN-based segment and the Ethernet-based segment did not hinder the performance because the end-to-end data transfer rates $r_{\mathrm{C}\rightarrow\mathrm{E}}$ and $r_{\mathrm{E}\rightarrow\mathrm{C}}$ are still close to what a direct CAN link can provide, that is r_{CAN}.

Summary

In this chapter, a network architecture for the seamless integration of CAN in IP-based Intranets has been proposed. Rather than defining a specific solution or protocol, attention has been focused on sketching a generic framework for achieving integration. At the same time, the inherent benefits and shortcomings stemming from design choices like the communication layer at which integration takes place (Ethernet versus IP), and how to support interoperability between those choices, have been highlighted. Experimental results derived from a prototype implementation of one of the proposed designs show that it is not only feasible, but its TCP data transfer performance on a popular class of embedded microcontrollers compares very favourably with respect to the raw ratio between CAN versus Ethernet link speeds.

Chapter 9

Special Purpose Protocol Support

In the previous chapter, the flexibility of CAN has been extended to support the most widely used protocol in the non real-time domain, namely the IP protocol, which enables the integration of CAN into Intranet and introduces CAN into even broader application scenarios. At the same time, the flexibility of CAN can be further enhanced by making it capable of delivering different kinds of real-time traffic as well, for instance Modbus, the one to be discussed in this chapter.

Modbus is an application layer communication protocol widely adopted in industrial automation (virtually, all vendors of industrial remote I/Os support Modbus) and more recently, also in building automation (for example, temperature/humidity sensors, remote control of heating and air conditioning equipment and so on). However, at the physical layer, it is supported only by TIA/EIA–485 bus and Ethernet. TIA/EIA–485 is extremely slow whereas Ethernet requires extra cabling and more complex topology, which add additional complexity in terms of both hardware and software. Instead, CAN represents a tradeoff in between.

In this chapter, a Modbus adaptation layer for the Controller Area Network (called Modbus CAN) is proposed, formally verified, and evaluated experimentally. The results show that, on a typical low-cost embedded system, its performance compares favorably with respect to a Modbus TCP implementation, based on a 100 Mbps Ethernet, carried out on the same system and using the same protocol stack.

9.1 Existing Support for Modbus

Although the Modbus protocol [60] has been introduced more than 20 years ago, it is still very popular nowadays at both the field and SCADA levels, thanks to its simplicity and its open, royalty-free nature [58, 59]. The original versions of the protocol, known as Modbus ASCII and Modbus RTU [62], are based on a TIA/EIA–485 [93] (formerly called RS485) physical level channel, at a typical data signalling rate of 9600 or 19200 bps. A TIA/EIA–232 channel may also be used, but only for short (less than 20 m) point-to-point interconnections. At a later time, also to work around the inherent speed limitations of the TIA/EIA–485 channel, Modbus TCP [61] was introduced. The new specification supports Modbus messaging on TCP/IP, which is typically layered on top of an Ethernet network.

On one hand, this innovation enabled Modbus-based applications to take advantage of the obvious speed advantage of Ethernet with respect to TIA/EIA–485 and readily incorporate further developments of the Ethernet technology. Another important advantage, from the software overhead point of view, is the possibility of switching from the "one interrupt per character" paradigm (typical of asynchronous serial line controllers for TIA/EIA–485) to "one interrupt per frame" (typical of more sophisticated controllers). This is because interrupt handling overhead is often the main limiting factor that prevents the use of higher line speeds, especially on low-cost microcontrollers. Although a Modbus-oriented ASIC[1] for TIA/EIA–485 is certainly feasible and would lift most of the communication burden from the main CPU, unfortunately none are commercially available today.

On the other hand, from the design point of view, the introduction of an Ethernet network also brings new challenges, namely, the requirement of using a star topology, point-to-point cabling, and active[2] network equipment. With respect to the shared bus topology based only on passive components typical of TIA/EIA–485, the extra complexity and cost may be unwelcome in low-cost applications.

Then, it becomes of interest to assess how the Modbus protocol can be layered on top of other kinds of network. The Controller Area Network (CAN) [44] is a suitable candidate because it is still based on a shared, passive bus topology, but it operates at a data rate of up to 1 Mbps and its controllers generate one interrupt per frame. Hence, it is definitely faster than TIA/EIA–485. Moreover, CAN is well established in the networked embedded systems area (as shown, for instance, by the multitude of CANopen [7] application profiles nowadays available) and is already well known to designers and practitioners. From the hardware and software design point of view, CAN technology is also very simple and fast to adopt because CAN controllers are already embedded into most popular off-the-shelf microcontrollers. The only external component needed for communication is a transceiver.

This chapter describes the design and formal verification of a Modbus-*specific* adaptation layer on CAN (called Modbus CAN), whose main purpose is to fragment and reassemble Modbus Protocol Data Units (PDUs) (up to 253 bytes in length) to fit them into CAN frames (that hold at most 8 bytes of payload). The goal of the design is to take the best possible advantage from the known properties of the Modbus application layer protocol to streamline the design, simplify its implementation, and enhance its performance. This design choice was also helpful to carry out formal verification in an effective yet straightforward way.

It is worth noting that the proposed protocol is radically different from what commercially available Modbus to CAN *gateways* or *bridges* implement. In fact, these devices act as *protocol converters* and transform specially-crafted Modbus requests into CAN messages and vice versa. Many of them also provide buffers to store data observed on the CAN bus and make them available on Modbus upon request but, in any case, they do not support the real-time transfer of arbitrary PDUs between two Modbus agents through the CAN bus.

[1] Application Specific Integrated Circuit

[2] *Active* means that it needs electrical power to work. As a matter of fact, in order to connect Ethernet nodes, switches are required and they are active. However, this also means one more cable is needed to bring power to switches. From the system point of view, this could be another failure point.

9.2 Modbus CAN Protocol Design

9.2.1 General Design

The general design of Modbus CAN remarks the known characteristics of the Modbus application protocol [60] and takes advantage of them to simplify the adaptation layer protocol in several key areas. Further simplifications have been achieved by looking at the properties of the Modbus RTU serial line protocol for the TIA/EIA–485 channel [62]. The adaptation layer has been designed to provide exactly the same properties, and no more. In particular:

1. Modbus strictly follows the *master–slave* communication paradigm and supports *one* master for each segment. Therefore, as it is done for the TIA/EIA–485 channel, there is no need to assign an address to the master and explicitly transmit it. For the same reason, simultaneous transmission attempts by different nodes never occur.

2. Split bus transactions are not supported by Modbus. As a consequence, there can be only up to *one* outstanding transaction at any given time. This property guarantees that at most one instance of the fragmentation and reassembly state machines will ever be needed on any node, either master or slave.

3. No confirmation mechanism is implemented by the Modbus RTU protocol, because the Modbus *request/reply* sequence already provides application-layer confirmation to the master, and no confirmations are needed on slaves. Similarly, Modbus CAN confirms neither the transmission of whole Modbus messages, nor their individual fragments, except for what is provided at the CAN bus level.

9.2.2 Addressing and Usage of CAN Identifiers

A Modbus PDU [60] consists of a one-byte *function code*, from 0 to 252 bytes of *data*, and is complemented by an application-layer *slave address*. For request PDUs, the address indicates the target slave. For reply PDUs, it identifies the slave that generated the reply. Valid unicast addresses lie in the range from 1 to 247. The special address 0 denotes a broadcast request that can be sent only by the master and to which no replies are allowed. Addresses from 248 to 255 are reserved.

Since the maximum size of a Modbus PDU is greater than the maximum size of the data field in a CAN frame, the most important activity to be performed by Modbus CAN is to *fragment* the Modbus PDU to be transmitted into multiple CAN frames and, conversely, to *reassemble* those fragments at the receiving end.

As illustrated in Figure 9.1 (next page), the adaptation layer protocol places the slave address in the 8 least-significant bits of the CAN identifier. No further address mapping is performed because it is assumed that Modbus CAN will not coexist with other CAN-based protocols. Being embedded in the identifier, the slave address is transmitted with every fragment and can be used to distinguish among fragments directed to different slaves (for requests) or coming from different slaves (for replies). By definition, those fragments cannot belong to the same message, and hence, they shall not be merged during reassembly. Moreover, the transmission of the slave address in

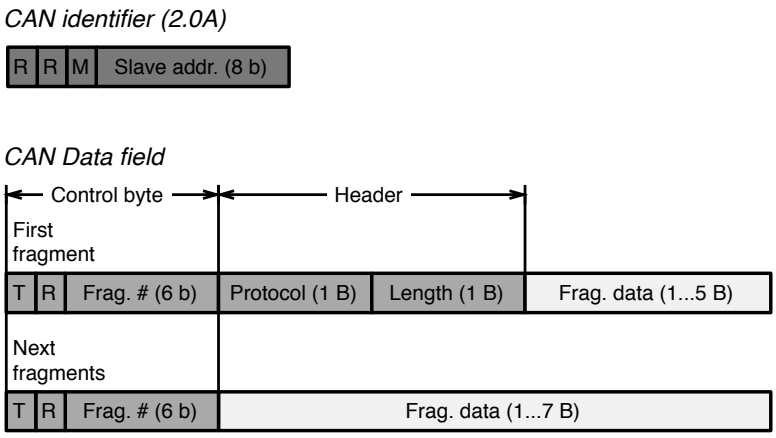

Figure 9.1. MODBUS CAN fragment format.

all fragments allows an effective use of CAN acceptance filters—whose logic is usually based on matching (part of) the identifier with a set of predefined values—on the slave side. No filtering can be performed on the master side because the master shall accept messages from all possible slaves.

In CAN 2.0A frames, 3 additional bits are available in the identifier. Of them, the least significant bit (M in the figure) is used to tell apart fragments sent by the master and belonging to a request (M=1) from fragments sent by a slave and belonging to a reply (M=0). The M bit avoids the failure of the CAN arbitration mechanism if the transmission of a fragment directed to a certain slave is attempted together with a fragment coming from it. In fact, without the M bit, these two fragments would bear the same identifier. This may occur, for instance, if the application-level timeout mechanism is misconfigured and the master transmits a new request to the same slave after a timeout, while the slave is replying to the previous one. The remaining two bits (R in the figure) can be used as a priority tag.

9.2.3 Control Byte, Header, and Data

Every fragment carries a *control byte* at the beginning of the data field, which contains control information to guide the reassembly process.

- The least significant 6 bits hold a *fragment number*. Fragments belonging to the same MODBUS PDU are numbered consecutively, starting from 0.
- The R bit is reserved to expand the fragment number. It shall be transmitted as zero and ignored by receivers.
- The most significant bit is the T (toggle) bit. It is used to tell apart fragments belonging to two different MODBUS PDUs sent in succession by the same node and carrying the same slave address.

The first (or only) fragment of a MODBUS PDU can be easily identified because its fragment number is 0. In this fragment, an additional 2-byte *header* contains a *protocol* identifier (0 for

Modbus) and the *length* of the Modbus PDU being transferred, in bytes. Part of the protocol identifier byte can be reassigned to extend the length field, if necessary, adopting a big-endian encoding of multibyte values, as it is done elsewhere in Modbus.

In all fragments, the remaining part of the data field (up to 5 bytes in the first fragment and up to 7 bytes in the others) contains part of the Modbus PDU. If the size of the Modbus PDU does not exceed 5 bytes, it will be transported in a single fragment. For multi-fragment PDUs, it is recommended that the maximum CAN data field of 8 bytes is used for all fragments, except the last one, for better efficiency.

As it is currently defined, the protocol can transport PDUs of up to 255 bytes in length (limited by the width of the PDU length field), which is adequate for Modbus. By extending the fragment number field to 7 bits and the length field to 10 bits, the same protocol can support PDUs up to $127 \cdot 7 + 5 = 894$ bytes. If necessary, the maximum length can be further extended by stipulating, for instance, that the fragment number is recycled modulus 2^6 and using the seventh bit to uniquely tag the first fragment.

As required by the specification, CAN controllers automatically append a 15-bit CRC field (not shown in the figure) to every CAN frame upon transmission, and check it upon reception. This mechanism adequately replaces the Modbus RTU CRC field, which covers the whole (unfragmented) PDU and has got a similar probability of residual errors.

9.2.4 Transfer Sequence

The Modbus CAN transfer sequence will be specified in Section 9.4, with the help of a formal protocol model. Informally speaking, the initiator of a Modbus PDU transfer (either the Modbus master or one of the slaves) transmits all fragments sequentially using a single transmit queue and setting the CAN identifier, control byte and header fields as shown in Figure 9.1. On the receive side, the reassembly automaton waits for the arrival of the first (or only) fragment of a Modbus PDU, discarding any stray fragments. After the first fragment has been received correctly, the total PDU length, and hence, the number of fragments needed to transfer it, is known. The protocol identifier is also checked to confirm that the PDU being received is indeed a Modbus PDU.

It is expected that the fragments will be received in the right sequence because CAN does not reorder frames sent sequentially by a single node. The reception of any out-of-order fragments or fragments belonging to a different PDU (according to their CAN identifier, fragment number and T bit) leads the receiver to discard the whole message. Duplicated fragments are silently discarded, but reassembly continues. By analogy with Modbus RTU and according to the general guidelines discussed in Section 9.2.1, neither flow control nor explicit fragment acknowledgments are foreseen.

It should also be noted that Modbus RTU does provide a frame spacing mechanism and associated timeout (called $t_{3.5}$ in [62]). This has no counterpart in Modbus CAN, in which no reassembly timeouts are implemented. In order to justify this difference, it is useful to recall what the goal of the $t_{3.5}$ timeout in Modbus RTU is, and how Modbus CAN fulfills the same goal in a different way. The purpose of the timeout is to reliably identify the *end of a PDU* transmitted on the serial line [62]. In Modbus CAN, the CAN controller is able to detect the end of a fragment by itself.

Since the last fragment of a PDU can be uniquely identified, provided the first fragment was received correctly, the end-of-fragment indication provided by the CAN controller concerning the last fragment of a PDU can be used to indicate end-of-PDU, too. As a side effect, in MODBUS CAN the end-of-frame indication immediately follows the successful reception of the last fragment instead of being delayed, as it happens in MODBUS RTU, by $t_{3.5}$. This leads to a better utilization of the available bus bandwidth.

Moreover, the MODBUS CAN protocol transparently makes use of the error detection and recovery mechanisms embedded in the CAN controller hardware to further increase the probability of a successful transfer. In fact, CAN controllers autonomously retransmit a frame (that is, a MODBUS CAN fragment) multiple times upon bus errors. In absence of persistent bus errors, this reduces the probability of fragment loss, so that the PDU is still received successfully although fragment-level retransmissions occur. Only in extremely low-probability cases the CAN controller aborts the transmission of a fragment after a number of attempts. This leads receivers to discard the MODBUS PDU as a whole and triggers the application-level timeout as it happens on MODBUS RTU.

9.3 PROMELA Modeling Language

The MODBUS CAN protocol has been modeled using PROMELA, the input language of the SPIN model checker [35]. After verification, the same model has been used to refine and improve its implementation. For clarity, only a passing glance at the language syntax, which is similar to the C programming language, will be given here, while the model description written in PROMELA will be given in the next section. Refer, for instance, to [36] for a more thorough description of PROMELA and model checking in general.

Processes are the main component in a Promela model. They are declared by means of the `proctype` keyword and contain all the *executable* statements of the model. One or more instances of a process can be created either right in the declaration itself, or during execution. In the first case, all process instances are constrained to be identical, whereas in the second case each instance may have its own arguments. A special process, `init`, is instantiated automatically, too, if declared.

Several commonly-used, integer data types are available for variables. For example, the `short` keyword corresponds to an integer data type with range $-32768, \ldots, 32767$. Boolean and byte-sized data types are available as well, by means of the `bool` and `byte` keywords, respectively. A variable can be either *local* to a process or *global*, and thus, accessible from all processes, depending on where it is declared.

The majority of C operators are supported in a PROMELA expression, with the restriction that PROMELA expressions must be *side-effect free*. The underlying reason is that a key concept in PROMELA is the *conditional executability* of statements, a feature useful to model a passive wait for an event in a simple way. A PROMELA process blocks when it encounters a statement that is not executable at the moment. In most cases, statement executability is assessed by evaluating one or more expressions and checking whether the result is true or false. It is therefore especially important to be able to repeatedly evaluate an expression during verification—to check whether a process can proceed or not—without perturbing the system state in any way.

For these reasons, PROMELA *assignment statements*, in the form:

```
1 var = expr
```

where `var` is a variable and `expr` is an expression, are not themselves expressions and are always executable. There are five additional *control statements* in the language: sequence, selection (`if` keyword), repetition (`do` keyword), jump (`goto` keyword), and the `unless` clause. Of these, only the first four kind of statements are actually used in the models discussed in this chapter. See [36, 3] for a description of the last one.

A sequence of statements, to be executed sequentially, can be built by putting several statements one after another, separated by a semicolon. The only difference with respect to the C language is that, in PROMELA, the semicolon is a statement *separator* rather than a *terminator*.

A selection statement is in the form:

```
1 if
2 :: guard_1 -> seq_1
3    ...
4 :: guard_n -> seq_n
5 fi
```

where `guard_1`, ..., `guard_n` are Boolean expressions, called *guards*, and `seq_1`, ..., `seq_n` are sequences of statements. The selection statement is executable if at least one of its guards is true and, in this case, execution proceeds with the sequence associated with one of the true guards.

If more than one guard is true, a *nondeterministic choice* exists among them: informally speaking, execution can proceed in more than one way and *all* possible execution paths will be considered during verification. This is especially useful, for instance, to model in a simple way the reaction of a process to concurrent external events, which can occur in a nondeterministic order. The special guard `else` is true if and only if none of the other guards in the same statement is true.

The repetition statement has the same syntactic structure as the selection statement:

```
1 do
2 :: guard_1 -> seq_1
3    ...
4 :: guard_n -> seq_n
5 od
```

but, after the execution of the statements associated with a true guard, the control flow goes back to the beginning of the `do`. The special statement `break` terminates the loop, that is, jumps to the statement that follows the `od` keyword. The jump statement is quite useful when the system can be represented as a Finite State Machine (FSM).

In PROMELA, elementary statements—for instance, an assignment—are executed *atomically*, regardless of their complexity. In other words, during verification any elementary statement is always evaluated as an indivisible unit and the system state does not change during the evaluation. On the contrary, since PROMELA's location counters work at the elementary statement level, compound statements—for instance, a sequence—are *not* atomic.

Two PROMELA keywords, `atomic` and `d_step`, can be used to define longer atomic sequences. The execution of an `atomic` sequence (see Figure 9.3, lines 6–25 for an example) starts if and only if its first statement, i.e. its guard, is executable. Then, it is executed atomically until it ends,

or a blocking statement is encountered. In the second case, atomicity is broken and the execution will resume if/when that statement becomes executable. A `d_step` sequence leads to a more efficient verification, at the expense of some additional semantics restrictions.

A channel, declared by means of the `chan` keyword, can be used to pass messages between PROMELA processes. Somewhat contrary to their name, channels are not point-to-point links, and any process can freely send/receive messages to/from any channel it has access to. For example, a statement like:

```
1 chan ch = [cap] of { type_1, ..., type_n }
```

declares a *buffered* channel named *ch*, holding up to *cap* messages. Each message is a data structure defined by the given sequence of types *type_1*, ..., *type_n*. A *cap* of zero denotes a *rendezvous* channel.

Messages can be sent/received to/from channels by means of the `!` and `?` constructs. For instance, the statements:

```
1 ch ! exp_1, ..., exp_n
2 ch ? var_1, ..., var_n
```

send a message built from the given expressions *exp_1*, ..., *exp_n* to channel *ch*, and receive a message from *ch*, respectively. In the second case, the message fields are stored in the sequence of variables *var_1*, ..., *var_n*. The special variable _ (underscore) denotes that the corresponding field must be discarded.

Compound data types can be defined by using the `typedef` keyword, followed by the name of the new data type and by a list of components. For instance:

```
1 typedef message {
2   <type> var_1;
3   <type> var_2
4 }
```

declares a compound data type called `message` with two fields: `var_1`, and `var_2`. Compound data types are especially useful, for instance, to represent the structure of a message. Individual fields within a compound data type can be referenced by the well-known C-language "dot notation". For instance, if `x` is a variable of type `message`, `x.var_1` refers to field `var_1` of `x`.

9.4 Formal Protocol Model

In this section, model corresponding to the Modbus CAN protocol described with PROMELA will be introduced. The main data types defined in the model, shown in Figure 9.2, represent a MODBUS CAN fragment (`fragment` data type) and its payload (`cookie` data type). The `fragment` data type contains the control byte fields introduced in Section 9.2.3, namely, the toggle bit `tog` and fragment number `f_n`. The total length field `t_l` is also present, whereas the protocol identifier (of secondary importance) has been omitted to simplify the verification.

The fragment data are the only aspect in which the model departs significantly from the real protocol implementation. In fact, in the real implementation, they consist of an uninterpreted sequence of bytes containing (part of) the MODBUS PDU being transferred. In the protocol model,

```
1 typedef cookie {
2   byte msg_id;
3   byte msg_f;
4   byte msg_t;
5 };
6
7 typedef fragment {
8   bit  tog;     /* Toggle bit */
9   byte f_n;     /* Fragment number */
10  byte t_l;     /* Total length (in 1st fragment) */
11  cookie data; /* Fragment data */
12 };
```

Figure 9.2. Main data types used in the model.

they have been replaced by a `cookie`, whose contents allow the receiving side to verify that all the fragments composing a message have been reassembled correctly by the adaptation layer protocol, in a way that will be better discussed in Section 9.4.3. The cookie contains a unique message identifier generated at the transmitting side, `msg_id`, the true position of the fragment within the message, `msg_f`, and the true total number of fragment of the message, `msg_t`.

9.4.1 CAN Bus and Error Model

The PROMELA model of the CAN bus, shown in Figure 9.3 (next page), mainly consists of the `broadcast` inline function. The CAN bus is modeled as an array of rendezvous channels `can[]`, one channel for each node. Then, the `broadcast` function broadcasts fragment `frag`, coming from node `n`, to all the other nodes. The `MAX_AGENT` macro represents the total number of nodes, or agents, in the system. The CAN arbitration mechanism is not modeled because, as explained in Section 9.2.1, simultaneous transmission attempts by different CAN nodes cannot occur.

The occurrence of global errors (on the transmitter side and affecting all receivers), or local errors (at a specific receiver) is modeled as a nondeterministic choice between normal behavior (Figure 9.3, line 14) and message drop (lines 23–24 and 15–16, respectively). The macros `MAX_N_TXLOST` and `MAX_N_RXLOST` set an upper limit on the number of global and local errors to be considered in the verification.

For simplicity, the model does not consider fragment duplication, which may occur as a consequence of very low-probability bus error conditions during the EOF sequence [85] because appropriate methods to address this peculiar kind of error have already been proposed in literature. A simpler alternative may consist of working around the issue by disabling automatic retransmission completely, at the CAN controller level. As a further, conservative hypothesis useful to simplify the model without hindering verification results, it is assumed that any CAN bus error goes undetected by the transmitter.

```
1  chan can[MAX_AGENT] = [0] of { fragment };
2  byte n_rxlost = 0;
3  byte n_txlost = 0;
4
5  inline broadcast(n, frag) {
6   atomic {
7    if
8    :: true -> {
9        i = 0;
10       do
11       :: (i == n) -> i++
12       :: (i < MAX_AGENT && i != n) -> {
13           if
14           :: true -> can[i] ! frag;
15           :: (n_rxlost < MAX_N_RXLOST) ->
16               n_rxlost++
17           fi;
18           i++
19          }
20       :: else -> break
21       od
22      }
23   :: (n_txlost < MAX_N_TXLOST) ->
24       n_txlost++
25   fi
26  }
27 }
```

Figure 9.3. PROMELA model of the CAN bus.

9.4.2 Transmitter

A transmitting node is modeled by means of the simple finite-state machine (FSM) shown in Figure 9.4. The input argument `id` contains the unique node identifier. Within an infinite loop (lines 10–44) the FSM first selects the length of the message to be sent, `t_l`. The choice is made nondeterministically, between a preset minimum and maximum number of fragments[3], by means of the function `generate_t_l` (line 11). Then, a unique message identifier `msg_id` is generated, to be used in the verification (line 12). Last, the toggle bit to be used to transmit the message is flipped (line 14).

The inner loop at lines 17–42 takes care of sending the message fragments as specified by the adaptation layer protocol, starting from the first one (lines 20–27) and continuing with the next ones (lines 32–38). The variable `f_n` keeps track of the fragment being sent. In both cases, the

[3]This is a small difference between the model and the real implementation of the protocol. The length is represented by the total number of bytes in a Modbus PDU. However, in the sense of verification correctness, they make no difference. This is because all lengths that correspond to the same number of fragments are equivalent from the protocol point of view. This kind of simplification is commonly done to speed up verification.

```
1  proctype tx_agent(byte id)
2  {
3    byte msg_id = 0; /* Current message identifier */
4    bit tog = 0;     /* Toggle bit of current message */
5    byte t_l;        /* Length of current message */
6    byte f_n;        /* Current fragment number */
7    fragment f;      /* Current fragment */
8    byte i;          /* For broadcast() */
9
10 tx_again:
11   generate_t_l(t_l);
12   generate_msg_id(msg_id);
13
14   tog = !tog;
15
16   f_n=0;
17   do
18   :: (f_n == 0) ->
19      {
20        /* Send first fragment (fragment 0) */
21        f.tog = tog;
22        f.f_n = f_n;
23        f.t_l = t_l;
24
25        generate_cookie(msg_id, f_n, t_l, f.data);
26        broadcast(id, f);
27        f_n++
28      }
29
30   :: (f_n != 0 && f_n < t_l) ->
31      {
32        /* Send fragment f_n > 0 */
33        f.tog = tog;
34        f.f_n = f_n;
35
36        generate_cookie(msg_id, f_n, t_l, f.data);
37        broadcast(id, f);
38        f_n++
39      }
40
41   :: else -> break
42   od;
43
44   goto tx_again
45 }
```

Figure 9.4. Model of the transmitting node.

control byte and header information pertaining to the fragment are filled in (lines 21–23 and 33–34). Then, an appropriate cookie is stored into the fragment payload for later verification (lines 25 and 36), by means of the function `generate_cookie`, and the fragment is broadcast on the CAN bus (lines 26 and 37).

9.4.3 Receiver

The receiving nodes FSM, listed in Figure 9.5, has two states. In the `S_IDLE` state (that is, the initial state) the FSM receives CAN frames (line 14) until it finds the first fragment of a message, discarding all the other fragments in the meantime (line 17). When the first fragment is found, it initializes several state variables containing the expected value of the toggle bit for all fragments of the message `tog`, the total length of the message `t_l`, and the expected fragment number of the next fragment `f_n` (lines 20–22). If the message is composed of only one fragment, the FSM stays in the `S_IDLE` state because the message is already complete, else it goes into the reassembly state `S_REASS` (lines 26–29).

In the `S_REASS` state, fragments received from the CAN bus are checked to ensure they contain the expected toggle bit and fragment number. If the check fails, the FSM drops the fragment and returns to the `S_IDLE` state (lines 42–46). Else, the FSM updates its state information to keep track that one more fragment has been successfully received (line 49). If the fragment just received is the last one, the message is complete and the FSM goes back to the `S_IDLE` state (lines 53–57), else it keeps reassembling fragments (line 59). In order to reduce model size and verification time, the FSM does not check the correctness of the CAN identifier received with every fragment. This conservative assumption does not affect the soundness of the verification results, because in this way the error checks performed by the model are weaker than the ones performed by the implementation.

Several hooks have been inserted in the FSM for verification purposes. Namely, the function `check_bom` is invoked whenever the FSM detects the beginning of a new message, the function `check_cookie` is invoked on all message fragments considered valid by the FSM, and the function `check_eom` is called when the FSM detects the end of a message. These three functions, not shown here for clarity, contain assertions on the fragment payload contents to verify message integrity. They make use of several verification-related state variables, also not shown. Last, the variable `drop` counts the number of fragments dropped by the FSM due to errors.

9.5 Verification Results

First of all, the protocol model described in Section 9.4 was verified in absence of CAN bus errors (that is, with `MAX_N_TXLOST` and `MAX_N_RXLOST` set to zero). The goal of this verification round was to confirm that the protocol is able to fragment and reassemble Modbus PDUs correctly (no assertion violations in the receiver's verification code) and is deadlock-free (no invalid end states detected by Spin). The same properties were also checked and confirmed to be true in all the subsequent verification rounds, discussed in the next sections, which focused on error tolerance and protocol optimization.

```
1  proctype rx_agent(byte id)
2  {
3    byte s = S_IDLE;
4    bit tog;          /* Toggle bit of current message */
5    byte t_l;         /* Total length of current message */
6    byte f_n;         /* Current fragment number */
7    byte drop = 0;
8    fragment f;
9
10   do
11   :: (s == S_IDLE) ->
12      {
13        if
14        :: can[id] ? f ->
15           {
16             if
17             :: (f.f_n != 0) -> drop++;
18             :: else ->
19                {
20                  tog = f.tog;
21                  t_l = f.t_l;
22                  f_n = 1;
23                  check_bom();
24                  check_cookie(f.data);
25
26                  if
27                  :: (f_n == t_l) -> check_eom();
28                  :: else -> s = S_REASS
29                  fi
30                }
31             fi
32           }
33        fi
34     }
35
36   :: (s == S_REASS) ->
37      {
38        if
39        :: can[id] ? f ->
40           {
41             if
42             :: (f.tog != tog || f.f_n != f_n) ->
43                {
44                  drop++;
45                  s = S_IDLE
46                }
47             :: else ->
48                {
49                  f_n++;
50                  check_cookie(f.data);
51
52                  if
53                  :: (f_n == t_l) ->
54                     {
55                       check_eom();
56                       s = S_IDLE
57                     }
58
59                  :: else -> skip
60                  fi
61                }
62             fi
63           }
64        fi
65     }
66   od
67 }
```

Figure 9.5. Model of the receiving nodes.

```
1  :: ((f.tog != tog && f.f_n != 0)
2      || (f.tog == tog && f.f_n != f_n)) ->
3     {
4       drop++;
5       s = S_IDLE
6     }
7  :: (f.f_n == 0 && f.tog != tog) ->
8     {
9       tog = f.tog;
10      t_l = f.t_l;
11      f_n = 1;
12      check_bom();
13      check_cookie(f.data);
14
15      if
16      :: (f_n == t_l) ->
17         {
18           check_eom();
19           s = S_IDLE
20         }
21      :: else -> skip
22      fi
23    }
```

Figure 9.6. Enhanced receiver logic.

9.5.1 Error Tolerance and Role of Toggle Bit

The second verification round focused on error tolerance. Namely, the verification was performed considering a single transmitter and receiver, and an increasing number of CAN bus errors local to the receiver, set by means of the `MAX_N_RXLOST` variable, while the `MAX_N_TXLOST` variable was kept at zero. Moreover, the role of the toggle bit in error tolerance was evaluated by selectively enabling and disabling it.

The verification results show that, when the toggle bit is enabled, the protocol is robust against double errors, by that we mean two fragments are lost. At the same time, they also show that the toggle bit plays a crucial role in this robustness. In fact, Spin was able to disprove that the protocol with the toggle bit disabled can tolerate double errors in about 11 s of execution time on a low-end PC. In any case, the protocol cannot tolerate triple errors, even with the toggle bit enabled. In the last case, the verification still took less than one minute.

9.5.2 Reduction of Dropped Fragments

According to the model shown in Figure 9.5 (lines 42–46), any out-of-sequence fragment received while the automaton is in the `S_REASS` state is dropped and resets the automaton to the `S_IDLE` state, in which it waits for the first fragment of a new message. This behavior is acceptable from the point of view of protocol correctness and robustness to errors, as shown in Section 9.5.1, but

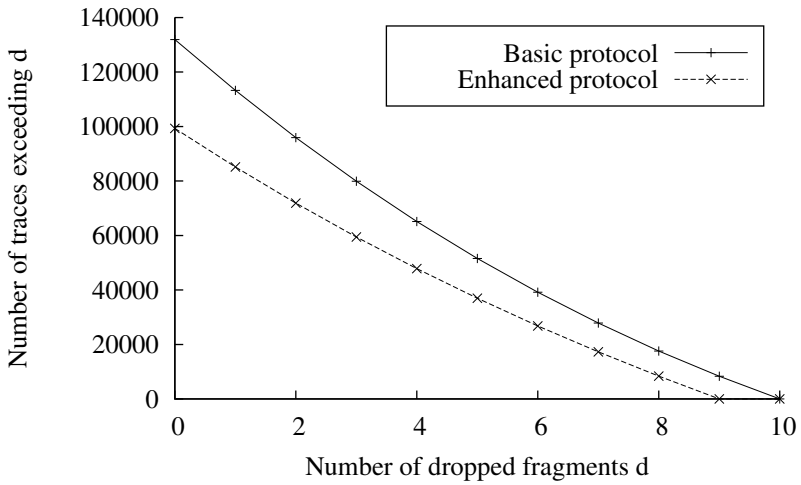

Figure 9.7. Fragment drop likelihood.

may lead—by intuition—to an unnecessarily large number of dropped fragments. This happens, for instance, when the out-of-sequence fragment being dropped is actually the first fragment of a new message. This causes the new message to be dropped before normal reception resumes, assuming no more errors occur.

In order to improve protocol performance with respect to this aspect, an enhanced receive automaton can be designed, in which an out-of-sequence fragment is double-checked to assess whether or not it is the first fragment of a new message. If this is the case, the fragment is not discarded and leads the automaton to start receiving the new message normally. The model of the enhanced transitions is shown in Figure 9.6, meant to replace lines 42–46 of Figure 9.5.

The performance improvement has been evaluated by means of formal verification, by instructing Spin to count in how many traces the number of dropped fragments—represented by the variable `drop` in the model—exceeded a given threshold d, as a consequence of one bus error local to the receiver. The model was configured to consider variable-length messages between 1 and 10 fragments.

The verification results, albeit not directly related to the absolute probability of dropping more than d fragments—because the traces considered by Spin during space exploration do not necessarily have the same probability of occurrence—are indeed a reliable indication on how two different versions of the protocol compare to each other.

The data are plotted in Figure 9.7 and show a significant advantage of the enhanced protocol with respect to the basic one, both with respect to the worst-case number of fragments dropped as a consequence of a single bus error (9 instead of 10) and, most importantly, to the likelihood of dropping a given number of fragments, which is significantly lower.

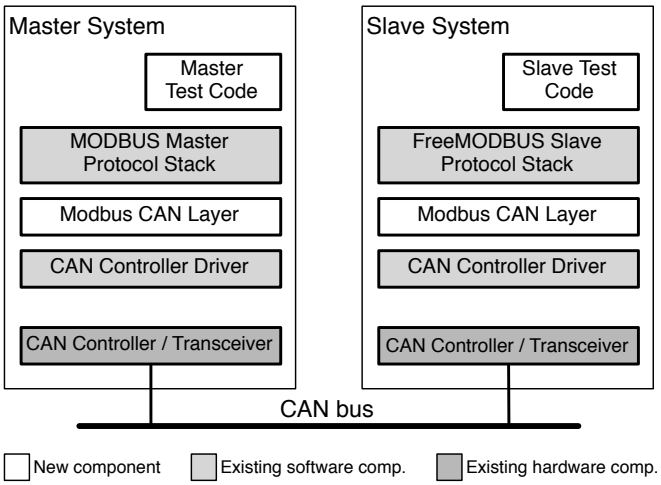

Figure 9.8. Experimental testbed.

9.6 Implementation and Performance Results

In order to evaluate the feasibility, complexity, and performance of Modbus CAN, the experimental testbed shown in Figure 9.8 was set up. It consists of two Modbus systems, a master (on the left) and a slave (on the right), connected through a CAN bus. The slave system is built around the open-source FreeMODBUS slave protocol stack [97], in which the Modbus CAN adaptation layer has been inserted. The master protocol stack [24] is based on the commercial counterpart of FreeMODBUS, extended in the same way.

On both sides, appropriate test code has been developed to implement and measure the round-trip time (RTT) of two typical Modbus requests, namely, *write multiple registers (wmr)* and *read holding registers (rhr)*. Register is the basic unit of data transfer in Modbus and it is fixed to 16 bits to make Modbus compatible across architecture for historical reasons. The first one makes use of a variable-length request PDU (of length $r_{\text{wmr}}(m) = 6 + 2m\,\text{B}$, where m is the number of registers) with a fixed-length reply ($\hat{r}_{\text{wmr}} = 5\,\text{B}$). The second one uses a fixed-length request ($r_{\text{rhr}} = 5\,\text{B}$) and a variable-length reply ($\hat{r}_{\text{rhr}}(m) = 2 + 2m\,\text{B}$). Considering the maximum allowable size of a Modbus PDU, it must be $m \leq 123$ for write multiple registers and $m \leq 125$ for read holding registers.

In both cases, their use makes the dependency between request/reply length and RTT easy to evaluate and, at the same time, highlights any asymmetry in the behavior of the master and slave protocol stacks when dealing with long PDUs. Moreover, the slave provides a dummy implementation of the requests to filter out any slave-dependent processing time (that varies according to the nature and purpose of the slave) from the RTT measurement proper.

The measurement itself has been performed by means of a hardware timer with a resolution of $1\,\mu$s. Data gathering took place on 10^4 samples, using a variable bin size, either $10\,\mu$s or $100\,\mu$s, to make results accurate up two significant digits regardless of their absolute value. In order to obtain a predictable amount of bit stuffing on the CAN bus, which contributes to both the RTT value and jitter, all experiments have been performed transferring randomly-generated,

uniformly-distributed register contents. Both systems make use of a typical low-cost embedded microcontroller, the NXP LPC2468 [70], based on a 72 MHz ARM7 CPU. They are both managed by the FreeRTOS real-time operating system [2].

9.6.1 Modbus CAN Performance Results

As shown in Figures 9.9 and 9.10 (both on the next page), the RTT of Modbus CAN communication exhibits a linear dependency on the number of registers being written or read, at both 500 kbps and 1 Mbps bit rate, with a rate-dependent slope. This is expected because, due to the relatively low speed of CAN, *message transfer time* should dominate on all other sources of delay. Denoting with $\mathrm{RTT}_{\mathrm{wmr}}(m)$ the RTT of write multiple registers, we can write

$$\overline{\mathrm{RTT}}_{\mathrm{wmr}}(m) = t_{\mathrm{wmr}}(m) + u_{\mathrm{wmr}}(m) \ , \tag{9.1}$$

where $t_{\mathrm{wmr}}(m)$ is the average amount of time spent transferring data on the CAN bus, considering bit stuffing, and $u_{\mathrm{wmr}}(m)$ is the average, total software processing time. The RTT measurement provides an overall evaluation of communication performance. Since, as shown in the following, we could derive an approximate analytic expression of $t_{\mathrm{wmr}}(m)$, it is also possible to accurately infer $u_{\mathrm{wmr}}(m)$ without measuring it directly.

Given the Modbus CAN protocol definition given in Section 9.2, the number of fragments $g(x)$ needed to transfer a Modbus PDU of length x, assuming no errors occur, is

$$g(x) = \begin{cases} 1 & \text{if } x \leq 5 \\ 1 + \lceil (x-5)/7 \rceil & \text{if } x > 5 \end{cases} \ . \tag{9.2}$$

Of these, $g(x) - 1$ are full-length CAN frames containing 8 B of data. The data length $l(x)$ of the last fragment, expressed in bytes, is

$$l(x) = \begin{cases} x + 3 & \text{if } x \leq 5 \\ x - (5 + 7(g(x) - 2)) + 1 & \text{if } x > 5 \end{cases} \ . \tag{9.3}$$

It should be noted that the last fragment includes not only the fragmented Modbus PDU but also the control byte, maybe even the Modbus CAN header if the last fragment is also the only fragment, as shown in Figure 9.1.

The average length, in bits, of a Modbus CAN frame with a y B data field can be linearly approximated from the results presented in Chapter 4 related to random payloads as

$$C(y, b) \simeq 45.6 + 8.226\, y + b \ . \tag{9.4}$$

The above expression does not include the CAN interframe spacing (IFS) because, on the systems being considered in this chapter, the measured inter-fragment software processing time is no lower than 15 μs, and hence, it largely exceeds the IFS at the bit rates used in the experiments. b is a compensation factor to account for the fact that the CAN identifiers used in Modbus CAN request and reply fragments are not the same as in Chapter 4, and hence, they contribute in different (but fixed and predictable) ways to the bit stuffing amount. By inspecting the identifiers, it has been determined that $b_{\mathrm{req}} = 1$ for requests and $b_{\mathrm{rep}} = 2$ for replies. The main source of uncertainty in

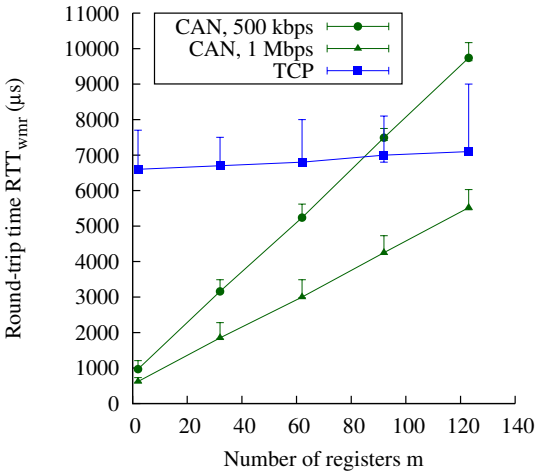

Figure 9.9. RTT of write multiple registers.

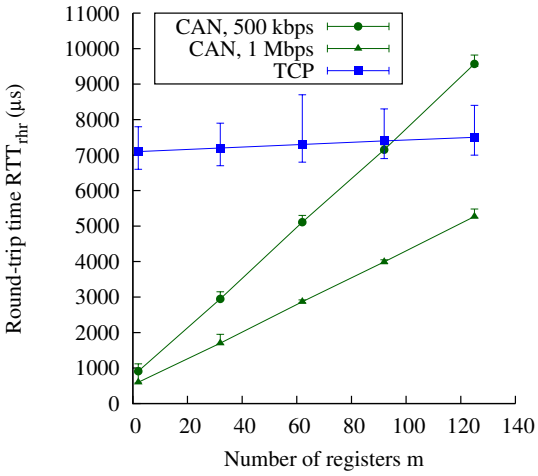

Figure 9.10. RTT of read holding registers.

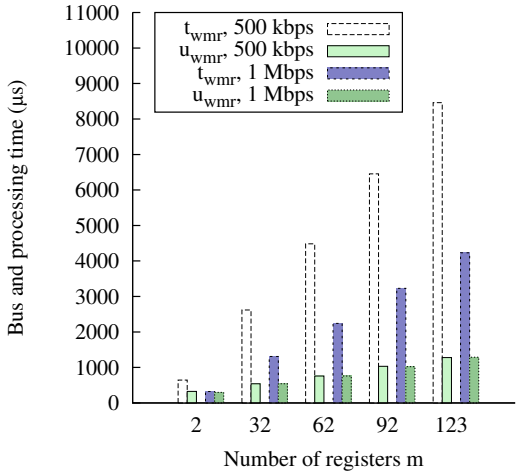

Figure 9.11. Breakdown of $\overline{\text{RTT}}_{\text{wmr}}$.

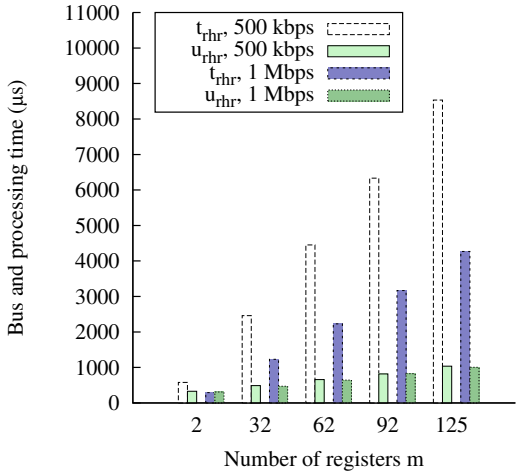

Figure 9.12. Breakdown of $\overline{\text{RTT}}_{\text{rhr}}$.

(9.4) comes from the fact that the fragments payload is not completely random (due to the control byte and header).

Combining (9.2) and (9.3) with (9.4), the average number of CAN bus bits needed to transfer a request or reply PDU of length x—assuming zero blocking time on the CAN bus due to the single-transmitter scenario enforced by the Modbus CAN protocol—is:

$$\begin{aligned} C_{\text{req}}(x) &= C(8, b_{\text{req}})(g(x) - 1) + C(l(x), b_{\text{req}}) \ , \\ C_{\text{rep}}(x) &= C(8, b_{\text{rep}})(g(x) - 1) + C(l(x), b_{\text{rep}}) \ . \end{aligned} \tag{9.5}$$

	J_{wmr} (μs)		J_{rhr} (μs)	
m	**500 kbps**	**1 Mbps**	**500 kbps**	**1 Mbps**
2	280	130	230	**290**
32	370	460	240	280
62	420	510	230	80
92	340	530	190	110
123	510	**570**	—	—
125	—	—	320	280

Table 9.1. Modbus CAN worst-case jitter.

The average number of CAN bus bits needed to transfer a write multiple registers request and reply PDU for m registers is then given by

$$C_{\text{wmr}}(m) = C_{\text{req}}(r_{\text{wmr}}(m)) + C_{\text{rep}}(\hat{r}_{\text{wmr}}) \ . \tag{9.6}$$

With a bit time of t_{bit}, the corresponding bus time is $t_{\text{wmr}}(m) = t_{\text{bit}}\ C_{\text{wmr}}(m)$. From (9.1) it is therefore possible to calculate $u_{\text{wmr}}(m)$ as

$$u_{\text{wmr}}(m) = \overline{\text{RTT}}_{\text{wmr}}(m) - t_{\text{wmr}}(m) \ . \tag{9.7}$$

The values of $t_{\text{wmr}}(m)$ and $u_{\text{wmr}}(m)$ are shown in Figure 9.11 and confirm the expectation. Namely, the message transfer time $t_{\text{wmr}}(m)$ always *dominates* on $u_{\text{wmr}}(m)$. Even at 1 Mbps, when the value of $t_{\text{wmr}}(m)$ is lower than at 500 kbps, $t_{\text{wmr}}(m) \simeq u_{\text{wmr}}(m)$ when $m = 2$ and it becomes $t_{\text{wmr}}(m) \simeq 3.2\ u_{\text{wmr}}(m)$ when $m = 123$.

Moreover, the software processing time $u_{\text{wmr}}(m)$ is bit-rate independent within an error band of $[-7, +33]\mu$s across the whole range of m, thus supporting the soundness of experimental results. The linear dependency on m can be justified by observing that software processing includes a pair of memory-to-memory PDU copy operations, while the error band can be attributed to the uncertainty in (9.4) and limited measurement accuracy.

The same considerations done in (9.1)–(9.7) can be extended to the read holding registers Modbus primitive, too, in order to calculate $t_{\text{rhr}}(m)$ and $u_{\text{rhr}}(m)$ and produce the results shown in Figure 9.12. The only difference worth remarking is that, instead of (9.6), it must be

$$C_{\text{rhr}}(m) = C_{\text{req}}(r_{\text{rhr}}) + C_{\text{rep}}(\hat{r}_{\text{rhr}}(m)) \ . \tag{9.8}$$

The worst-case Modbus CAN communication jitters (denoted as $J_{\text{wmr}}(m)$ and $J_{\text{rhr}}(m)$) are depicted as error bars in Figures 9.9 and 9.10. They are also summarized in Table 9.1 as a function of m and the CAN bit rate. In absolute terms, the worst-case jitter never exceeded 570 μs across the whole range of experiments and was fairly independent from m. We believe that it is mainly due to the Modbus protocol stacks. For this reason, the worst *relative* jitter was encountered when reading $m = 2$ holding registers at 1 Mbps. In that case, the average measured round-trip time was $\overline{\text{RTT}}_{\text{rhr}}(2) = 600\,\mu$s and $J_{\text{rhr}}(2) = 290\,\mu$s.

9.6.2 Comparison with Modbus TCP

In order to put the results discussed in Section 9.6.1 in a better perspective, the same experiments were repeated on a second testbed, very similar to the one shown in Figure 9.8. The two testbeds share the same hardware platform and most of their software components, including the two Modbus protocol stacks. The only difference is that one testbed uses Modbus CAN, whereas the other uses a third-party Modbus TCP implementation, layered on top of the open-source lwIP TCP/IP protocol stack [23]. In turn, lwIP makes use of the 100 Mbps Ethernet interface embedded in the LPC2468 chip by means of an optimized device driver thoroughly evaluated in the past [17]. For easier comparison with Modbus CAN, experimental results have been overlaid on Figures 9.9 and 9.10. Based on the results, the following considerations can be made:

- In Modbus TCP the RTT is dominated by software processing time, that is, $\mathrm{RTT}(m) \simeq u(m)$. In fact, the Ethernet data transfer time never exceeds 30 μs, even for the longest Modbus PDU, and it is two orders of magnitude smaller than the measured RTT. With respect to Modbus CAN, the difference can be attributed to the extra complexity of the communication software that includes a full-fledged TCP/IP protocol stack implemented as a separate task [23]. Moreover, the Modbus master protocol stack requires an additional *helper task* when it is layered on top of TCP.

- At a bit rate of 1 Mbps, Modbus CAN outperforms Modbus TCP in terms of RTT for all the considered values of m. At 500 kbps, the break-even point is between $m = 80$ and $m = 100$ for both primitives. From a practical standpoint, Modbus CAN still has an advantage, though, because most Modbus slaves export a much more limited number of registers, except for configuration, diagnostics and firmware updates, that is, activities that are usually performed *offline*, without hard real-time constraints.

- The Modbus TCP communication jitter exceeded 1 ms in most experiments. This behavior can be easily justified by recalling again the above-mentioned additional complexity of the Modbus TCP communication software. Besides extra delay, it also introduces response time variability through well-known mechanisms, such as, mutual task interference and bounded priority inversion at the synchronization primitive level.

It must also be mentioned that a comparison exclusively based on performance and determinism may not be completely fair. In this particular case, two more aspects should be taken into account, namely:

- First of all, given that the RTT of Modbus CAN and Modbus TCP is dominated by two distinct quantities (bus transfer versus software processing time), it can be argued that the break-even point depends on the CPU speed and is likely to change if much faster CPUs are adopted. However, microcontrollers with a clock speed on the order of 100 MHz, like the one considered here, are still worth considering because they are currently very widespread in industrial automation, due to their affordability and easiness of deployment.

- Secondly, the maximum physical extent of a Modbus TCP network exceeds what can be achieved with Modbus CAN. When using an ISO 11898-2 medium access unit [45], the

maximum length of a CAN bus cannot exceed 40 m at 1 Mbps. Longer lengths are possible only by reducing the bit rate. At 500 kbps, a bus length on the order of 100 m is typically feasible, also depending on the number of stations connected to the bus and their characteristics. The applicability of Modbus CAN is therefore limited to relatively small networks. On the other hand, the extra cost and complexity implied by a bigger Modbus TCP network (mainly due to network switches) should also be taken into account.

Summary

Nowadays, Modbus is still considered a viable option for inexpensively connecting devices in both factory and building automation systems, as long as timing requirements are not particularly tight. Currently, the only two choices available for the underlying network, specified by the Modbus standard, rely on serial lines and the TCP/IP protocol stack. In this chapter a third solution is introduced, known as Modbus CAN, that is stacked above CAN. With respect to Modbus RTU, Modbus CAN offers noticeably higher performance, yet preserving similarly low implementation costs and simplified wiring based on a bus topology.

It is proved that Modbus CAN is able to outperform Modbus TCP as well, at least for process data transfer. The advantage mostly depends on the efficient implementation of the fragmentation protocol, whose software processing times are about one order of magnitude lower than those required by lwIP on typical embedded architectures. No doubt the performance of Modbus TCP could be easily increased, by either raising the CPU speed or rewriting the TCP/IP protocol stack. However, in both cases this leads to higher implementation costs, which is probably not the best option given the target application fields typically envisaged for Modbus.

An aspect that deserves special attention is backward compatibility. Modbus CAN encodes every Modbus PDU separately, by means of a thin fragmentation layer located below the Modbus protocol entity. Hence, it is fully compatible with existing application programs. Concerning existing Modbus devices, gateways can be defined, which resemble those used to make Modbus TCP and Modbus RTU interoperable, that permit transparent interconnection to CAN-based subnetworks, too.

Summing up, Modbus CAN is a quite interesting alternative to serial lines, whether it is used alone or as a subnetwork of a larger Modbus TCP network. While retaining full compatibility with Modbus at both the application and device level, similar (or better) performance level as TCP/IP can be achieved at a fraction of the cost.

Conclusion

In the past three years, research activities have been carried out in the domain of embedded systems, for what concerns real-time communication. In particular, the Controller Area Network (CAN), which was originally designed for automotive applications and lately gained popularity in industrial automation and networked embedded systems, has been studied. On the one hand, effort has been spent to achieve *deterministic* communication on CAN so as to make it also suitable for critical systems with tight timing constraints. On the other hand, we extended the *flexibility* of CAN in both the real-time and non real-time domains. Namely, CAN has been extended to support the most widely spread Internet Protocol (IP), which permits the seamless integration of CAN networks used at the field level into company Intranet. Instead, in the real-time domain, a transport communication protocol has been designed and implemented. It makes it possible for CAN to carry real-time traffic according to the Modbus protocol, which is a de facto standard for connecting industrial electronic devices and building automation equipment.

More specifically, deterministic communication can not be achieved unless the duration of frame transmission is fixed and known. However, the *bit stuffing* mechanism, which is adopted at the physical layer of CAN for clock synchronization between transmitter and receivers, introduces variability in frame transmission times, which affects quality of control in a significant way. There are *two* main issues when dealing with bit stuffing jitter: first of all, the exact amount of jitter introduced by bit stuffing depends not only on the payload length, but also its *content*. Secondly, bit stuffing applies not only to the payload of a frame but also to the *CRC*, whose value is calculated by the hardware at run time.

Bit stuffing jitter has been *completely* prevented step by step following different philosophies. For what concerns the payload, suitable encoding schemes have been designed and implemented, including *fixed* length payload encoding, namely 8B9B, and *variable* length payload encoding, namely VHCC, which is able to further improve the encoding efficiency. The general principle of them is to encode the payload in such a way that no stuff bits will be added to the encoded payload, without affecting clock synchronization. Instead, for what concerns the CRC, a Zero Stuff bit CRC (ZSC) mechanism has been proposed. By just using 3 bits of the data field, it is able to tune the CRC to a value which does not require bit stuffing, without affecting its effectiveness.

A big advantage of all of them is that *no* modification to existing hardware is required. They are implemented as separate intermediate software layers, which can also be used together. What's more, their execution is *efficient*, just a couple of microseconds, and *deterministic* across different architectures. In addition, simulation results show that payload encoding schemes like 8B9B and VHCC, which are adopted to prevent bit stuffing jitter from the data field, also help lowering down the residual error probability and improve *data integrity* during communication. Last but not the

least, the ZSC mechanism led to an Italian patent application, while at the same time, a European extension is under preparation.

For what concerns flexible CAN communication, it has been extended to support both a general-purpose protocol, namely IP, and a special-purpose real-time protocol, namely Modbus. Correspondingly, their main design concerns are different from each other. When aiming at *integrating* CAN into Intranet, instead of the conventional way of doing which just connects different subsystems without having traffic cross the boundary, *coexistence* becomes a major issue. This has been solved by appropriate priority assignment, in which real-time messages always have higher priority than messages related to IP traffic. The research activity concerning Modbus has been carried out under an industrial research contract. A customized transport protocol stack for Modbus Protocol Data Units (PDUs) on a CAN segment has been designed and implemented in order to build a Modbus CAN network. The main concern of this design is to optimize the *real-time performance* by exploiting Modbus communication characteristics. Experimental results demonstrate that Modbus CAN exhibits comparable real-time performance with respect to Modbus TCP. This activity has been concluded by on-site deployment and acceptance tests.

Research work was also carried out in other research areas during the PhD, for instance, formal verification. Activities were conducted at both the communication and system level. At the communication level, the *correctness* of a distributed multiple master election protocol [8] has been formally verified with model checking. At the system level, the formal foundation of a verification tool for the automatic analysis of security policies was first laid out [15], and then used to build a model of a realistic industrial control system and perform the corresponding *security* verification [16]. They are not included in the dissertation for clarity and conciseness. The knowledge gained in this area also led to a book chapter "Model checking" [38] I co-authored which has been published in the "Digital Avionics Handbook", 3rd edition (Taylor and Francis), 2014.

Experience gained in the past about embedded systems also provided me the great opportunity to be involved as a coauthor in the preparation of the book "Embedded Software Development through Open-Source Software", ISBN 978-1-4665-9392-3 (Taylor and Francis), which is currently in progress and scheduled for publication in 2015. Overall, activities carried out in the past three years have led to the publication of a book, a book chapter, a patent, 5 journal papers and 11 conference papers.

After the PhD, the research work will be continued and emphasis will be given to real-time scheduling and communication in networked *multi-core* embedded systems.

Bibliography

[1] James Aweya. Technique for differential timing transfer over packet networks. *IEEE Transactions on Industrial Informatics*, 9(1):325–336, February 2013.

[2] Richard Barry. *Using the FreeRTOS Real Time Kernel – Standard Edition*. Lulu Press, Raleigh, North Carolina, 2nd edition, January 2011.

[3] Mordechai Ben-Ari. *Principles of the Spin Model Checker*. Springer-Verlag, London, 2008.

[4] Jan Beran, Petr Fiedler, and Frantisek Zezulka. Virtual automation networks. *IEEE Industrial Electronics Magazine*, 4(3):20–27, September 2010.

[5] Robert Braden, editor. *Requirements for Internet Hosts – Communication Layers, RFC 1122*. Internet Engineering Task Force, October 1989.

[6] Petr Cach and Petr Fiedler. *Internet Draft – IP over CAN*. Brno University of Technology, March 2001.

[7] CAN in Automation e. V. *CiA 301 V4.2.0 – CANopen application layer and communication profile*, February 2011.

[8] Gianluca Cena, Ivan Cibrario Bertolotti, and Tingting Hu. Formal verification of a distributed master election protocol. In *Proc. 9th IEEE Intl. Workshop on Factory Communication Systems (WFCS)*, pages 245–254, May 2012.

[9] Gianluca Cena, Ivan Cibrario Bertolotti, Tingting Hu, and Adriano Valenzano. Performance evaluation and improvement of the CPU–CAN controller interface for low-jitter communication. In *Proc. 17th IEEE Intl. Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, September 2012.

[10] Gianluca Cena, Ivan Cibrario Bertolotti, Tingting Hu, and Adriano Valenzano. Software-based assessment of the synchronization and error handling behavior of a real CAN controller. In *Proc. 18th IEEE Intl. Conference on Emerging Technologies and Factory Automation (ETFA)*, 2013.

[11] Gianluca Cena, Ivan Cibrario Bertolotti, and Adriano Valenzano. An efficient fixed-length encoding scheme for CAN. In *Proc. 9th IEEE Intl. Workshop on Factory Communication Systems (WFCS)*, pages 265–274, 2012.

[12] Gianluca Cena and Adriano Valenzano. Overclocking of Controller Area Networks. *Electronics Letters*, 35(22):1923–1925, October 1999.

[13] Steve Chamberlain, Roland Pesch, Red Hat Support, and Jeff Johnston. *The Red Hat Newlib C Library*, December 2010.

[14] Joachim Charzinski. Performance of the error detection mechanisms in CAN. In *Proc. 1st Intl. CAN Conference (iCC)*, pages 20–29, September 1994.

[15] Ivan Cibrario Bertolotti, Luca Durante, Tingting Hu, and Adriano Valenzano. A unified class model for checking security policies in ICT infrastructures. In *Proc. 1st IEEE AESS European Conference on Satellite Telecommunications (ESTEL)*, pages 1–6, October 2012.

[16] Ivan Cibrario Bertolotti, Luca Durante, Tingting Hu, and Adriano Valenzano. A model for the analysis of security policies in industrial networks. In *Proc. 1st Intl. Symposium for ICS & SCADA Cyber Security Research*, pages 66–77. BCS Learning and Development Ltd., September 2013.

[17] Ivan Cibrario Bertolotti and Tingting Hu. Real-time performance of an open-source protocol stack for low-cost, embedded systems. In *Proc. 16th IEEE Intl. Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, September 2011.

[18] Compaq Computer Corp., Hewlett-Packard Company, Intel Corp., Lucent Technologies Inc., Microsoft Corp., NEC Corp., Koninklijke Philips Electronics N.V. *Universal Serial Bus Specification*, April 2000. Revision 2.0.

[19] Robert I. Davis, Alan Burns, Reinder Bril, and Johan Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, April 2007.

[20] Robert I. Davis, Steffen Kollmann, Victor Pollex, and Frank Slomka. Schedulability analysis for Controller Area Network (CAN) with FIFO queues priority queues and gateways. *Real-Time Systems*, 49(1):73–116, January 2013.

[21] Zu De Zhou, Ricardo Valerdi, Shang-Ming Zhou, and Li Wang. Guest editorial: Special section on IoT. *IEEE Transactions on Industrial Informatics*, 10(2):1413–1416, May 2014.

[22] Michael Ditze, Reinhard Bernhardi, Guido Kämper, and Peter Altenbernd. Porting the Internet Protocol to the Controller Area Network. In *Proc. 2nd Intl. Workshop on Real-Time LANs in the Internet Age*, pages 1–4, July 2003.

[23] A. Dunkels. Full TCP/IP for 8-bit architectures. In *Proc. 1st Intl. Conference on Mobile Applications, Systems and Services (MOBISYS)*, May 2003.

[24] Embedded Solutions. Modbus master. Available online, at `http://www.embedded-solutions.at/`, September 2008.

[25] Esd electronic system design GmbH. *NTCAN Part 1: Structure, Function and C/C++ API Application Developers Manual*, August 2013.

[26] Esd electronic system design GmbH. *ELLSI Manual — EtherCAN Low Level Socket Interface*, January 2014.

[27] Joaquim Ferreira, Arnaldo Oliveira, Pedro Fonseca, and José Fonseca. An experiment to assess bit error rate in CAN. In *Proc. 3rd Intl. Workshop of Real-Time Networks (RTN)*, pages 15–18, June 2004.

[28] Piotr Gaj, Jürgen Jasperneite, and Max Felser. Computer communication within industrial distributed environment—a survey. *IEEE Transactions on Industrial Informatics*, 9(1):182–189, February 2013.

[29] David Gessner, Manuel Barranco, and Julián Proenza. Design and verification of a media redundancy management driver for a CAN star topology. *IEEE Transactions on Industrial Informatics*, 9(1):237–245, February 2013.

[30] Rachana A. Gupta and Mo-Yuen Chow. Networked control system: Overview and research trends. *IEEE Transactions on Industrial Electronics*, 57(7):2527–2535, July 2010.

[31] Florian Hartwich and Armin Bassemir. The configuration of the CAN bit timing. In *Proc. 6th Intl. CAN Conference (iCC)*, pages 1–10, November 1999.

[32] Peter Hellekalek. Good random number generators are (not so) easy to find. *Mathematics and Computers in Simulation*, 46(5–6):485–505, June 1998.

[33] Thomas A. Henzinger, Benjamin Horowitz, and Christoph M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, January 2003.

[34] HMS Industrial Networks AB. *User Manual — Anybus® X-gateway*, August 2014.

[35] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[36] Gerard J. Holzmann. *The Spin model checker: Primer and Reference Manual*. Pearson Education, Boston, MA, September 2003.

[37] Charles Hornig. *A Standard for the Transmission of IP Datagrams over Ethernet Networks, RFC 894*. Symbolics Cambridge Research Center, April 1984.

[38] Tingting Hu and Ivan Cibrario Bertolotti. *Digital Avionics Handbook*, chapter Model Checking. CRC Press, Taylor & Francis Group, 3rd edition, September 2014.

[39] David M. E. Ingram, Pascal Schaub, Richard R. Taylor, and Duncan A. Campbell. Performance analysis of IEC 61850 sampled value process bus networks. *IEEE Transactions on Industrial Informatics*, 9(3):1445–1454, August 2013.

[40] Institute of Electrical and Electronics Engineers. *IEEE Std 802.3™-2008, Standard for Information technology—Telecommunications and information exchange between systems—Local and metropolitan area networks—Specific requirements Part 3: Carrier sense multiple access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, December 2008.

[41] Institute of Electrical and Electronics Engineers. *IEEE Std 802.1AB™-2009, Standard for local and metropolitan area networks—Station and media access control connectivity discovery*, September 2009.

[42] International Electrotechnical Commission. *Low-voltage switchgear and controlgear – Controller-device interfaces (CDIs) – Part 3: DeviceNet*, 2.0 edition, January 2008. IEC 62026-3.

[43] International Organization for Standardization. *ISO 13239 – Information technology – Telecommunications and information exchange between systems – High-level data link control (HDLC) procedures*, July 2002.

[44] International Organization for Standardization. *ISO 11898-1 – Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling*, December 2003.

[45] International Organization for Standardization. *ISO 11898-2 – Road vehicles – Controller area network (CAN) – Part 2: High-speed medium access unit*, December 2003.

[46] International Organization for Standardization. *ISO 11898-4 – Road vehicles – Controller area network (CAN) – Part 4: Time-triggered communication*, August 2004.

[47] International Organization for Standardization and International Electrotechnical Commission. *ISO/IEC 13211-1 – Information technology – Programming languages – Prolog – Part 1: General core*, June 1995.

[48] International Organization for Standardization and International Electrotechnical Commission. *ISO/IEC 9899, Programming Languages — C*, 2nd edition, December 1999.

[49] Mathias Johanson, Lennart Karlsson, and Tore Risch. Relaying Controller Area Network frames over wireless internetworks for automotive testing applications. In *Proc. 4th Intl. Conference on Systems and Networks Communications (ICSNC)*, pages 1–5, September 2009.

[50] Donald E. Knuth. *The Art of Computer Programming, Volume II: Seminumerical Algorithms*. Addison-Wesley, 3rd edition, November 1997.

[51] G Leen and D Heffernan. TTCAN: a new time-triggered controller area network. *Microprocessors and Microsystems*, 26(2):77 – 94, March 2002.

[52] Xiaoting Li, Jean-Luc Scharbarg, and Christian Fraboul. Worst-case delay analysis on a real-time heterogeneous network. In *Proc. 7th IEEE Intl. Symposium on Industrial Embedded Systems (SIES)*, pages 11–20, June 2012.

[53] Per Lindgren, Simon Aittamaa, and Johan Eriksson. IP over CAN, transparent vehicular to infrastructure access. In *Proc. 5th IEEE Intl. Consumer Communications and Networking Conference (CCNC)*, pages 758–759, January 2008.

[54] Camilo Lozoya, Manel Velasco, and Pau Martí. The one-shot task model for robust real-time embedded control systems. *IEEE Transactions on Industrial Informatics*, 4(3):164–174, August 2008.

[55] Pau Martí, Antonio Camacho, Manel Velasco, and Mohamed El Mongi Ben Gaid. Runtime allocation of optional control jobs to a set of CAN-based networked control systems. *IEEE Transactions on Industrial Informatics*, 6(4):503–520, November 2010.

[56] Pau Martí, José Yépez, Manel Velasco, Ricard Villà, and Josep M. Fuertes. Managing quality-of-control in network-based control systems by controller and message scheduling co-design. *IEEE Transactions on Industrial Electronics*, 51(6):1159–1167, December 2004.

[57] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, January 1998.

[58] Jim McConahey. Using Modbus for process control and automation, part 1. *Applied Automation*, pages A12–A14, December 2011.

[59] Jim McConahey. Using Modbus for process control and automation, part 2. *Applied Automation*, pages A12–A14, February 2012.

[60] Modbus-IDA. *MODBUS Application Protocol Specification V1.1b*. Modbus Organization, Inc., 2006. Available online, at http://www.modbus-ida.org/.

[61] Modbus-IDA. *MODBUS Messaging on TCP/IP Implementation Guide V1.0b*. Modbus Organization, Inc., 2006. Available online, at http://www.modbus-ida.org/.

[62] Modbus-IDA. *MODBUS over Serial Line Specification and Implementation Guide V1.02*. Modbus Organization, Inc., 2006. Available online, at http://www.modbus-ida.org/.

[63] Abdelaziz A. Nacer, Katia Jaffres-Runser, Jean-Luc Scharbarg, and Christian Fraboul. Strategies for the interconnection of CAN buses through an Ethernet switch. In *Proc. 8th IEEE Intl. Symposium on Industrial Embedded Systems (SIES)*, pages 77–80, June 2013.

[64] Mouaaz Nahas. Applying eight-to-eleven modulation to reduce message-length variations in distributed embedded systems using the Controller Area Network (CAN) protocol. *Canadian Journal on Electrical and Electronics Engineering*, 2(7):282–293, July 2011.

[65] Mouaaz Nahas and Michael J. Pont. Using XOR operations to reduce variations in the transmission time of CAN messages: A pilot study. In *Proc. Second UK Embedded Forum*, pages 4–17, October 2005.

[66] Mouaaz Nahas, Michael J. Pont, and Michael Short. Reducing message-length variations in resource-constrained embedded systems implemented using the CAN protocol. *Journal of Systems Architecture*, 55(5–6):344–354, May 2009.

[67] Mouaaz Nahas, Michael Short, and Michael J. Pont. The impact of bit stuffing on the real-time performance of a distributed control system. In *Proc. 10th Intl. CAN Conference (iCC)*, pages 10.1–10.7, March 2005.

[68] Thomas Nolte, Hans Hansson, and Christer Norström. Minimizing CAN response-time jitter by message manipulation. In *Proc. IEEE Intl. Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 197–206, September 2002.

[69] Thomas Nolte, Hans Hansson, Christer Norström, and Sasikumar Punnekkat. Using bit-stuffing distributions in CAN analysis. In *Proc. IEEE/IEE Intl. Real-Time Embedded Systems Workshop (RTES)*, December 2001.

[70] NXP Semiconductors N.V. *LPC2468 Product data sheet, rev. 4*, October 2008.

[71] NXP Semiconductors N.V. *LPC24XX User manual, UM10237 rev. 2*, December 2008.

[72] NXP Semiconductors N.V. *LPC1769/68/67/66/65/64/63 Product data sheet, rev. 6*, 2010.

[73] NXP Semiconductors N.V. *LPC17XX User manual, UM10360 rev. 2*, August 2010.

[74] Myeongjin Oh, Young-Gab Kim, Seungpyo Hong, and Sung D. Cha. ASA: Agent-based secure ARP cache management. *IET Communications*, 6(7):685–693, May 2012.

[75] Michael Paulitsch, Jennifer Morris, Brendan Hall, Kevin Driscoll, Elizabeth Latronico, and Philip Koopman. Coverage and the use of cyclic redundancy codes in ultra-dependable systems. In *Proc. Intl. Conference on Dependable Systems and Networks (DSN)*, pages 346–355, June 2005.

[76] Roland H. Pesch, Jeffrey M. Osier, and Cygnus Support. *The GNU Binary Utilities*, March 2014.

[77] Michael J. Pont. *Patterns for Time-triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers*. Addison-Wesley, July 2001.

[78] Jon Postel, editor. *Internet Protocol — DARPA Internet Program Protocol Specification, RFC 791*. USC/Information Sciences Institute, September 1981.

[79] Tony F. Pulgar, Jean-Luc Scharbarg, Katia Jaffres-Runser, and Christian Fraboul. Extending CAN over the air: An interconnection study with IEEE802.11. In *Proc. 18th IEEE Intl. Conference on Emerging Technologies Factory Automation (ETFA)*, pages 1–8, September 2013.

[80] Mehrnoush Rahmani, Ktawut Tappayuthpijarn, Benjamin Krebs, Eckehard Steinbach, and Richard Bogenberger. Traffic shaping for resource-efficient in-vehicle communication. *IEEE Transactions on Industrial Informatics*, 5(4):414–428, November 2009.

[81] Yakov Rekhter and Tony Li, editors. *An Architecture for IP Address Allocation with CIDR, RFC 1518*. T.J. Watson Research Center, IBM Corp. and Cisco Systems, August 1993.

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

[83] Robert Bosch GmbH. *CAN with Flexible Data-Rate Specification Version 1.0*, April 2012.

[84] Guillermo Rodriguez-Navas, Sebastià Roca, and Julián Proenza. Orthogonal, fault-tolerant, and high-precision clock synchronization for the Controller Area Network. *IEEE Transactions on Industrial Informatics*, 4(2):92–101, May 2008.

[85] José Rufino, Paulo Veríssimo, Guilherme Arroz, Carlos Almeida, and Luís Rodrigues. Fault-tolerant broadcasts in CAN. In *Proc. 28th Intl. Symposium on Fault-Tolerant Computing*, pages 150–159, June 1998.

[86] Thilo Sauter and Maksim Lobashov. How to access factory floor information using internet technologies and gateways. *IEEE Transactions on Industrial Informatics*, 7(4):699–712, November 2011.

[87] Thilo Sauter, Stefan Soucek, Wolfgang Kastner, and Dietmar Dietrich. The evolution of factory and building automation. *IEEE Industrial Electronics Magazine*, 5(3):35–48, September 2011.

[88] Jean-Luc Scharbarg, Marc Boyer, and Christian Fraboul. CAN-Ethernet architectures for real-time applications. In *Proc. 10th IEEE Intl. Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 2, pages 245–252, September 2005.

[89] Frank Schiller and Tina Mattes. Residual error probability of embedded CRC by stochastic automata. In *Proc. Intl. Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, volume 6351 of *Lecture Notes in Computer Science*, pages 155–168. Springer Berlin Heidelberg, September 2010.

[90] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, November 2004.

[91] Imran Sheikh, Musharraf Hanif, and Michael Short. Improving information throughput and transmission predictability in Controller Area Networks. In *Proc. IEEE Intl. Symposium on Industrial Electronics (ISIE)*, pages 1736–1741, July 2010.

[92] Richard M. Stallman and the GCC Developer Community. *Using the GNU Compiler Collection*, October 2013.

[93] Telecommunications Industry Association. *Electrical Characteristics of Generators and Receivers for Use in Balanced Digital Multipoint Systems (ANSI/TIA/EIA-485-A-98) (R2003)*, March 2003.

[94] Eushiuan Tran. *Multi-Bit Error Vulnerabilities in the Controller Area Network Protocol*. PhD thesis, Carnegie Mellon University, May 1999.

[95] University of Amsterdam. *SWI-Prolog 6.0 Reference Manual*, March 2012.

[96] George Varghese and Anthony Lauck. Hashed and hierarchical timing wheels: Efficient data structures for implementing a timer facility. *IEEE/ACM Transactions on Networking*, 5(6):824–834, December 1997.

[97] Christian Walter. FreeMODBUS - a Modbus ASCII/RTU and TCP implementation. Available online, at `http://freemodbus.berlios.de/`, August 2007.

[98] Yuanqing Xia, Jingjing Yan, Peng Shi, and Mengyin Fu. Stability analysis of discrete-time systems with quantized feedback and measurements. *IEEE Transactions on Industrial Informatics*, 9(1):313–324, February 2013.

[99] Haibo Zeng, Marco Di Natale, Paolo Giusto, and Alberto Sangiovanni-Vincentelli. Using statistical methods to compute the probability distribution of message response time in Controller Area Network. *IEEE Transactions on Industrial Electronics*, 6(4):678–691, November 2010.

[100] Lixian Zhang, Huijun Gao, and Okyay Kaynak. Network-induced constraints in networked control systems—a survey. *IEEE Transactions on Industrial Informatics*, 9(1):403–416, February 2013.