CONTROLLER AREA NETWORK WITH OFFSET SCHEDULING: IMPROVED OFFSET ASSIGNMENT ALGORITHMS AND COMPUTATION OF RESPONSE TIME DISTRIBUTIONS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

AHMET BATUR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
ELECTRICAL AND ELECTRONICS ENGINEERING

FEBRUARY 2018

Approval of the thesis:

CONTROLLER AREA NETWORK WITH OFFSET SCHEDULING: IMPROVED OFFSET ASSIGNMENT ALGORITHMS AND COMPUTATION OF RESPONSE TIME DISTRIBUTIONS

submitted by **AHMET BATUR** in partial fulfillment of the requirements for the degree of **Master of Science in Electrical and Electronics Engineering Department, Middle East Technical University** by,

Prof. Dr. Gülbin Dural Ünver
Dean, Graduate School of **Natural and Applied Sciences** ________________

Prof. Dr. Tolga Çiloğlu
Head of Department, **Electrical and Electronics Engineering** ________________

Assoc. Prof. Dr. Klaus Werner Schmidt
Supervisor, **Electrical and Electronics Eng. Dept., METU** ________________

Prof. Dr. Şenan Ece Güran Schmidt
Co-supervisor, **Electrical and Electronics Eng. Dept., METU** ________________

Examining Committee Members:

Prof. Dr. Gözde Bozdağı Akar
Electrical and Electronics Engineering Department, METU ________________

Assoc. Prof. Dr. Klaus Werner Schmidt
Electrical and Electronics Engineering Department, METU ________________

Assoc. Prof. Dr. Cüneyt F. Bazlamaçcı
Electrical and Electronics Engineering Department, METU ________________

Assist. Prof. Dr. Ebru Aydın Göl
Computer Engineering Department, METU ________________

Assoc. Prof. Dr. Orhan Gazi
Elec. and Comm. Eng. Department, Çankaya University ________________

Date: **February 7, 2018**

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last Name:	AHMET BATUR
Signature :	

ABSTRACT

CONTROLLER AREA NETWORK WITH OFFSET SCHEDULING: IMPROVED OFFSET ASSIGNMENT ALGORITHMS AND COMPUTATION OF RESPONSE TIME DISTRIBUTIONS

BATUR, Ahmet
M.S., Department of Electrical and Electronics Engineering
Supervisor : Assoc. Prof. Dr. Klaus Werner Schmidt
Co-Supervisor : Prof. Dr. Şenan Ece Güran Schmidt

February 2018, 86 pages

The Controller Area Network (CAN) is the most widely-used in-vehicle communication bus in the automotive industry. CAN enables the exchange of data among different electronic control units (ECUs) of a vehicle via messages. The basic requirement for the design of CAN is to guarantee that the worst-case response time (WCRT) of each message is smaller than its specified deadline. Hereby, it is generally desired to achieve small WCRTs that leave sufficient slack to the message deadline. In addition, it has to be noted that it might be very unlikely that a message experiences the WCRT when being transmitted on CAN. That is, instead of only considering the message WCRT for the design of CAN, it is beneficial to determine the actual response-time distribution of each message, which indicates the probability of experiencing a certain response time.

In order to achieve small WCRTs, the idea of offset scheduling has been introduced. In this setting, messages on CAN are released with offsets in order to avoid message bursts that lead to undesirably large response times. In order to use offset scheduling efficiently, it is required to assign a suitable offset to each message. To this end, a load distribution (LD) algorithm is proposed in the existing literature. The first contribution of this thesis is the development of new algorithms for the offset assignment on CAN. Evaluating different example scenarios, the thesis shows that the proposed algorithms outperform the existing LD algorithm in most of the cases. As the second contribution, the thesis studies the computation of response time distributions. First, an algorithm for determining the exact response-time distribution of each message on CAN is proposed. Since this algorithm comes with a high computational complexity, it cannot be applied if there are too many messages on a CAN bus. Moreover, experimental results show that the response time distribution depends mostly on the initial phasing of the nodes. Therefore exact response time distribution as computed is not observed in the measurements. In response to this observation, the thesis proposes the computation of a local response time distribution and develops and implements a weak synchronization method which bounds the phase shift between the nodes. The resulting computed local response time distribution shows a very tight match with measured response time distributions.

Keywords: Controller Area Network, Offset Scheduling, Worst Case Response Time, Response Time Distribution, Weak Synchronization

ÖZ

CAN AĞLARI İÇİN OFSET ÇİZELGELEME: GELİŞTİRİLMİŞ OFSET ATAMA ALGORİTMALARI VE TEPKİ ZAMANI DAĞILIMI HESAPLAMALARI

BATUR, Ahmet
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü

Tez Yöneticisi	: Doç. Dr. Klaus Werner Schmidt
Ortak Tez Yöneticisi	: Prof. Dr. Şenan Ece Güran Schmidt

Şubat 2018 , 86 sayfa

Denetleyici Alan Ağı (CAN), otomotiv endüstrisinde en yaygın kullanılan araç-içi haberleşme veriyoludur. CAN, aracın farklı elektronik kontrol üniteleri (EKÜ) arasında mesajlar aracılığıyla veri transferi sağlar. CAN tasarımının ana ihtiyacı her mesajın tepki zamanının belirtilen son gönderim zamanından kısa olmasını garantilemektir. Genellikle, son gönderim zamanına yeterli serbestlik sağlayan kısa tepki zamanlarının elde edilmesi amaçlanmaktadır. Buna ek olarak, mesajın CAN üzerinden iletilirken en uzun tepki zamanını (worst-case response time-WCRT) deneyimlemesinin muhtemel olmayabileceğine dikkat edilmelidir. Yani, CAN tasarımı için mesajın WCRT değerini dikkate almak yerine her mesajın belirli bir tepki zamanını deneyimleme olasılığını gösteren gerçek tepki zamanı dağılımını belirlemek faydalı olacaktır.

Kısa WCRT değerleri elde etmek için ofset çizelgeleme yöntemi ortaya çıkarılmıştır. Bu yöntemle, istenmeyen uzun tepki zamanlarına yol açabilecek mesaj kümelenmesini önlemek için CAN mesajları ofsetlerle gönderilirler. Ofset çizelgeme yöntemini verimli bir şekilde kullanmak için her mesaja uygun bir ofset atanması gerekmektedir. Bu amaçla, mevcut literatürde buluşsal bir yük dağılımı (load distribution-LD) algoritması önerilmiştir. Bu tezin ilk katkısı CAN üzerinde ofset ataması için yeni algoritmaların geliştirilmesidir. Farklı senaryo örneklerini değerlendiren bu tez çalışmasında, bulguların çoğunda önerilen algoritmaların başarımlarının halihazırdaki LD algoritmasından daha iyi olduğu gösterilmektedir. İkinci katkı olarak tez, tepki zamanı dağılımlarının hesaplanmasını incelemektedir. İlk olarak, CAN üzerindeki her bir mesajın gerçek tepki zamanı dağılımını belirlemek için bir algoritma önerilmektedir. Bu algoritma yüksek bir hesaplama karmaşıklığına sahip olduğundan, bir CAN veriyolunda çok fazla mesaj olması durumunda uygulanamamaktadır. Ayrıca, deneysel bulgular tepki zamanı dağılımının daha çok düğümler arasındaki faz farkının ilk değerine bağlı olduğunu göstermektedir. Bu nedenle hesaplanan gerçek tepki zamanı dağılımı ölçümlerde gözlenmemektedir. Buna çözüm olarak bu tez, yerel tepki zamanı hesaplamasını önermekte ve düğümler arasındaki faz farkını sınırlayan bir zayıf senkronizasyon metodu geliştirmekte ve uygulamaktadır. Sonuçta hesaplanan yerel tepki zamanı dağılımlarının ölçülen tepki zamanı dağılımlarıyla oldukça iyi eşleştiği görülmektedir.

Anahtar Kelimeler: Denetleyici Alan Ağı (CAN), Ofset Çizelgeleme, En Kötü Durum Tepki Zamanı, Tepki Zamanı Dağılımı, Zayıf Senkronizasyon

To My Family

ACKNOWLEDGMENTS

I would like to express my great appreciations to my supervisor Assoc. Prof. Dr. Klaus Werner Schmidt and co-supervisor Prof. Dr. Şenan Ece Güran Schmidt for their support and guidance throughout this thesis. I am thankful for their guidance, which was very helpful in my research and writing of the thesis.

I would also like to thank TÜBİTAK-BİDEB for their financial support during my graduate education. I am supported by TÜBİTAK-BİDEB MS scholarship.

I wish to thank ASELSAN A.Ş. for giving me the opportunity of continuing my education and providing financial support for my conference attendances.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

CAN	Controller Area Network
ECU	Electronic Control Unit
CSMA/CR	Carrier Sense Multiple Access/Collision Resolution
WCRT	Worst Case Response Time
ID	Identifier (Priority)
LCM	Least Common Multiple
TW	Time Window
LD	Load Distribution
MND	Maximum Neighbor Distance
MBD	Maximum Bit Distance
ABD	Accumulated Bit Distance
NS	Neighborhood Search
LIN	Local Interconnect Network
UART	Universal Asynchronous Receiver-Transmitter
MCU	Microcontroller Unit
CPU	Central Processing Unit
PCB	Printed Circuit Board
PCMCIA	Personal Computer Memory Card International Association
PMF	Probability Mass Function
CDF	Cumulative Distribution Function
WS	Weak Synchronization

CHAPTER 1

INTRODUCTION

General

The most commonly used communication network in current in-vehicle applications is the controller area network (CAN) [15, 17, 4, 14]. CAN is a priority-based non-preemptive communication bus with a maximum data rate of 1 Mbit/s. Due to the priority-based arbitration it is expected that low-priority CAN messages observe large response times (RTs) [10].

When designing the communication schedule on a CAN bus, deterministic and probabilistic requirements are taken into account. Most of the existing literature focuses on deterministic requirements, whereby the main aim is to ensure that the worst-case response time (WCRT) of each message is smaller than the message deadline. Hereby, the deadline specifies the longest allowable time between the message generation on the transmitter node and the reception on the receiver node [16, 4]. In addition to this basic requirement, it is desired in practical applications to obtain small response times with a sufficient time difference (slack) to the message deadline [5, 6, 14]. A further desirable property is to achieve a balanced distribution of WCRTs such that messages with the same deadline also have similar WCRTs.

The study of the probabilistic properties of message response times is based on the observation that, although it is possible that a message experiences its WCRT, this case is highly unlikely [12, 20]. In particular CAN messages can experience very different response times due to two factors that introduce nondeterminism. First, the CAN protocol inserts stuff bits during the message transmission such that the CAN message transmission time varies with its payload data. Second, the nodes on a

CAN bus are not synchronized such that the phases between nodes change over time. Hence, it is important in practice to determine the probability of certain response times, which are summarized in the so-called response time distribution. Then, the most relevant probabilistic requirement is to compute this response time distribution which allows determining the probability of exceeding each possible response time.

Several research works in the literature address the stated deterministic and probabilistic requirements. Regarding the deterministic requirements, the idea of *offset scheduling* has been introduced [10, 11]. That is, in order to avoid messages bursts from individual CAN nodes, the simultaneous release of messages from the same CAN node is avoided. Instead, messages are released with an offset. In the setting of offset scheduling, the existing literature provides algorithms for computing WCRTs for each message [8, 19, 3, 18], assuming that the offset for each message is preassigned. Considering the offset assignment, [10] presents a *load distribution* (LD) heuristic that aims at minimizing the message WCRTs.

Regarding the probabilistic requirements, different problems are addressed. The work in [13, 12] studies the probabilistic WCRT. Considering the variable bit stuffing only, the probability of different response times in a worst-case message generation scenario is determined. Although this work gives an indication about how likely it is to observer the WCRT, it only considers a worst-case message generation scenario and does not capture the idea of offset scheduling. The estimation of the response time distribution for a target message that is generated by a target node is proposed in [20]. This work is based on the definition of a characteristic message that represents the transmission characteristics of each remote node node (not the target node). The response time distribution is then computed using the messages of the target node and the characteristic messages of the remote nodes. A disadvantage of this work is that optimistic estimates of the response time distribution might be obtained.

This thesis presents contributions for both the offset assignment and the response time distribution computation. Regarding the offset assignment, the main contribution of this thesis is the development of new algorithms for the offset assignment on CAN. First, three algorithms that directly assign an offset to each CAN message are presented. These algorithms use additional information compared to the existing LD algorithm without an observable increase in the computational run-time. Second an algorithm that performs an iterative neighborhood search (NS) for better offset assignments in order to improve the WCRT of each message is proposed. Since this algorithm requires WCRT computations for each message, this algorithm has a longer (but still practical) computational run-time than the other algorithms. In order to evaluate the proposed algorithms, the thesis performs comprehensive computational experiments. Our evaluation shows that the proposed NS algorithm provides the best offset assignments according to the specified performance criteria in most of the cases. In addition, it is observed that the remaining algorithms (including the existing LD algorithm) can be used in most of the cases to obtain suitable offset assignments with shorter computation times.

Regarding the probabilistic analysis, the thesis develops an original method for computing the response time distribution for non-preemptive systems such as CAN. The proposed method is based on the computation of a backlog distribution for the target message that is due to the interference of other messages and the execution time of the target message. The method is first stated for given node phases. Then, averaging over all node phases allows obtaining the overall response time distribution for the given target message. Moreover, the thesis makes an additional observation from practical measurements: although nodes on CAN are not synchronized, phases between nodes remain nearly constant for a certain amount of time in the order of several tens of minutes. The main consequence of this observation is that locally (for a short period of time), the response time distribution of the current combination of node phases is observed. This response time distribution then changes over time due to the clock drift and is generally very different from the computed overall response time distribution. As a novel contribution, this thesis proposes to evaluate the local response time distribution and to enforce the validity of this local response time distribution by a weak synchronization of all CAN nodes. As a result, it is possible to keep the node phases within a small range and to perform a very exact computation of local response time distribution that does not change over time. This is the first method for the computation of response time distributions that can be evaluated computationally and that shows good agreement with measured response time distributions. All computational results in the thesis are supported by hardware measurements.

In summary, the main contributions of the thesis are listed as follows.

1. New offset assignment algorithms for CAN with low complexity are developed and evaluated
2. A neighborhood search algorithm for the offset assignment on CAN is proposed
3. An original algorithm for computing the response time distribution of a target message is developed
4. The local response time distribution is defined and a tight range for node phases is established using weak synchronization

The remainder of the thesis is organized as follows. Chapter 2 provides the necessary background on offset scheduling on CAN and introduces the relevant performance metrics. In addition, the existing LD algorithm is described and its potential limitations are discussed. In Chapter 3, four new algorithms for the offset assignment on CAN are proposed. The different offset assignment algorithms are compared by means of comprehensive computation experiments. Moreover, Chapter 4 develops a hardware measurement setup for CAN message response times and evaluates the proposed algorithms. Algorithms for the computation of response time distributions for CAN messages are developed and validated by measurements in Chapter 5. Chapter 6 gives conclusions.

CHAPTER 2

CONTROLLER AREA NETWORKS: BACKGROUND

CAN is an asynchronous multi-master serial data bus that was designed by Robert Bosch GmbH in 1983 and was standardized in 1993 [15]. The operation of CAN is based on Carrier Sense Multiple Access/Collision Resolution (CSMA/CR) and the maximum data rate of CAN is 1 Mbit/s. Fig. 2.1 depicts the layout of a standard format CAN data frame. Each CAN frame has a frame header and contains up to 8 bytes of data. The frame header comprises a unique CAN identifier (ID) of length 11 bit or 29 bit. The ID of the message serves two purposes. First, it determines the priority of the message among the messages contending for the bus. Second, it identifies the message and thus receiver nodes can use a filter mechanism to discard unnecessary messages.

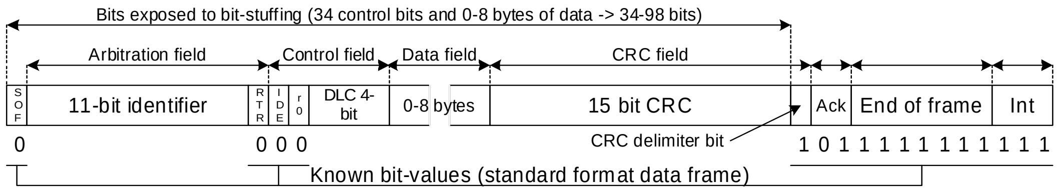


Figure 2.1: Standard Format CAN Data Frame.

Together, the maximum *frame duration* for b data bytes is [4]

$$C = (55 + 10b)\tau_{\text{bit}} \text{ (11-bit)}; \; C = (80 + 10b)\tau_{\text{bit}} \text{ (29-bit)}. \tag{2.1}$$

CAN nodes can start a message transmission whenever the bus is idle. If multiple nodes start to transmit simultaneously, a non-destructive bit-wise arbitration is applied on the CAN IDs such that the CAN ID with the lowest binary value wins the arbitration. A node loosing the arbitration retransmits its frame when the bus becomes idle again.

2.1 Scheduling Model

We assume that a set $\mathcal{M}$ of CAN messages has to be transmitted on a CAN network with data rate B bits/sec. The corresponding bit time is denoted as $\tau_{\text{bit}} = 1/B$. From the networking perspective, the relevant parameters of each message $M \in \mathcal{M}$ are given by the recurrence *period* T_M, the *message length* L_M in bit, the *deadline* D_M and the node that generates the message N_M. Here, T_M is considered as the minimum inter-arrival time of message M, which denotes the message period for periodic messages. D_M is the longest allowable time from the message generation to the end of its successful transmission. Assuming a time unit of 1 ms, T_M and D_M are measured in ms. We further write $\mathcal{M}_D = \{M \in \mathcal{M} | D_M = D\}$ for the set of messages with deadline D. Each message $M \in \mathcal{M}$ has a unique priority P_M such that messages with a smaller value of P_M have a higher priority. The bus load on CAN is defined as

$$bl = \sum_{M \in \mathcal{M}} \frac{L_M}{T_M}. \tag{2.2}$$

2.2 Scheduling with Offsets

The classical usage of CAN is such that whenever a message is generated by a task running on an ECU, it is directly released to the respective CAN hardware buffer and it enters the CAN arbitration. As a consequence, it is possible that bursts of messages can be released by an ECU, leading to long response times for the messages of the burst and, at the same time, blocking messages from other ECUs. The idea of offset scheduling [10, 11] was introduced in order to circumvent this problem. On each ECU, time windows (TWs) that repeat periodically are introduced. Then, instead of allowing message releases at any time, each message $M \in \mathcal{M}$ is assigned to specific TWs starting from a *base offset* O_M. In this thesis, we assume that TWs have a duration of 1 ms. The base offset denotes the position of each message with respect to the first TW and the number of TWs of a node $N \in \mathcal{N}$ is given by its *hyperperiod* HP_N as the least common multiple of its message periods:

$$HP_N = \text{lcm}(\{p_M | M \in \mathcal{M}, N_M = N\}). \tag{2.3}$$

Figure 2.2 shows an example for an offset schedule for two CAN nodes N_1 and N_2 with the messages in Table 2.1. N_1 has $HP_{N_1} = 10$ TWs and N_2 has $HP_{N_2} = 20$ TWs. All occupied TWs are shown in gray.

Table2.1: Message Properties

M	M_1	M_2	M_3	M_4	M_5
T_M	2	5	4	2	20
L_M	120	140	130	90	160
N_M	N_1	N_1	N_2	N_2	N_2
O_M	0	0	0	1	10
TWs	0,2,4,6,8	0,5	0,4,8,12,16	1,3,5,7,9,11,13,15,17,19	10

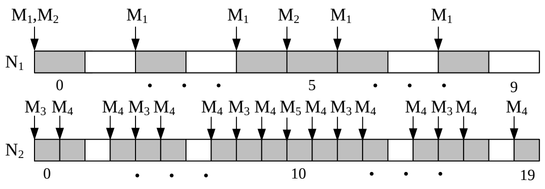


Figure 2.2: Offset scheduling example.

From the practical perspective, it is important to note that offset scheduling is a method that is exclusively applied to individual ECUs. That is, there is no synchronization among ECUs such that TWs of different ECUs are not synchronized.

2.3 Performance Metrics

We introduce the worst-case response time (WCRT) W_M of a message $M \in \mathcal{M}$ as the maximum possible time between the message generation on the transmitter node and the message reception on the receiver node. Then, the essential requirement for CAN scheduling is given by

$$W_M \leq D_M. \tag{2.4}$$

The exact WCRT time for CAN scheduling without offsets can be computed as proposed in [4]. Differently, when considering offset scheduling, the exact computation of W_M for any $M \in \mathcal{M}$ is a combinatorial problem in the number of messages. Accordingly, the literature offers several methods for computing an upper bound on W_M [8, 19, 3, 18]. In this thesis, the methods in [8, 3] are employed.

In addition to (2.4), there are further desirable properties of CAN schedules. First, it is desired to minimize the maximum WCRT of any message

$$W^{\max} = \max_{M \in \mathcal{M}} W_M. \tag{2.5}$$

Second, it is beneficial if there is a sufficient distance to a deadline violation. In this thesis, we capture this performance metric by the average slack

$$S^{\mathrm{av}} = \sum_{M \in \mathcal{M}} \frac{D_M - W_M}{D_M}. \tag{2.6}$$

Both (2.5) and (2.6) quantify the robustness of a CAN schedule to additional interference such as bit errors [5, 6, 14].

As specified in Section 2.1, CAN message sets $\mathcal{M}$ comprise subsets $\mathcal{M}_D$ of messages with the same deadline. Since messages in each group should fulfill the same timing requirements, it is desirable that such messages have similar WCRTs. To this end, we consider the average delay for messages with a certain deadline D as

$$W_D^{\mathrm{av}} = \sum_{M \in \mathcal{M}_D} \frac{W_M}{|\mathcal{M}_D|}. \tag{2.7}$$

Using (2.7), it is possible to define the standard deviation of WCRTs of messages with the same deadline D as

$$W_D^{\mathrm{std}} = \sqrt{\sum_{M \in \mathcal{M}_D} \frac{(W_M - W_D^{\mathrm{av}})^2}{|\mathcal{M}_D|}}. \tag{2.8}$$

W_D^{std} captures the WCRT variation of messages with the same deadline and should hence be small.

2.4 Offset Assignment Problem Statement

When considering offset scheduling as described in Section 2.2, it is generally assumed that the message priorities are fixed. Hence, it is desired to determine an offset assignment for all messages on the CAN bus such that (2.4) holds and the additional performance metrics are addressed. Accordingly, the problem studied in this thesis is as follows.

Problem 1. *Assume a set of messages $\mathcal{M}$ is given with T_M, L_M, D_M, N_M, P_M for each $M \in \mathcal{M}$. Determine an offset assignment O_M for all $M \in \mathcal{M}$ such that*

$$W_M \leq D_M, \forall M \in \mathcal{M}. \tag{2.9}$$

and respecting the performance metrics that are defined in Section 2.3. □

It has to be noted that the offset assignment in Problem 1 is computed offline and then implemented on electronic control units (ECUs) for each automotive application. Finding a suitable offset assignment is a difficult problem since the number of possible offset assignments is factorial in the number of messages on the CAN bus.

2.5 Offset Assignment using Load Distribution

The existing offset assignment algorithm in [10] suggests to perform *load distribution* (LD) in each node. We next describe this algorithm in a notation that agrees with the presentation in this thesis and that is different from [10]. We introduce the *time window usage* $U_N : \{0, \dots, HP_N - 1\} \mapsto \mathbb{N}$ for each node $N \in \mathcal{N}$ such that $U_N(tw)$ is the sum of message lengths occupying tw. For example, node N_1 in Fig. 2.2 has $U_{N_1}(0) = L_{M_1} + L_{M_2} = 260$, $U_{N_1}(2) = U_{N_1}(4) = U_{N_1}(6) = U_{N_1}(8) = L_{M_1} = 120$, $U_{N_1}(5) = L_{M_2} = 140$. and $U_{N_1}(1) = U_{N_1}(3) = U_{N_1}(7) = U_{N_1}(9) = 0$. Then, the left distance $d_{N,\ell} : \{0, \dots, HP_N - 1\} \mapsto \mathbb{N}$ and right distance $d_{N,\mathrm{r}} : \{0, \dots, HP_N - 1\} \mapsto \mathbb{N}$ is introduced for each TW tw such $d_{N,\ell}(tw)$ ($d_{N,\mathrm{r}}(tw)$) denotes the number of unoccupied TWs to the left (right) of tw including tw. Furthermore, $d_{N,\ell} = d_{N,\mathrm{r}} = 0$ if $U_N(tw) > 0$. For node N_1 in Fig. 2.2 $d_{N_1,\ell}(3) = d_{N_1,\mathrm{r}}(3) = 1$ and $d_{N_1,\ell}(4) = d_{N_1,\mathrm{r}}(4) = 0$.

Then, [10] proposes Algorithm 1 as a solution for Problem 1.

Algorithm 1 processes each node separately since CAN nodes are not synchronized and evolve independently in time (line 1). An offset is determined for each message of the selected node (line 3 to 9). The first message processed obtains an arbitrary offset (line 5). For the other messages, the function ComputeBestOffset in Algorithm 2 determines an offset value. After the offset selection, the time window usage U_N is updated for all TWs that are occupied by the current message (line 9). The specific

```
  input  : $\mathcal{N}$, $\mathcal{M}$, message properties
  output: Offset assignment $O_M$ for all $M \in \mathcal{M}$
1 for all $N \in \mathcal{N}$ do
2     Compute hyper-period $HP_N$
3     for all $M$ with $N_M = N$ do
4         if $M$ is the first message processed then
5             Assign $O_M = \lceil \frac{T_M}{2} \rceil$
6         else
7             $O_M$ = ComputeBestOffset($N$,$M$,$\mathcal{M}$,$HP_N$)
8         for $h = 0, \ldots, HP_N/T_M$ do
9             Set $U_N(O_M + h \cdot T_M) = 1$
```

Algorithm 1: Existing load distribution (LD) algorithm.

offset selection in [10] is formulated in the following algorithm.

```
1 Function $O_M$ = ComputeBestOffset($N$,$M$,$\mathcal{M}$, $HP_N$)
2     for $tw = 0, \ldots, T_M - 1$ do
3         minDist $= -1$
4         Determine $d_{N,\ell}(tw)$ and $d_{N,\mathrm{r}}(tw)$
5         Set dist $= \min\{d_{N,\ell}(tw), d_{N,\mathrm{r}}(tw)\}$
6         if dist $>$ minDist then
7             Set minDist $=$ dist
8             Set $O_M = tw$
9     return $O_M$
```

Algorithm 2: Offset Assignment using Load Distribution.

Algorithm 2 considers that the possible offsets for message M are $0, \ldots, T_M - 1$. Each value in this range is tried and the value with the largest minimum distance to an occupied TW is computed (line 6 to 7). Then, the offset with the largest minimum distance is assigned to the message (line 9).

As an illustration, consider the message set given in Table 2.2.

Assume that the algorithm has already assigned offset of message M_1 as 0 and trying

Table2.2: Message Properties

M	M_1	M_2
T_M	5	10
O_M	0	?

to find the best offset for message M_2. For all possible offsets, it determines the left and right distances. Then it assings the minimum of these two distances as the distance of that specific offset value. An illustration for offset candidate 2 is shown in Fig. 2.3.

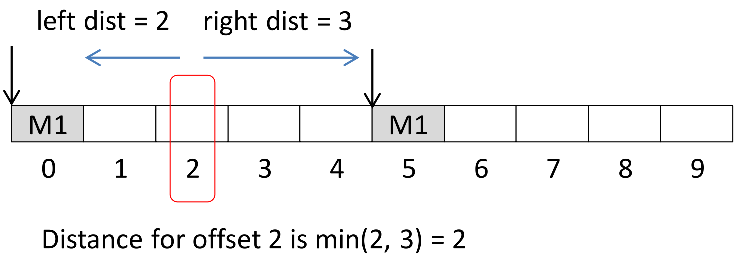


Figure 2.3: Distance Computation in LD

Eventually the offset value with the largest distance is assigned as offset to M_2 which is 2 in this example.

2.6 Discussion

Algorithm 1 tries to assign offsets such that the release of messages of each node to the CAN bus is separated in time as much as possible. In our study, we observed three potential improvements of this algorithm.

1. The distance computation in Algorithm 1 only evaluates the distance of the first TW $tw \in \{0, \ldots, T_M - 1\}$ occupied by a message. However, later instances $tw + i \cdots T_M$ of a message might observe a smaller distance.
2. The distance computation in Algorithm 1 only considers if a TW is occupied or not but omits the information about the number and size of messages occupying a TW. In principle, a larger distance value should be found if a TW is occupied by less/smaller messages.

3. The offset assignment of different nodes is performed independently. That is, the interference from other nodes is neglected when assigning offsets to each node.

The first main contribution of this thesis is the development of algorithms taking into account the above items in the subsequent section.

CHAPTER 3

IMPROVED OFFSET ASSIGNMENT ALGORITHMS

This chapter develops new algorithms for the offset assignment on CAN. First, three algorithms that directly assign an offset to each CAN message are presented in Section 3.1.1 to 3.1.3. These algorithms use additional information compared to the existing LD algorithm without an observable increase in the computational run-time. Second an algorithm that performs an iterative neighborhood search (NS) for better offset assignments in order to improve the WCRT of each message is proposed in Section 3.1.4. Since this algorithm requires WCRT computations for each message, this algorithm has a longer (but still practical) computational run-time than the other algorithms. We note that the main results of this chapter are presented in the conference papers [2, 1].

3.1 Algorithms

3.1.1 Maximum Neighbor Distance (MND)

In this section, we address item 1) in Section 2.6. To this end, we modify the function `ComputeBestOffset` in Algorithm 1. The function in Algorithm 3 now maximize the minimum neighbor distance $d_{N,\mathrm{l}}(tw)$ and $d_{N,\mathrm{r}}(tw)$ for all instances of a message.

That is, instead of computing the minimum distance only for the first instance of message M, we determine the minimum distance among all instances of message M within the hyper-period HP_N.

As an example, consider the message set shown in Table 3.1.

```
1  Function $O_M$ = ComputeBestOffset($N$,$M$,$\mathcal{M}$,$HP_N$)
2      maxDist $= 0$
3      for $o = 0, \dots, T_M - 1$ do
4          Set dist $= \infty$
5          for all $i = 0, \dots, HP_N/T_M$ do
6              Set $tw := o + i \cdot T_M$
7              Compute $d_{N,\mathrm{l}}(tw)$ and $d_{N,\mathrm{r}}(tw)$
8              if $\min\{d_{N,l}(tw), d_{N,r}(tw)\} <$ dist then
9                  Set dist $= \min\{d_\mathrm{l}, d_\mathrm{r}\}$
10         if dist $>$ maxDist then
11             maxDist = dist
12             $O_M = o$
13     return $O_M$
```
Algorithm 3: Offset Assignment using Maximum Neighbor Distance (MND).

Table3.1: Message Properties

M	M_1	M_2	M_3
T_M	4	8	4
O_M	0	6	?

Assume that messages M_1 and M_2 have already been assigned the offsets 4 and 8, respectively. The offset assignment for message M_3 is compared for LD and MND algorithms and shown in Fig. 3.1.

As can be seen, the best offset for message M_3 is 2 which is true for the first instance of M_3. However, the second instance coincides with M_2, which results in larger response time. On the other hand, the proposed MND algorithm searches for all the instances of M_3 for the best offset, which is 1 in this case.

3.1.2 Maximum Bit Distance (MBD)

In this section, we also address item 2) in Section 2.6. To this end, we introduce the left neighbor TW tw_l and the right neighbor TW tw_r for each TW tw. In Fig. 2.2

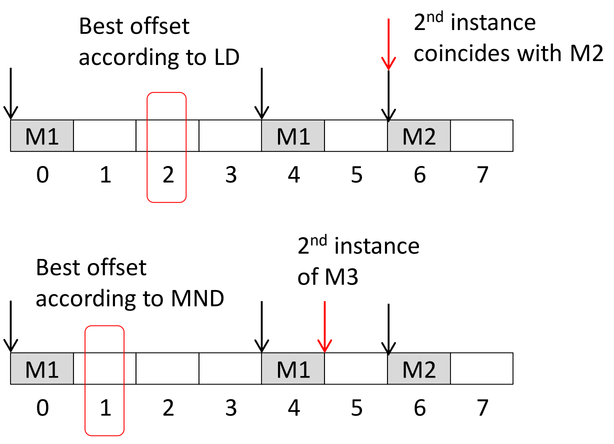


Figure 3.1: LD and MND Offset Assignment Comparison

$1_\mathrm{l} = 0$ and $1_\mathrm{r} = 2$ for TW $tw = 1$. Finally, we consider that the number of bits per TW is $L_\mathrm{TW} = 1\,\mathrm{ms}/\tau_\mathrm{bit}$. Then, we replace the function ComputeBestOffset in Algorithm 1 by the function in Algorithm 4 to determine an offset for each message.

Algorithm 4 follows the same outline as Algorithm 3. The main difference is between line 4 and 10. Here, the left (right) neighbors are computed in line 8 and the minimum distance is computed taking into account the number of free bits $d_{N,\mathrm{l}}(tw) \cdot L_\mathrm{TW}$ ($d_{N,\mathrm{r}}(tw) \cdot T_\mathrm{TW}$) and the number of occupied bits $U_N(tw_\mathrm{l})$ ($U_N(tw_\mathrm{r})$) in line 9. That is, Algorithm 4 always assigns the offset with the largest minimum distance (line 11 to 13).

As an example, consider the case shown in Fig.3.2. As can be seen, there are two messages occupying the TW TW $tw = 0$. However the LD algorithm does not take this into account and computes the distance for offset 2 as 2, whereas the actual distance is 1 as computed by the MBD algorithm.

3.1.3 Accumulated Bit Distance

We note that all previous algorithms only consider the largest minimum distance among all message instances in one hyper-period. Differently, our next algorithm records the accumulated minimum distance for all instances within a hyper-period. To this end, the function ComputeBestOffset in Algorithm 1 is implemented as follows.

```
Function $O_M$ = ComputeBestOffset($N$, $M$, $\mathcal{M}$, $HP_N$)
    maxDist $= -\infty$
    for $o = 0, \ldots, T_M - 1$ do
        oldDist $= \infty$
        for all $i = 0, \ldots, HP_N / T_M$ do
            Set $tw = o + i \cdot T_M$
            Determine $d_{N,\mathrm{l}}(tw)$ and $d_{N,\mathrm{r}}(tw)$
            Determine $tw_\mathrm{l}$ and $tw_\mathrm{r}$
            dist $= \min\{d_{N,\mathrm{l}}(tw) \cdot T_\mathrm{TW} - U_N(tw_\mathrm{l}), d_{N,\mathrm{r}}(tw) \cdot T_\mathrm{TW} - U_N(tw_\mathrm{r})\}$
            dist $= \min\{$dist, oldDist$\}$
        if dist $>$ maxDist then
            Set maxDist $=$ dist
            Set $O_M = o$
    return $O_M$
```

Algorithm 4: Offset Assignment with Maximum Bit Distance (MBD).

```
Function $O_M$ = ComputeBestOffset($N$, $M$, $\mathcal{M}$, $HP_N$)
    minDist $= \infty$
    for $o = 0, \ldots, T_M - 1$ do
        dist $= 0$
        for all $i = 0, \ldots, HP_N / T_M$ do
            Set $tw = o + i \cdot T_M$
            Determine $d_{N,\mathrm{l}}(tw)$ and $d_{N,\mathrm{r}}(tw)$
            Determine $tw_\mathrm{l}$ and $tw_\mathrm{r}$
            dist $=$
              dist $+ \min\{d_{N,\mathrm{l}}(tw) \cdot T_\mathrm{TW} - U_N(tw_\mathrm{l}), d_{N,\mathrm{r}}(tw) \cdot T_\mathrm{TW} - U_N(tw_\mathrm{r})\}$
        if dist $<$ minDist then
            minDist $=$ dist
            $O_M = o$
    return $O_M$
```

Algorithm 5: Offset Assignment with Minimum Accumulated Bit Distance.

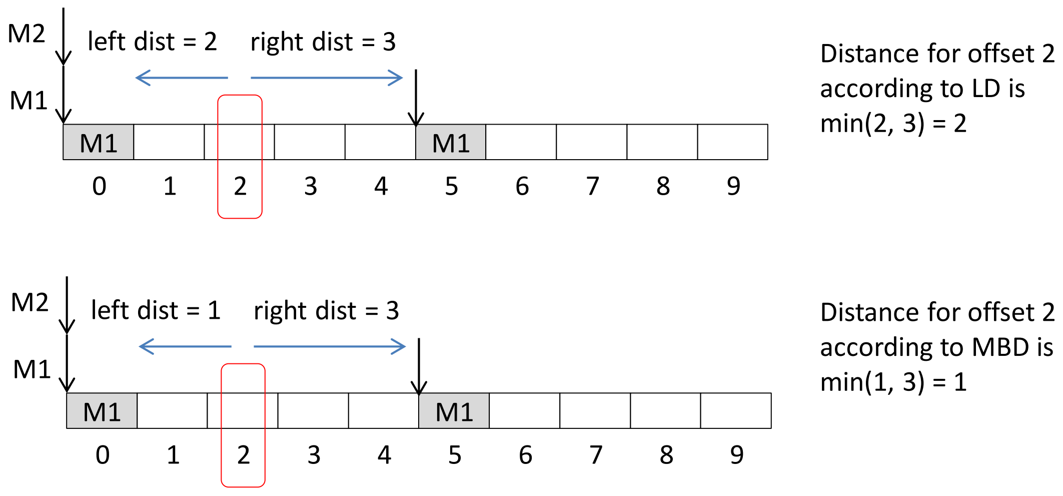

M2
left dist = 2
right dist = 3
M1
M1
M1
0
1
2
3
4
5
6
7
8
9
Distance for offset 2 according to LD is min(2, 3) = 2
M2
left dist = 1
right dist = 3
M1
M1
M1
0
1
2
3
4
5
6
7
8
9
Distance for offset 2 according to MBD is min(1, 3) = 1

Figure 3.2: Comparison of Distance Computation in LD and MBD

That is, the minimum distances in each TW for an offset candidate o are added up in line 9. Then, the offset with the minimum accumulated distance is selected in line 9 to 12.

3.1.4 Neighborhood Search (NS)

The proposed algorithms in Section 3.1.1 to 3.1.3 compute an offset assignment for each individual CAN node without taking into consideration the interference from other nodes as pointed out in item 3) in Section 2.6. Our last algorithm attempts to improve an given offset assignment by an iterative search. In each iteration, a candidate offset assignment is selected, one of the performance metrics in (2.6) to (2.8) is evaluated and the offset candidate with the best result is selected.

The evaluation of the performance metrics in (2.6) to (2.8) requires the WCRT computation for the candidate offset assignments. As is noted in Section 2.3, there are different algorithms for finding upper bounds on W_M for each $M \in \mathcal{M}$. According to our experiments, the algorithm in [3] generally provides tighter bounds on W_M but requires significantly longer computation times than the algorithm in [8]. Since the proposed Algorithm 6 requires a considerable number of WCRT computations, we use the algorithm in [8] when evaluating candidate offset assignments. The final WCRT result is then evaluated using the algorithm in [3].

```
input  : $\mathcal{N}$, $\mathcal{M}$, message properties, initial offset assignment $\hat{O}_M$ for all
         $M \in \mathcal{M}$, $\Delta_{\text{max}}$
output: Offset assignment $O_M$ for all $M \in \mathcal{M}$
Evaluate the selected performance metric as $PM^{\text{opt}}$
$PM^{\text{old}} = PM^{\text{opt}}$
$\Delta := 1$; $it = 1$
Set $O_M^{\text{old}} := \hat{O}_M$ for all $M \in \mathcal{M}$
Compute hyperperiod $HP_N$ for each $N \in \mathcal{N}$
while $it \leq nIt$ do
    for all $M \in \mathcal{M}$ do
        $F = \{\hat{O}_M - \Delta \mod HP_{N_M}, \hat{O}_M + \Delta \mod HP_{N_M}\}$
        Set $\hat{O}_M := O_M^{\text{old}}$ for all $M \in \mathcal{M}$
        for all $o \in F$ do
            $\hat{O}_M := o$
            Evaluate the selected performance metric as $PM$
            if $PM$ improves $PM^{opt}$ then
                $PM^{\text{opt}} := PM$
                for all $M \in \mathcal{M}$ do
                    $O_M^{\text{opt}} := \hat{O}_M$
                    $M^{\text{opt}} := M$
    if $PM^{opt} \neq PM^{old}$ then
        $\Delta := 1$
    else if $\Delta \leq \Delta_{max}$ then
        $\Delta := \Delta + 1$
    else
        Set $O_M := O_M^{\text{opt}}$ for all $M \in \mathcal{M}$
        return
    Set $O_M^{\text{old}} := O_M^{\text{opt}}$ for all $M \in \mathcal{M}$
    Set $PM^{\text{old}} := PM^{\text{opt}}$
    $it := it + 1$
Set $O_M := O_M^{\text{opt}}$ for all $M \in \mathcal{M}$
```

Algorithm 6: Offset Assignment with Neighborhood Search (NS).

Algorithm 6 starts from a given offset assignment that can be computed by any of the previously described methods. For the given offset assignment, the selected performance metric among (2.5) to (2.8) is evaluated and recorded (line 1 and 2) as PM^{opt} (currently optimal value) and PM^{old} (value before the next iteration). Then, in each of at most nIt iterations, different candidate offset values are selected (line 8) for each message by increasing/decreasing the currently selected offset value by Δ (Δ is initialized by 1 in line 3). For each candidate offset assignment, the selected performance metric is evaluated and the offset assignment with the best evaluation is recorded (line 10 to 17). If there was an improvement in the current iteration, Δ is set to 1 (line 19). If there was no improvement and Δ is below a maximum value Δ_{max}, Δ is incremented (line 21) in order to try different candidate offset assignments in the next iteration. Otherwise, the algorithm terminates with the currently optimal assignment (line 24) and it is assumed that no better offset assignment could be found. At the end of each iteration, the currently optimal offset assignment is recorded (line 25 and 26) and the iteration count is incremented (line 27).

We note that it is expected that Algorithm 6 has a considerably longer computation time than the remaining algorithms due to the repeated evaluation of the selected performance metric in line 12 which requires a WCRT computation.

3.2 Computational Evaluation

This section evaluates and compares the proposed offset assignment algorithms for a large set of test cases and regarding the different performance metrics in Section 2.3. First, Section 3.2.1 presents the general setting. Then, Section 3.2.2 comments on the observed computation times and Section 3.2.3 to 3.2.6 present computational results for different performance metrics and message sets. The obtained results are discussed in Section 3.2.7.

3.2.1 Setting of the Computational Experiments

We perform offset assignment experiments for a CAN bus with a data rate of 125 kbit/s. That is $\tau_{\text{bit}} = 8\mu\,\text{s}$ and $L_{\text{TW}} = 125\,\text{bit}$. We use message sets with periods and deadlines $T_M, D_M \in \{10, 20, 50, 100, 200, 1000\}$ for a chassis network as in [10]. Since offset scheduling is applied in order to reduce the WCRTs at high bus loads, we perform computations for bus loads of $k \cdot 125\,\text{kbit/s}$ with $k = 0.5$ (medium), $k = 0.7$ (high) and $k = 0.9$ (very high). In addition, we respect that CAN applications can have different numbers of nodes. We consider the case of networks with 5, 15, 25 nodes. In all our experiments, we first apply the algorithms LD (Section 2.5), MND (Section 3.1.1), MBD (Section 3.1.2), ABD (Section 3.1.3). Then, the best solution among these algorithms is used to initialize the algorithm NS (Section 3.1.4), which is used with the average slack in (2.6) as performance metric.

In the following, various parameter combinations of network type, bus load and number of nodes are investigated. For each parameter combination, 30 message sets are generated randomly and the performance metrics in Section 2.3 are evaluated by taking the average over the 30 message sets. The number of messages generated for different bus loads and network types is summarized in Table 3.2.

Table3.2: Number of Messages per Bus Load and Network Type.

	chassis			body		
k	0.5	0.7	0.9	0.5	0.7	0.9
Number of messages	70	99	130	83	117	152

In addition to evaluating the performance metrics, we also define N_x^y as the number of times, a certain algorithm $y \in \{\text{NO}, \text{LD}, \text{MND}, \text{MBD}, \text{ABD}, \text{NS}\}$ achieves the best solution regarding the performance metric $x \in \{W^{\max}, S^{\text{av}}\}$. Hereby, "NO" represents the case where offset scheduling is not applied. Then,

$$P_x^y = \frac{N_x^y}{30} \tag{3.1}$$

represents the percentage of best solutions achieved by each algorithm. Note that the sum of these percentages may exceed 100% since multiple algorithms might obtain the same best solution. Finally, we introduce $\hat{P}_x^y$ for the percentage of best solutions among all the algorithms excluding NS in Section 3.1.4. We use $\hat{P}_x^y$ in order to compare the algorithms with a short computation time.

3.2.2 Computation Times

We first summarize our observations regarding the computations times of the different algorithms. All algorithms were implemented using C++ and run on a desktop PC with Intel(R)Core(TM)2 Duo CPU @ 2.93Ghz and 8GB of RAM. The results for the different combinations of (number of nodes,bus load) are summarized in Table 3.3. It can be seen that the computation times for the algorithms LD, MND, MBD, ABD

Table3.3: Computation Times in Seconds

	LD	MND	MBD	ABD	NS
$(5, 0.5)$	0.002	0.005	0.015	0.03	17.5
$(5, 0.7)$	0 0.031	0.002	0.015	0.003	531
$(5, 0.9)$	0.1	0.003	0.016	0.002	1 820
$(15, 0.5)$	0.015	0.031	0.016	0.016	156
$(15, 0.7)$	0.03	0.016	0.031	0.016	900
$(15, 0.9)$	00.23	0.015	0.005	0.016	1 800
$(25, 0.5)$	0 0.031	0.015	0.016	0.031	136
$(25, 0.7)$	0.005	0.031	0.031	0.016	480
$(25, 0.9)$	0.015	0.031	0.016	0.015	2 100

are negligible in all cases. Increasing computations times depending on the number of nodes and the bus load are observed for the algorithm NS. Considering that the offset assignment is computed offline, it holds that these computation times are still practicable.

3.2.3 Maximum WCRT Comparison

In this section, we compare the different algorithms regarding the maximum WCRT in (2.5). The average results are shown in Table 3.4 to 3.6.

The general observation from Table 3.4 is that the maximum WCRTs in increase with an increasing bus load and number of nodes. It is further observed that NS provides the smallest values in almost all the cases. This is expected since NS starts with the best solution among the other algorithms. Nevertheless, it has to be noted that NS uses the WCRT algorithm in [8] in order to achieve practical computation times. Hence, deviations from the expected improved performance as for example seen for

Table3.4: $W^{\max}$ Comparison

	NO	LD	MND	MBD	ABD	NS
$(5, 0.5)$	87.20	9.48	9.71	9.10	9.20	9.04
$(5, 0.7)$	175.66	27.07	31.15	18.25	19.08	16.93
$(5, 0.9)$	356.7	139.3	164.6	65.4	76.4	64.3
$(15, 0.5)$	87.73	26.36	26.81	25.62	25.58	25.31
$(15, 0.7)$	175.94	51.62	49.87	50.42	50.60	47.27
$(15, 0.9)$	384.23	156.74	162.49	166.24	160.57	143.27
$(25, 0.5)$	88.29	40.90	40.40	40.36	40.32	40.34
$(25, 0.7)$	175.18	84.73	84.96	88.33	88.05	83.89
$(25, 0.9)$	376.58	218.12	215.06	218.16	218.21	211.71

the case $(25, 0.5)$ are possible. Regarding the remaining algorithms, it can be seen that the proposed MDB and ABD algorithms provide better results than the existing LD algorithm especially for small numbers of nodes.

Table3.5: $P^{y}_{W^{\max}}$ Comparison

	NO	LD	MND	MBD	ABD	NS
$(5, 0.5)$	0.0%	46.7%	43.3%	60.%	46.7%	86.7%
$(5, 0.7)$	0.0%	3.3%	0.0%	23.3%	16.7%	76.7%
$(5, 0.9)$	0.0%	0.0%	0.0%	33.3%	13.3%	60.0%
$(15, 0.5)$	0.0%	33.3%	16.7%	16.7%	30.0%	63.3%
$(15, 0.7)$	0.0%	3.3 %	16.7%	13.3%	6.7%	70.0%
$(15, 0.9)$	0.0%	26.7%	13.3%	10.0%	10.0%	53.3%
$(25, 0.5)$	0.0%	63.3%	56.7%	63.3%	70.0%	70.0%
$(25, 0.7)$	0.0%	46.7%	33.3%	3.3%	10.0%	60.0%
$(25, 0.9)$	0.0%	26.7%	43.3%	3.3%	6.7%	60.0%

Table 3.5 suggests that NS always has the highest percentage of solutions. Regarding the algorithms without search, there is not much difference for small bus loads. This is expected since there is enough free space for assigning offsets. For bus loads above $k = 0.7$, MBD and ABD provide better results if the number of nodes is small. Conversely, LD and MND show better results in the case of many nodes.

In summary, NS proves to be the best algorithm when measuring $W^{\max}$, whereas different algorithms among LD, MND, MBD, ABD are preferable depending on the bus load and the number of nodes if it is desired to avoid a time-consuming search.

Table3.6: $\hat{P}^{y}_{W^{\max}}$ Comparison

	NO	LD	MND	MBD	ABD
$(5, 0.5)$	0.0%	63.3%	46.7%	80.0%	63.3%
$(5, 0.7)$	0.0%	10.0%	3.3%	50.0%	63.3%
$(5, 0.9)$	0.0%	3.3%	3.3%	66.7%	33.3%
$(15, 0.5)$	0.0%	43.3%	23.3%	30.0%	50.0%
$(15, 0.7)$	0.0%	16.7%	53.3%	$40.\%$	26.7%
$(15, 0.9)$	0.0%	36.7%	40.0%	16.7%	20.0%
$(25, 0.5)$	0.0%	63.3%	56.7%	63.3%	70.0%
$(25, 0.7)$	0.0%	63.3%	50.0%	3.3%	10.0%
$(25, 0.9)$	0.0%	26.7%	50.0%	10.0%	16.7%

3.2.4 Slack Comparison

We next perform a comparison of the average slack values in (2.6) for the different algorithms. To this end, we show the slack improvement of each algorithm compared to the case without offset in Table 3.7 to 3.8. It can be seen that the average slack

Table3.7: S^{av} Comparison

	NO	LD	MND	MBD	ABD	NS
$(5, 0.5)$	0.0	0.523	0.522	0.526	0.524	0.527
$(5, 0.7)$	0.0	0.491	0.495	0.492	0.492	0.511
$(5, 0.9)$	0.0	0.774	0.720	0.914	0.894	0.915
$(15, 0.5)$	0.0	0.374	0.373	0.375	0.376	0.379
$(15, 0.7)$	0.0	0.491	0.495	0.492	0.492	0.511
$(15, 0.9)$	0.0	0.603	0.587	0.566	0.576	0.620
$(25, 0.5)$	0.0	0.269	0.269	0.271	0.271	0.271
$(25, 0.7)$	0.0	0.345	0.350	0.344	0.356	0.360
$(25, 0.9)$	0.0	0.414	0.419	0.398	0.404	0.441

always increases when using offset scheduling. Hereby, the largest improvement is achieved by the NS algorithm. This is expected since the average slack is used as performance metric. This result is also confirmed by Table 3.8.

Regarding the algorithms without search, it can be observed from Table 3.9 that MBD performs well for small numbers of nodes or small bus load, whereas MND obtains better results for larger bus loads and numbers of nodes.

Table3.8: $P^{y}_{S^{\text{av}}}$

	NO	LD	MND	MBD	ABD	NS
$(5, 0.5)$	0.0%	33.3%	13.3%	56.7%	16.7%	70.0%
$(5, 0.7)$	0.0%	3.3%	0.0%	23.3%	16.7%	76.7%
$(5, 0.9)$	0.0%	3.3%	6.7%	26.7%	16.7%	50.0%
$(15, 0.5)$	0.0%	33.3%	16.7%	16.7%	30.0%	63.3%
$(15, 0.7)$	0.0%	0.0%	0.0%	0.0%	6.7%	96.7%
$(15, 0.9)$	0.0%	30.0%	10.0%	3.3%	0.0%	60.0%
$(25, 0.5)$	0.0%	46.7%	20.0%	53.3%	56.7%	60.0%
$(25, 0.7)$	0.0%	16.7%	23.3%	13.3%	23.3%	76.7%
$(25, 0.9)$	0.0%	13.3%	20.0%	3.3%	6.7%	80.00%

Table3.9: $\hat{P}^{y}_{S^{\text{av}}}$

	NO	LD	MND	BD	ABD
$(5, 0.5)$	0.0%	36.7%	16.7%	73.3%	23.3%
$(5, 0.7)$	0.0%	10.0%	3.3%	50.%	63.3%
$(5, 0.9)$	0.0%	3.3%	6.7%	70.%	26.7%
$(15, 0.5)$	0.0%	43.3%	23.3%	30.0%	50.0%
$(15, 0.7)$	0.0%	6.7%	43.3%	20.0%	33.3%
$(15, 0.9)$	0.0%	33.3%	33.3%	16.7%	26.7%
$(25, 0.5)$	0.0%	46.7%	20.0%	53.3%	56.7%
$(25, 0.7)$	0.0%	26.7%	33.3%	20.0%	30.0%
$(25, 0.9)$	0.0%	23.3%	50.0%	13.3%	23.3%

3.2.5 Average WCRT Comparison

This section investigates the performance metrics in (2.7) and (2.8) for messages with a medium deadline of $D_M = 50$ ms and a large deadline of $D_M = 1000$ ms. Messages with small deadlines are omitted from this study since only a small number of such messages (usually 1) is in the generated message sets, which is not suitable for a statistical evaluation.

It can be seen from Table 3.10 for $D_M = 50$ ms that offset scheduling leads to an improvement compared to the case without offset scheduling (NO) in all cases. In addition, it holds that the NS and MBD algorithms achieve the smallest value of W^{av}_{50} in almost all cases.

Regarding the performance metric W^{std}_{50} in (2.8) for messages with deadline $D_M =$

Table3.10: W_{50}^{av} Comparison

	NO	LD	MND	AND	MBD	ABD	NS
$(5, 0.5)$	5.84	5.00	4.97	4.77	4.60	4.97	4.56
$(5, 0.7)$	8.46	6.94	6.77	6.42	6.22	6.70	6.28
$(5, 0.9)$	10.97	8.37	8.02	7.71	7.55	8.00	7.57
$(15, 0.5)$	5.79	5.43	5.43	5.34	5.24	5.43	5.27
$(15, 0.7)$	8.49	7.89	7.77	7.63	7.62	7.75	7.66
$(15, 0.9)$	10.23	9.49	9.42	9.30	9.34	9.34	9.35
$(25, 0.5)$	5.83	5.76	5.76	5.73	5.68	5.76	5.69
$(25, 0.7)$	8.41	8.00	7.94	7.89	7.88	7.93	7.90
$(25, 0.9)$	11.04	10.46	10.42	10.56	10.42	10.41	10.41

50 ms, it can be concluded from Table 3.11 that the smallest value of W_{50}^{std} is generally not obtained by the algorithm that minimizes W_{50}^{av}. That is, it is difficult to minimize both performance metrics in (2.7) and (2.8) simultaneously.

Table3.11: W_{50}^{std} Comparison

	NO	LD	MND	AND	MBD	ABD	NS
$(5, 0.5)$	2.96	0.76	0.72	0.70	0.73	0.68	0.74
$(5, 0.7)$	2.15	1.57	1.46	1.42	1.39	1.39	1.38
$(5, 0.9)$	3.14	1.81	1.62	1.69	1.76	1.58	1.64
$(15, 0.5)$	1.42	1.33	1.33	1.36	1.31	1.33	1.33
$(15, 0.7)$	2.18	1.86	1.78	1.80	1.88	1.77	1.86
$(15, 0.9)$	3.03	2.53	2.55	2.62	2.61	2.54	2.59
$(25, 0.5)$	1.42	1.35	1.35	1.37	1.34	1.35	1.33
$(25, 0.7)$	2.17	1.96	1.94	1.96	1.99	1.94	1.97
$(25, 0.9)$	3.17	2.71	2.68	2.76	2.73	2.67	2.68

Considering messages with deadline $D_M = 1000$ ms, Table 3.12 indicates that the NS algorithm clearly improves W_{1000}^{av}. This effect is most visible for high bus loads. Regarding the algorithms without search, different algorithms are suitable for different bus loads and numbers of nodes.

Looking at W_{1000}^{std} in Table 3.13, it turns out that MBD, ABD, NS are advantageous if the number of nodes is small, whereas LD and NS are preferable for larger bus loads and number of nodes.

Table3.12: W_{1000}^{av}

	NO	LD	MND	AND	MBD	ABD	NS
$(5, 0.5)$	59.50	8.69	8.86	8.85	8.53	8.63	8.46
$(5, 0.7)$	112.99	17.22	18.46	17.44	14.14	14.79	14.38
$(5, 0.9)$	204.95	64.15	74.87	86.00	43.32	41.19	37.53
$(15, 0.5)$	59.85	22.67	22.82	22.62	22.58	22.42	22.24
$(15, 0.7)$	113.13	43.21	41.92	42.99	42.40	42.35	39.78
$(15, 0.9)$	216.48	89.62	94.27	108.47	97.80	95.90	87.26
$(25, 0.5)$	60.20	33.39	33.40	33.33	33.25	33.24	33.25
$(25, 0.7)$	112.57	62.31	62.03	63.36	62.87	62.61	60.47
$(25, 0.9)$	210.75	132.43	132.31	138.35	136.48	135.31	128.34

Table3.13: W_{1000}^{std}

	NO	LD	MND	AND	MBD	ABD	NS
$(5, 0.5)$	18.13	0.58	0.56	0.51	0.47	0.43	0.46
$(5, 0.7)$	40.57	4.54	5.56	4.28	2.12	2.34	1.81
$(5, 0.9)$	88.97	30.94	35.59	30.31	11.45	13.79	11.5
$(15, 0.5)$	18.18	2.41	2.60	2.33	2.24	2.26	2.08
$(15, 0.7)$	40.59	5.19	5.09	6.47	6.16	6.28	5.16
$(15, 0.9)$	94.85	27.66	32.91	43.95	35.29	34.76	29.10
$(25, 0.5)$	18.29	5.89	5.99	6.03	6.04	6.03	6.03
$(25, 0.7)$	40.49	15.78	16.58	17.88	17.58	17.47	16.15
$(25, 0.9)$	91.61	46.26	47.47	51.31	49.30	49.46	46.64

3.2.6 Example WCRT Result

Fig. 3.3 shows the WCRTs of all messages on an example chassis network with 15 nodes and a bus load with $k = 0.7$. In this example, NS and MND mostly achieve smaller WCRTs than the existing LD algorithm. However, for this particular example, MBD is not suitable.

3.2.7 Discussion

In summary, our results suggest that the proposed algorithms allow finding offsets for reducing the WCRTs of messages on CAN in most of the cases. Hereby, three important observations have to be emphasized:

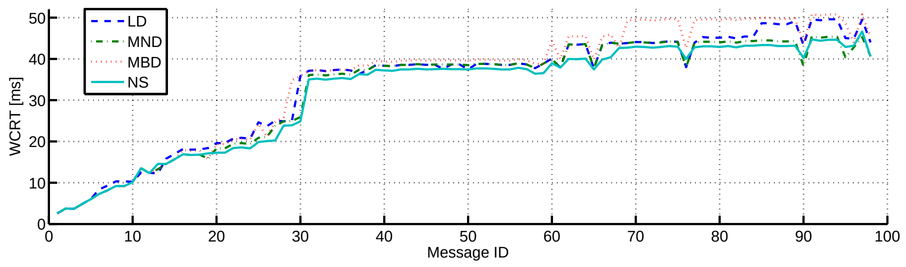


Figure 3.3: WCRT Comparison for LD, MND, MBD, NS using 15 Nodes and $k = 0.7$.

1. In the majority of the cases, the proposed algorithms outperform the existing LD algorithm.
2. There is no unique algorithm that always achieves the best result. It is clearly observed from Table 3.5 and 3.8 that any of the studied algorithms has the potential to improve the respective performance metric depending on the specific test case (message set).
3. The algorithms LD, MND, MBD, ABD have very small computation times in the order of milliseconds. Only the NS algorithm requires longer computation times in the order of 30 min for high bus loads.

Considering the stated points, we suggest to apply all of the presented algorithms and select the best result when determining offset assignments for a specific message set $\mathcal{M}$.

CHAPTER 4

EXPERIMENTAL TESTS

In order to verify the theoretical results in real hardware, a test setup is constructed and a response time measurement method is developed. CAN is an asynchronous bus where each node has its own clock. In order to measure the response time of CAN messages, a global clock that is synchronous among all of the nodes is required. This is achieved using the FlexRay protocol. The hardware used in the experiments is selected accordingly.

4.1 Development and Test Environment

To have a better understanding about how the experiments are held and how the projects are developed, the hardware and the tools used during the studies will be explained briefly in the following sections.

4.1.1 Fujitsu SK-91465X-100PMC Evaluation Board

The SK-91465X-100PMC is the main building block of the experiments. It is a multi-functional evaluation board developed by Fujitsu for the 32-bit FR60 Flash microcontroller series MB91F465XA which is tailored for automotive applications. The board includes both CAN and FlexRay interfaces which makes it suitable for response time measurement experiments. Besides the CAN and FlexRay support, the evaluation board also has LIN and UART channels. As a whole, the board has 2 FlexRay, 2 CAN, 2 LIN/UART and a dedicated UART interfaces. In addition to the communication interfaces, it also includes 8 user LEDs and 6 user push buttons.

The two FlexRay channels (namely Channel A and Channel B) are redundant. The physical layer of these channels is implemented by AMS8221B transceiver. Different from the CAN, data is not directly transmitted by the microcontroller to the transceiver. Instead, the transceiver is directly connected to the MB88121 series Standalone Communication Controller and the CPU controls and configures this controller.

The physical layer of CAN channels is implemented by TLE620GV33 high speed transceivers.

The SK-91465X-100PMC multifunctional evaluation board is shown in Fig. 4.1.

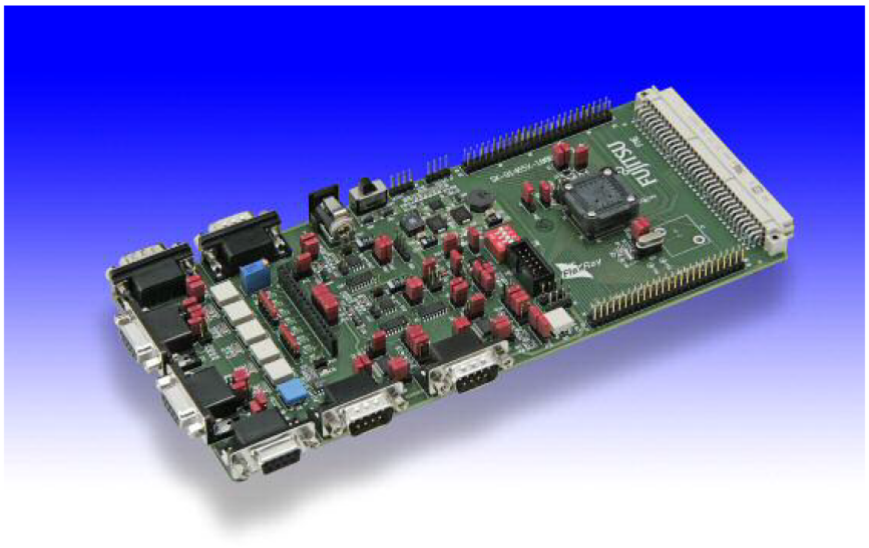
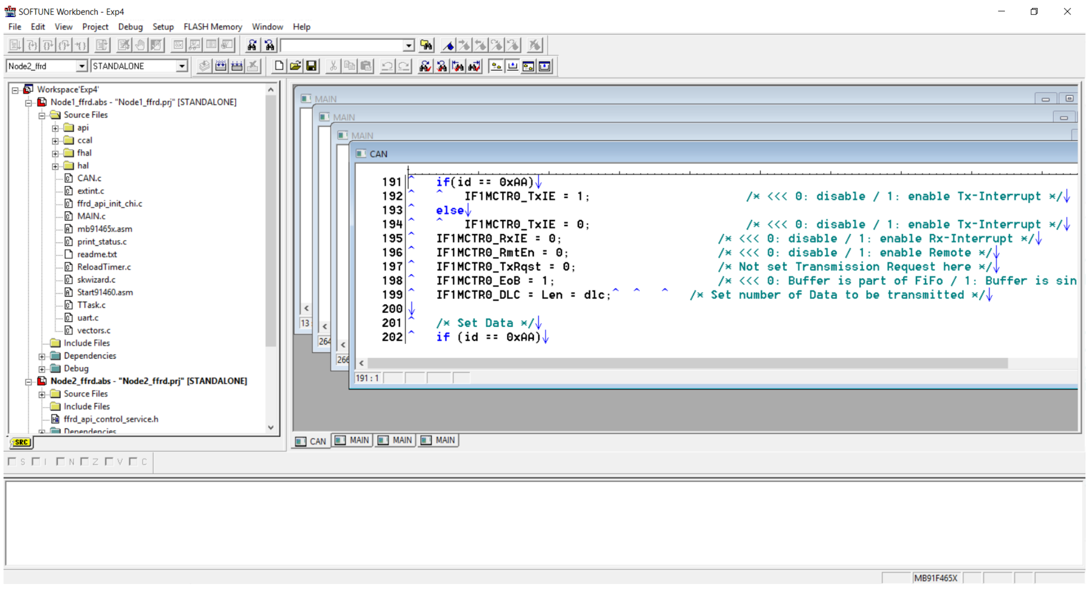


Figure 4.1: SK-91465X-100MPC Evaluation Board

4.1.2 Softune Workbench Software Development Environment

Softune Workbench is developed by Fujitsu as the development environment for the FR family microcontrollers. It has all the necessary interfaces for creating, editing, building and debugging the projects. All the projects used in the experiments are developed in C programming language by the V60L08 version of the Softune Workbench. After the build process Softune Workbench creates a *.mhx file as the output which can be directly downloaded to the flash memory of the microcontroller MB91F465XA. Softune Workbench has the ability to combine and save multiple projects in a single workspace which is useful for working with multiple projects at the same time. A view of the Softune Workbench can be seen in Fig. 4.2.



```
    if(id == 0xAA)
        IF1MCTR0_TxIE = 1;                          /* <<< 0: disable / 1: enable Tx-Interrupt */
    else
        IF1MCTR0_TxIE = 0;                          /* <<< 0: disable / 1: enable Tx-Interrupt */
    IF1MCTR0_RxIE = 0;                          /* <<< 0: disable / 1: enable Rx-Interrupt */
    IF1MCTR0_RmtEn = 0;                         /* <<< 0: disable / 1: enable Remote */
    IF1MCTR0_TxRqst = 0;                        /* Not set Transmission Request here */
    IF1MCTR0_EoB = 1;                           /* <<< 0: Buffer is part of FiFo / 1: Buffer is sin
    IF1MCTR0_DLC = Len = dlc;               /* Set number of Data to be transmitted */

    /* Set Data */
    if (id == 0xAA)
```

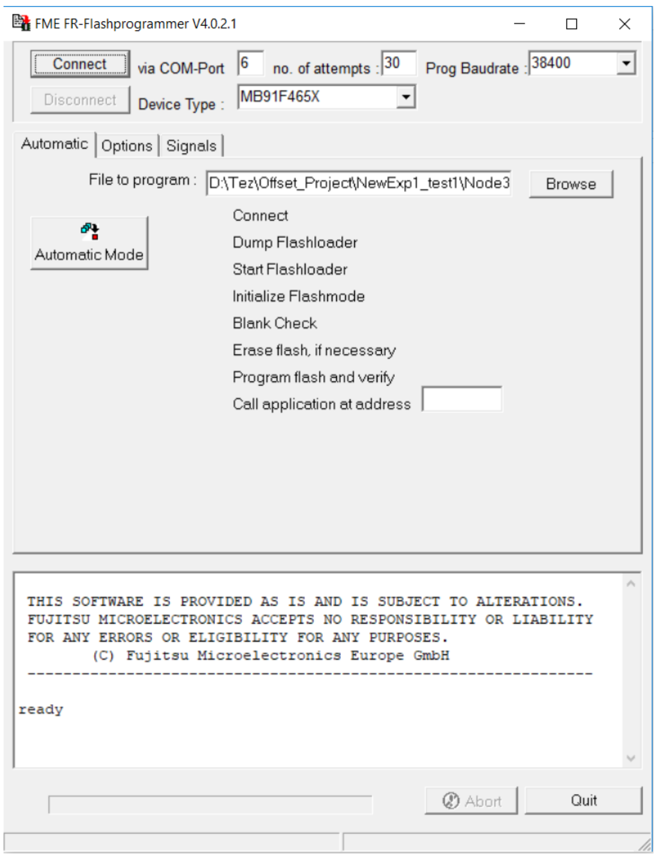
Figure 4.2: FR Family SOFTUNE Workbench V60L08

4.1.3 FME FR Flash Programmer

FME FR Flash Programmer is developed by Fujitsu and is used to download the projects into the flash memory of the microcontroller. FR Flash Programmer is capable of downloading the code with RS-232 serial interface, eliminating the need for an emulator. Besides having many additional features, it has an automatic mode which automatically connects to the microcontroller, erases the necesseray flash sectors and programs the flash with the machine code located in the *.mhx file with a single button. Throughout the experiments, the FME FR Flash Programmer V4.0.2.1 is used to program the boards. Fig. 4.3 shows a view from the FME FR Flash Programmer.

4.1.4 FlexRay Communication Controller Driver

As stated in section 4.1.1, FlexRay message handling and communication tasks are not performed by the CPU in the SK-91465X-100PMC evaluation board. These tasks are handled by a specific communication controller located on the board which is Bosch ERay series Standalone Communication Controller.



```
THIS SOFTWARE IS PROVIDED AS IS AND IS SUBJECT TO ALTERATIONS.
FUJITSU MICROELECTRONICS ACCEPTS NO RESPONSIBILITY OR LIABILITY
FOR ANY ERRORS OR ELIGIBILITY FOR ANY PURPOSES.
        (C) Fujitsu Microelectronics Europe GmbH
---------------------------------------------------------------

ready
```

Figure 4.3: FME FR Flash Programmer V4.0.2.1

The task of the microcontroller is to configure and manage this controller. For this purpose, Fujitsu offers a library named Fujitsu Communication Controller Driver which implements all the necessary functions and structures for communication between the MCU and the communication controller. This library is included in all of the projects where FlexRay communication is needed.

4.1.5 FlexCard Cyclone II SE

The FlexCard Cyclone II SE is a network analyzer hardware developed by Eberspacher Electronics. It is a CardBus card which is connected to a personal computer through PCMCIA slot. FlexCard Cyclone II SE supports two FlexRay and two high speed CAN channels which makes it suitable to monitor and analyze both FlexRay and CAN network at the same time with a single hardware interface. FlexCard Cyclone II SE is also able to send messages to both FlexRay and CAN networks connected to it. The FlexCard Cyclone II SE hardware is shown in Fig. 4.4.

Figure 4.4: FlexCard Cyclone II SE

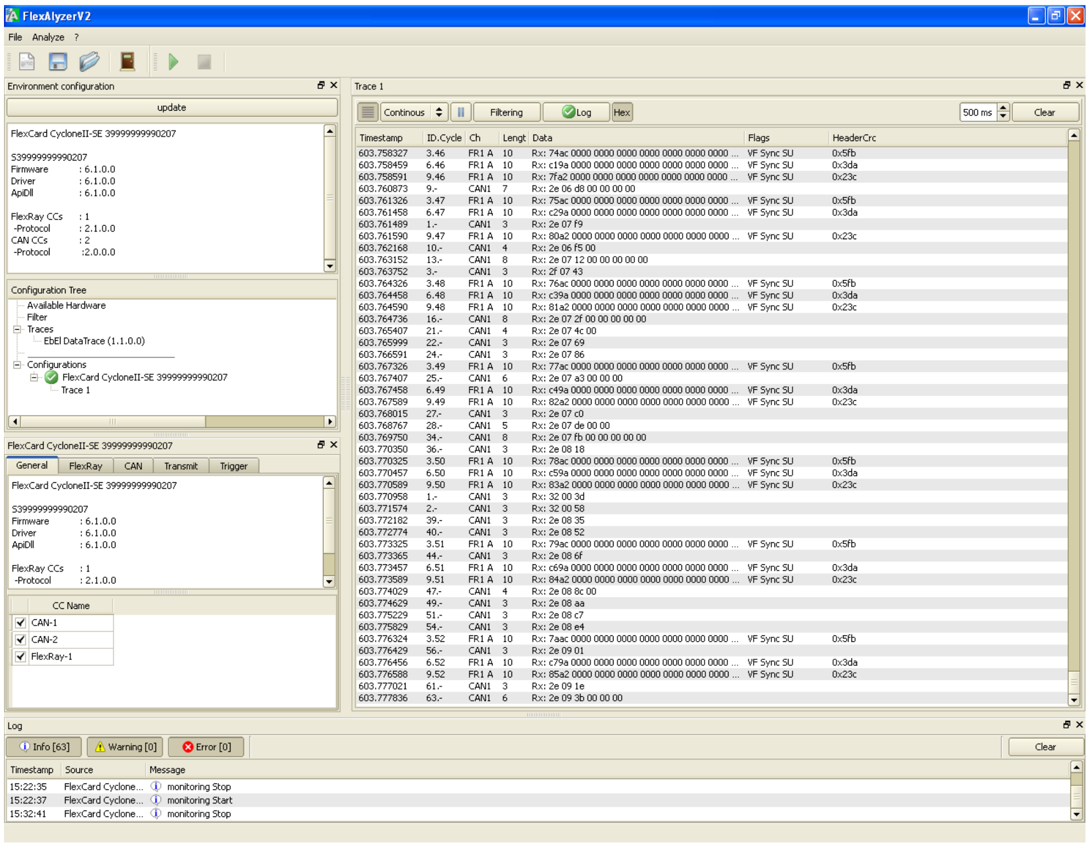
4.1.6 FlexAlyzer

FlexAlyzer is the software tool developed by Eberspacher Electronics and works in accordance with the FlexCard Cyclone II SE. It is used to monitor and analyze the message traffic in CAN and FlexRay networks which the FlexCard is connected.

The FlexAlyzer has many useful features such as displaying the data in decimal or hexadecimal format, filtering the content to show in the monitoring window according to various criteria. It also has an important feature of showing in the monitoring window the local time stamp for the receive time, ID, payload and other features of both FlexRay and CAN messages. In addition, FlexAlyzer is able to take log of the network traffic including all of the message contents and save as a *.txt file without any time limitation. This feature is also very important for time measurement experiments since the *.txt log file is parsed offline for all of the timing analysis. Fig. 4.5 shows a view form the FlexAlyzer user interface.

4.2 Experiment Setup

The experiment setup consists of SK-91465X-100PMC evaluation boards which are used as individual nodes composing the network, FlexCard Cyclone II SE network analyzer card and a PC with a PCMCIA slot. In order to construct the FlexRay and CAN networks, two PCBs busses are used. Each PCB bus have 9 D-Sub-9 male



```
FlexCard CycloneII-SE 39999999990207

S39999999990207
Firmware      : 6.1.0.0
Driver        : 6.1.0.0
ApiDll        : 6.1.0.0

FlexRay CCs   : 1
 -Protocol    : 2.1.0.0
CAN CCs       : 2
 -Protocol    :2.0.0.0
```

```
FlexCard CycloneII-SE 39999999990207

S39999999990207
Firmware      : 6.1.0.0
Driver        : 6.1.0.0
ApiDll        : 6.1.0.0

FlexRay CCs   : 1
 -Protocol    : 2.1.0.0
```

	CC Name
✓	CAN-1
✓	CAN-2
✓	FlexRay-1

```
Timestamp   ID.Cycle Ch    Lengt Data                                          Flags        HeaderCrc
603.758327  3.46     FR1 A 10    Rx: 74ac 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x5fb
603.758459  6.46     FR1 A 10    Rx: c19a 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x3da
603.758591  9.46     FR1 A 10    Rx: 7fa2 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x23c
603.760873  9.-      CAN1  7     Rx: 2e 06 d8 00 00 00 00
603.761326  3.47     FR1 A 10    Rx: 75ac 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x5fb
603.761458  6.47     FR1 A 10    Rx: c29a 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x3da
603.761489  1.-      CAN1  3     Rx: 2e 07 f9
603.761590  9.47     FR1 A 10    Rx: 80a2 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x23c
603.762168  10.-     CAN1  4     Rx: 2e 06 f5 00
603.763152  13.-     CAN1  8     Rx: 2e 07 12 00 00 00 00 00
603.763752  3.-      CAN1  3     Rx: 2f 07 43
603.764326  3.48     FR1 A 10    Rx: 76ac 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x5fb
603.764458  6.48     FR1 A 10    Rx: c39a 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x3da
603.764590  9.48     FR1 A 10    Rx: 81a2 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x23c
603.764736  16.-     CAN1  8     Rx: 2e 07 2f 00 00 00 00 00
603.765407  21.-     CAN1  4     Rx: 2e 07 4c 00
603.765999  22.-     CAN1  3     Rx: 2e 07 69
603.766591  24.-     CAN1  3     Rx: 2e 07 86
603.767326  3.49     FR1 A 10    Rx: 77ac 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x5fb
603.767407  25.-     CAN1  6     Rx: 2e 07 a3 00 00 00
603.767458  6.49     FR1 A 10    Rx: c49a 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x3da
603.767589  9.49     FR1 A 10    Rx: 82a2 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x23c
603.768015  27.-     CAN1  3     Rx: 2e 07 c0
603.768767  28.-     CAN1  5     Rx: 2e 07 de 00 00
603.769750  34.-     CAN1  8     Rx: 2e 07 fb 00 00 00 00 00
603.770350  36.-     CAN1  3     Rx: 2e 08 18
603.770325  3.50     FR1 A 10    Rx: 78ac 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x5fb
603.770457  6.50     FR1 A 10    Rx: c59a 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x3da
603.770589  9.50     FR1 A 10    Rx: 83a2 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x23c
603.770958  1.-      CAN1  3     Rx: 32 00 3d
603.771574  2.-      CAN1  3     Rx: 32 00 58
603.772182  39.-     CAN1  3     Rx: 2e 08 35
603.772774  40.-     CAN1  3     Rx: 2e 08 52
603.773325  3.51     FR1 A 10    Rx: 79ac 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x5fb
603.773365  44.-     CAN1  3     Rx: 2e 08 6f
603.773457  6.51     FR1 A 10    Rx: c69a 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x3da
603.773589  9.51     FR1 A 10    Rx: 84a2 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x23c
603.774029  47.-     CAN1  4     Rx: 2e 08 8c 00
603.774629  49.-     CAN1  3     Rx: 2e 08 aa
603.775229  51.-     CAN1  3     Rx: 2e 08 c7
603.775829  54.-     CAN1  3     Rx: 2e 08 e4
603.776324  3.52     FR1 A 10    Rx: 7aac 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x5fb
603.776429  56.-     CAN1  3     Rx: 2e 09 01
603.776456  6.52     FR1 A 10    Rx: c79a 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x3da
603.776588  9.52     FR1 A 10    Rx: 85a2 0000 0000 0000 0000 0000 0000 0000 ... VF Sync SU   0x23c
603.777021  61.-     CAN1  3     Rx: 2e 09 1e
603.777836  63.-     CAN1  6     Rx: 2e 09 3b 00 00 00
```

```
Timestamp  Source              Message
15:22:35   FlexCard Cyclone...  monitoring Stop
15:22:37   FlexCard Cyclone...  monitoring Start
15:32:41   FlexCard Cyclone...  monitoring Stop
```

Figure 4.5: FlexAlyzer User Interface

connectors mounted on it where all 9 pins of these connectors are connected to each other within the PCB. These PCBs therefore enable up to 9 distinct nodes to connect and communicate with each other. A view of the PCB busses used in the experiments is shown in Fig. 4.6.

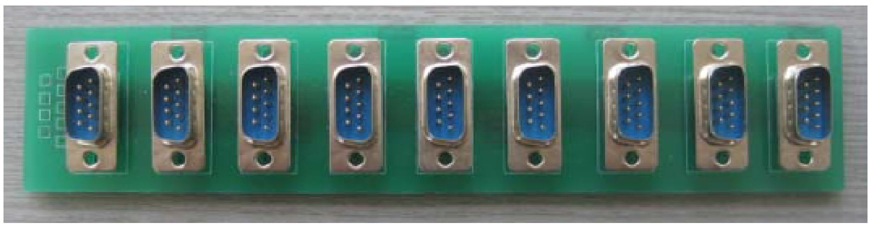


Figure 4.6: The PCB Bus Used for CAN Bus and FlexRay

The CAN bus and the FlexRay bus is constructed as a similar manner with the only difference that there are termination resistors on the CAN bus. The SK-91465X-100MPC boards can communicate through CAN bus up to 100 kbps data rate without termination. The FlexCard, however, is not able to communicate via CAN bus without termination even for data rates smaller than 100 kbps. Therefore, to properly establish the CAN communication for all bit rates, two 120 ohm termination resistors are welded on the two end connectors of the PCB bus between the live pins, which are CANH and CANL. A photograph of the setup can be seen in Fig. 4.7.

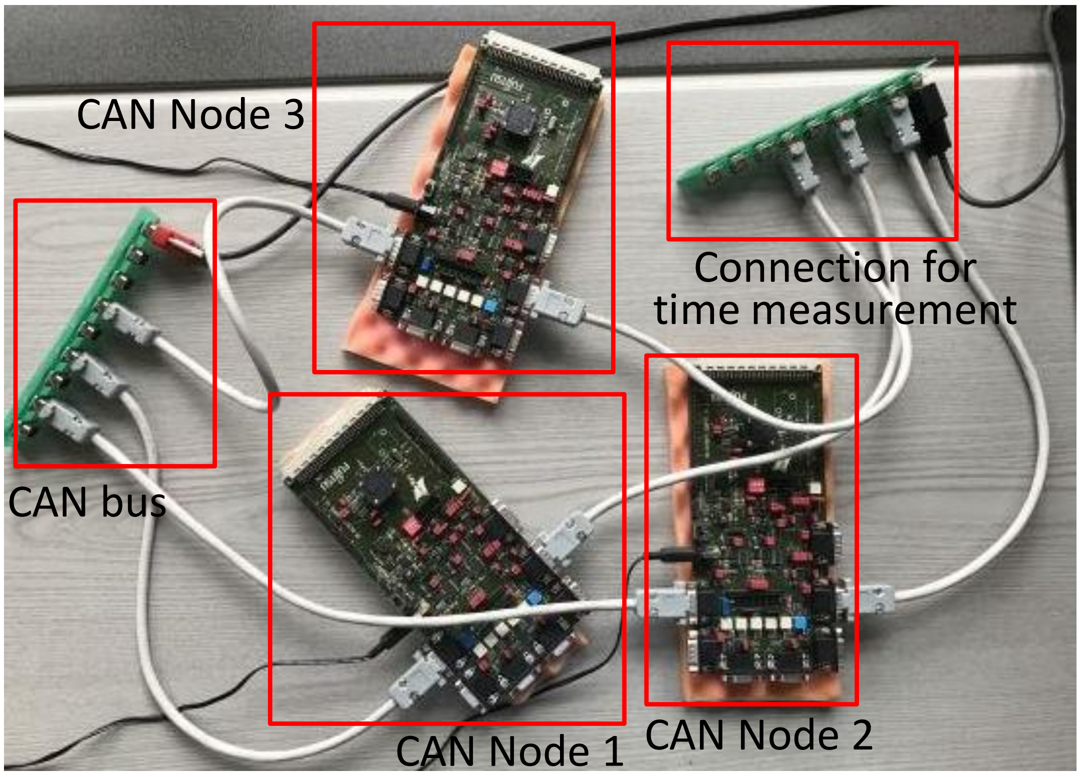


Figure 4.7: Experiment Setup Photograph

In order to connect the PCB busses with the SK-91465X-100MPC evaluation boards, cables which have 1-to-1 D-Sub-9 female connectors at both end are used.

FlexCard Cyclone II SE is used as a CAN node or a FlexRay node or both at the same time depending on the experiment. In the experiments, FlexCard is used as a silent node which does not send any message but receives and monitors the message traffic on the CAN and the FlexRay networks.

The last component of the experiment setup is a PC. FlexCard is connected to the PC through the PCMCIA slot and the FlexAlayzer is run on the PC for monitoring and logging the message traffic. The logged data is parsed offline for evaluation via various parsing programs we wrote in the PC. Lastly, the PC is also used to program the boards by using FME FR Flash Programmer described in Section 4.1.3.

4.3 Response Time Measurement Method

The verification of analytical results with the practical results requires correct time measurements. As stated in the preceding sections, a global clock is needed that is synchronous among all of the nodes in the network and this global clock is achieved using the FlexRay protocol. After all of the nodes are synchronized according to the FlexRay clock, the exchanged CAN messages can be time stamped properly. Lastly these time stamps can be used safely for evaluation purposes. The setup used for time measurement experiments is illustrated in Fig. 4.8.

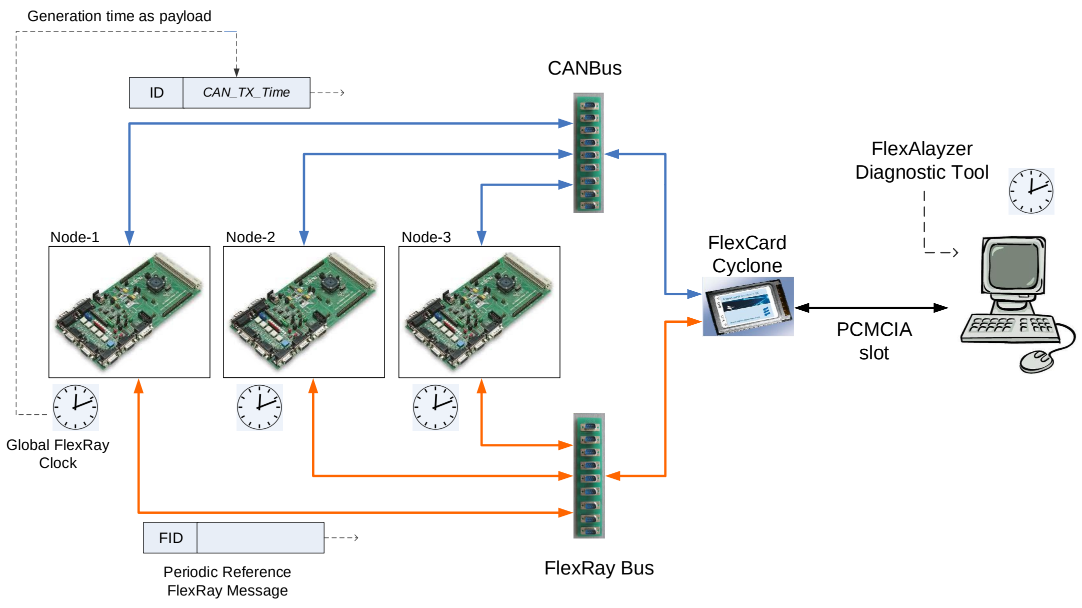
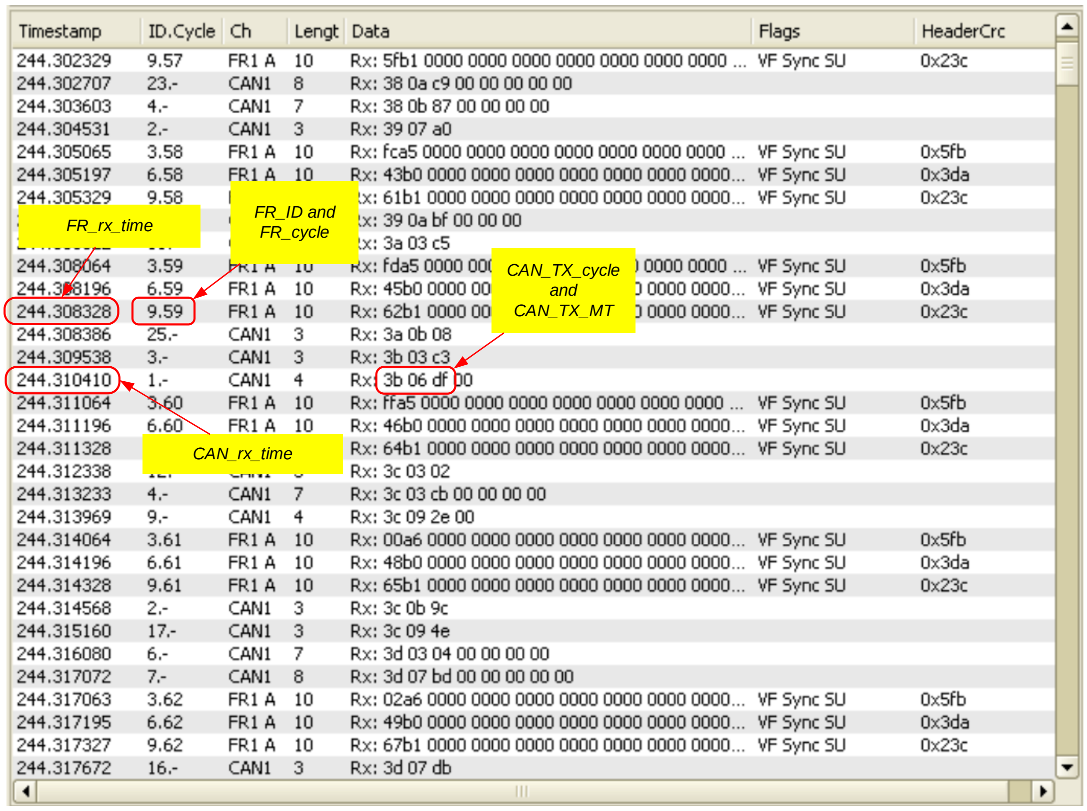


Figure 4.8: Experiment Setup Illustration

As shown in Fig. 4.8, the timestamp CANTX which represents the generation time of the message is obtained just before loading the payload to the output message buffer of the CAN controller. The timestamp is obtained via two different time service functions of the FlexRay Software driver. These functions return the network time in terms of the fundamental time units of the FlexRay network, namely the cycle number and the macrotick count. Since the cycle number is 8-bits and the macrotick count is 16-bits, each timestamp engraved in the packets is 3 bytes long. Therefore the minimum data size of the CAN messages used in the experiments is 3 bytes.

In order to calculate the response time of a message, we also have to know the reception time of the message. However, obtaining the reception time is not as straightforward as the generation time. To illustrate the situation, a view from the log file of an experiment exported by the FlexAlyzer software is given in Fig. 4.9.



```
Timestamp    ID.Cycle  Ch      Lengt  Data                                              Flags          HeaderCrc
244.302329   9.57      FR1 A   10     Rx: 5fb1 0000 0000 0000 0000 0000 0000 0000 ...   VF Sync SU     0x23c
244.302707   23.-      CAN1    8      Rx: 38 0a c9 00 00 00 00 00
244.303603   4.-       CAN1    7      Rx: 38 0b 87 00 00 00 00
244.304531   2.-       CAN1    3      Rx: 39 07 a0
244.305065   3.58      FR1 A   10     Rx: fca5 0000 0000 0000 0000 0000 0000 0000 ...   VF Sync SU     0x5fb
244.305197   6.58      FR1 A   10     Rx: 43b0 0000 0000 0000 0000 0000 0000 0000...    VF Sync SU     0x3da
244.305329   9.58      [illegible]    Rx: 61b1 0000 0000 0000 0000 0000 0000 0000...    VF Sync SU     0x23c
[illegible]                           Rx: 39 0a bf 00 00 00
[illegible]                           Rx: 3a 03 c5
244.308064   3.59      FR1 A   10     Rx: fda5 0000 00[illegible] 0000 0000 ...         VF Sync SU     0x5fb
244.308196   6.59      FR1 A   10     Rx: 45b0 0000 00[illegible] 0000 0000...          VF Sync SU     0x3da
244.308328   9.59      FR1 A   10     Rx: 62b1 0000 00[illegible] 0000 0000...          VF Sync SU     0x23c
244.308386   25.-      CAN1    3      Rx: 3a 0b 08
244.309538   3.-       CAN1    3      Rx: 3b 03 c3
244.310410   1.-       CAN1    4      Rx: 3b 06 df 00
244.311064   3.60      FR1 A   10     Rx: ffa5 0000 0000 0000 0000 0000 0000 0000 ...   VF Sync SU     0x5fb
244.311196   6.60      FR1 A   10     Rx: 46b0 0000 0000 0000 0000 0000 0000 0000...    VF Sync SU     0x3da
244.311328   [illegible]              Rx: 64b1 0000 0000 0000 0000 0000 0000 0000...    VF Sync SU     0x23c
244.312338   [illegible]              Rx: 3c 03 02
244.313233   4.-       CAN1    7      Rx: 3c 03 cb 00 00 00 00
244.313969   9.-       CAN1    4      Rx: 3c 09 2e 00
244.314064   3.61      FR1 A   10     Rx: 00a6 0000 0000 0000 0000 0000 0000 0000...    VF Sync SU     0x5fb
244.314196   6.61      FR1 A   10     Rx: 48b0 0000 0000 0000 0000 0000 0000 0000...    VF Sync SU     0x3da
244.314328   9.61      FR1 A   10     Rx: 65b1 0000 0000 0000 0000 0000 0000 0000...    VF Sync SU     0x23c
244.314568   2.-       CAN1    3      Rx: 3c 0b 9c
244.315160   17.-      CAN1    3      Rx: 3c 09 4e
244.316080   6.-       CAN1    7      Rx: 3d 03 04 00 00 00 00
244.317072   7.-       CAN1    8      Rx: 3d 07 bd 00 00 00 00 00
244.317063   3.62      FR1 A   10     Rx: 02a6 0000 0000 0000 0000 0000 0000 0000...    VF Sync SU     0x5fb
244.317195   6.62      FR1 A   10     Rx: 49b0 0000 0000 0000 0000 0000 0000 0000...    VF Sync SU     0x3da
244.317327   9.62      FR1 A   10     Rx: 67b1 0000 0000 0000 0000 0000 0000 0000...    VF Sync SU     0x23c
244.317672   16.-      CAN1    3      Rx: 3d 07 db
```

Figure 4.9: A View of the Log File

As can be seen from the Fig. 4.9, the timestamp that the FlexAlyzer includes in the beginning of every line in the log file (the leftmost colomn) is the local time of the FlexCard which indicates the amount of time that has passed since the beginning of the logging. Therefore, this local time is not useful alone. In order to make this information useful, a connection between the synchronous clock, which is the FlexRay clock, and the local FlexAlyzer clock has to be established. This is achieved as follows.

If we look at the log file we see that the FlexAlyzer logs the FlexRay messages including the ID of the message and the cycle number in which the message is transmitted. Since each message is allocated a slot in a cycle, the ID and the cycle number can be used to obtain the reception time of any FlexRay message in terms of the FlexRay network time units.

Secondly, we know that the reception time of each message is displayed by the Flex-

Alyzer in terms of its local clock. Therefore, the reception time of a CAN message can be obtained by choosing a reference FlexRay message and adding the local timestamp difference to the reception time of this reference FlexRay message. This method is formulated as

$$\begin{aligned}
&[(RefFR.Cycle - CANTX.Cycle)CycleLength + (RefFR.ID - 1)SSLength \\
&+ ActionOffset - CANTX.MT] + [CAN.rxtime - RefFR.rxtime] \\
&+ [RefFR.transmissiontime]
\end{aligned} \tag{4.1}$$

where

CANTX.Cycle and *CANTX.MT* are the timestamps indicating the generation time of the CAN message in terms of FlexRay cycle number and macrotick count, respectively.

RefFR.Cycle and *RefFR.ID* are the cycle number and frame ID of the reference FlexRay message as logged by the FlexAlyzer.

CycleLength, *SSLength* and *ActionOffset* are the macrotick correspondences of the FlexRay network parameters the Cycle Length, the Static Slot Length and the Action Offset Length, respectively.

CAN.rxtime and *RefFR.rxtime* are the local receive times of the CAN message and the reference FlexRay message tagged by the FlexAlyzer.

RefFR.transmissiontime is the transmission time of the reference FlexRay message. Since the bus speed and the FlexRay message length is known, this value can be determined.

The visual illustration of the measurement method as a timeline is provided in Fig. 4.10.

In Fig. 4.10, SS-n shows the static slot number in a FlexRay communication cycle. The FlexRay is configured such that the static slot length is 40 macroticks and the cycle length is 3000 macroticks. The duration of a macrotick is 1 μs. Since the cycle

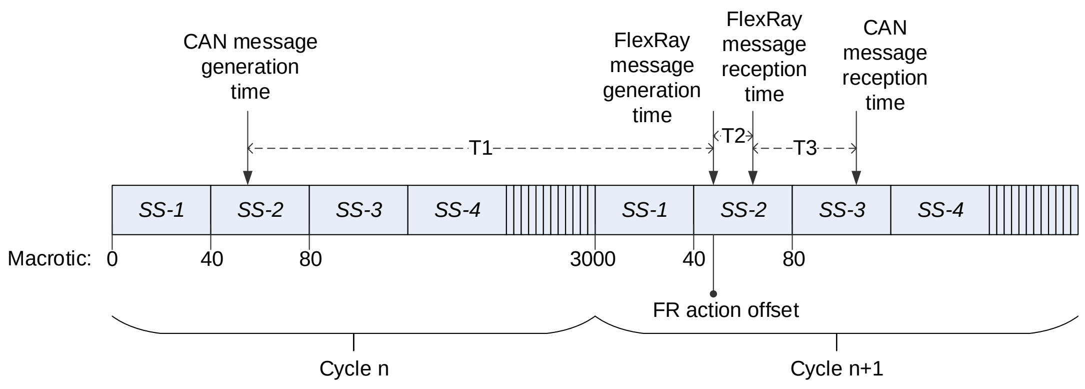

CAN message generation time
FlexRay message generation time
FlexRay message reception time
CAN message reception time
T1
T2
T3
SS-1
SS-2
SS-3
SS-4
SS-1
SS-2
SS-3
SS-4
Macrotic: 0
40
80
3000
40
80
FR action offset
Cycle n
Cycle n+1

Figure 4.10: A View of the Log File

number can go up to 64, response times up to 64 x 3 ms = 192 ms can be measured with a precision of 1 μs.

It can be seen on Fig. 4.10 that the overall response time (from CAN message generation time to CAN message reception time) is divided into three periods for better understanding, namely T1, T2, and T3. These three periods have four border times.

CAN message generation time is the time obtained in terms of FlexRay cycle number (*CANTX.Cycle*) and the macrotick count (*CANTX.MT*) just before queuing the CAN message in the buffer.

FlexRay message generation time is obtained using the frame ID (*RefFR.ID*) and cycle (*RefFR.Cycle*) values of the reference message as logged by the FlexAlyzer. The frame ID of the message gives information about the static slot number in which the message is generated. Together with the action offset and the cycle number, the exact generation time of the reference message is easily determined in terms of FlexRay cycle number and macrotick count.

FlexRay message reception time is the local time logged by the FlexAlyzer software (*RefFR.rxtime*) indicating the time when the reference FlexRay message is received by the FlexCard.

CAN message reception time is the local time logged by the FlexAlyzer software (*CAN.rxtime*) indicating the time when the CAN message is received by the FlexCard.

The correspondance of the three periods with (4.1) is as follows.

$$\begin{aligned}T1 =&(RefFR.Cycle - CANTX.Cycle)CycleLength\\&+ (RefFR.ID - 1)SSLength + ActionOffset - CANTX.MT\\T2 =&RefFR.transmissiontime\\T3 =&CAN.rxtime - RefFR.rxtime\end{aligned}$$

4.4 Hardware Measurements

In order to verify the theoretical results for the WCRT analysis by hardware experiments, a realistic message set is constructed. In the experiments, three CAN nodes are used with a data rate of 125 kbit/s. The message set used for the experiment can be seen in Table 4.1.

For this message set, offset assignment is done according to two different offset scheduling algorithms: the existing LD algorithm and the proposed MBD algorithm. The tests performed for this message set are;

- Offset scheduling using LD
- Offset scheduling using MBD
- Without offsets

The first aim of these tests is to show that offset scheduling decreases the WCRT of the messages considerably. The second aim is to verify that the proposed MBD algorithm outperform the existing LD algorithm in consistence with the computational results obtained in Section 3.2.

Each test is run for 20 minutes and the log file obtained for each test is parsed offline using a parsing program written in C.

Before the experiments are held, the computational WCRT values are computed according to the methods existing in the literature both for offset scheduling and without scheduling. These methods were referenced in Section 2.3. The computational WCRT values for the three different offset cases is shown in Fig. 4.11.

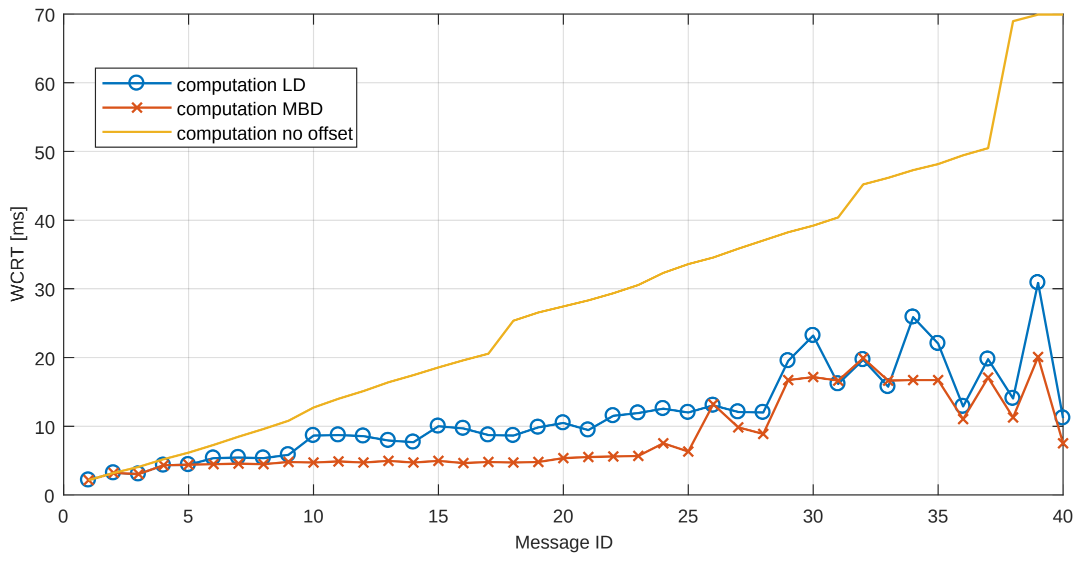


Figure 4.11: Comparison of Computational WCRT

As can be seen in Fig. 4.11, the proposed MBD algorithm results in smaller WCRTs compared to the existing LD algorithm. Moreover, it is obvious that WCRT values with offset scheduling are much smaller compared to the case with no offsets. The test results obtained for the same cases are shown in Fig. 4.12.

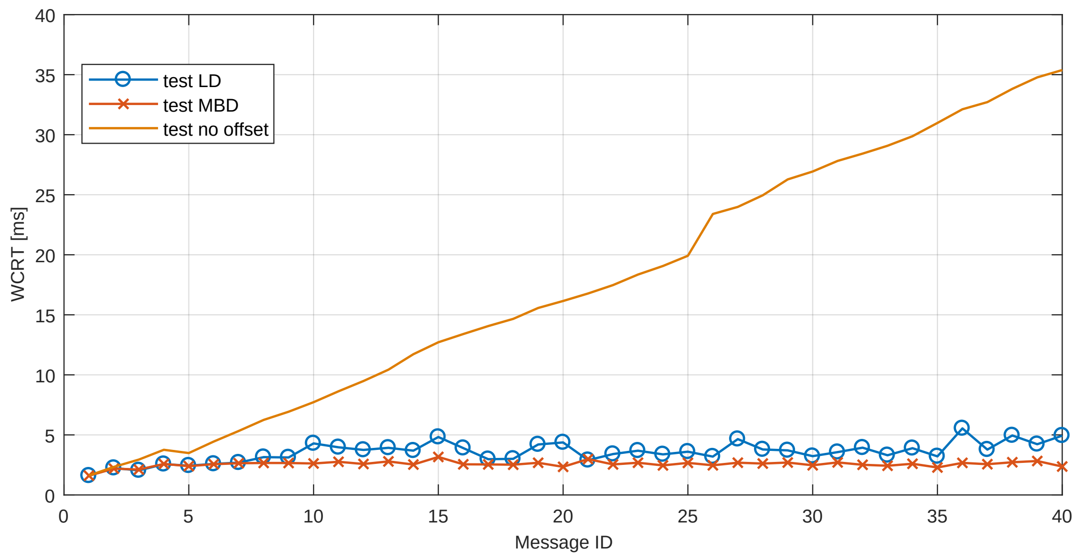


Figure 4.12: Comparison of WCRT Measurements

The results in Fig. 4.12 validate that offset scheduling improves the response time considerably. Moreover, it can also be seen that the WCRT values for offset scheduling by MBD algorithm are smaller than those obtained by LD algorithm, which verifies the computational results.

The comparisons of computational results with the measurements are provided for each offset scheduling case in Fig. 4.13, Fig. 4.14, and Fig. 4.15

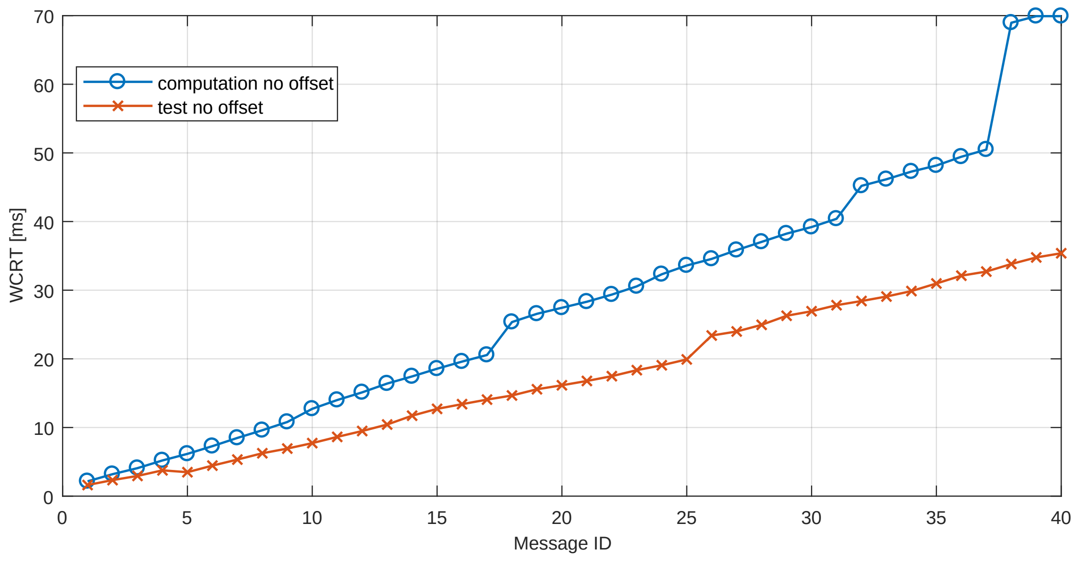


Figure 4.13: WCRT Comparison for No Offset

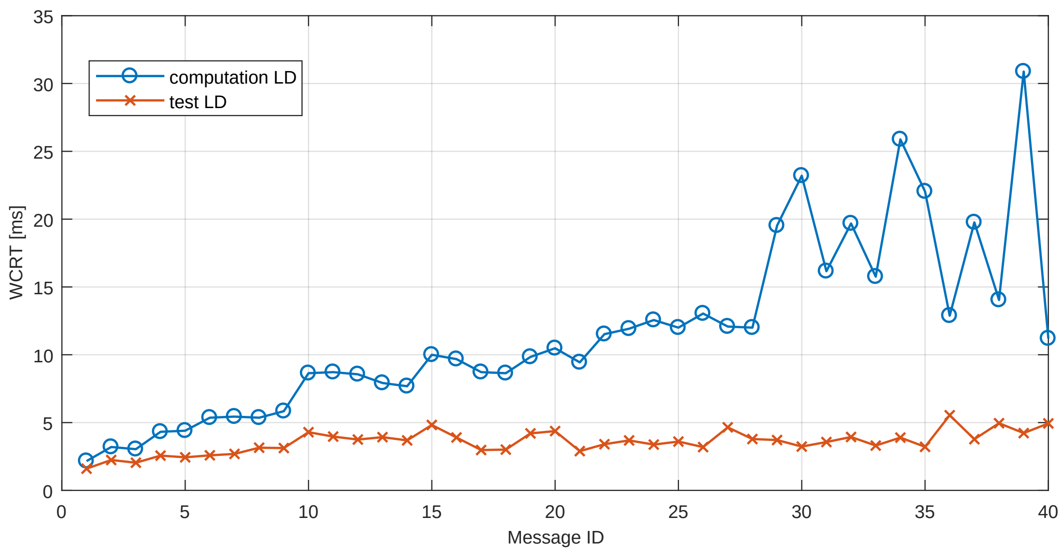


Figure 4.14: WCRT Comparison for Offset Scheduling by LD

For all three cases it is observed that the measured WCRTs are considerably smaller than the computational ones. The computations are based on the so-called "critical instant" worst case scenario where phasings between the nodes are such that the interference from the other nodes is maximum. However, these computations are

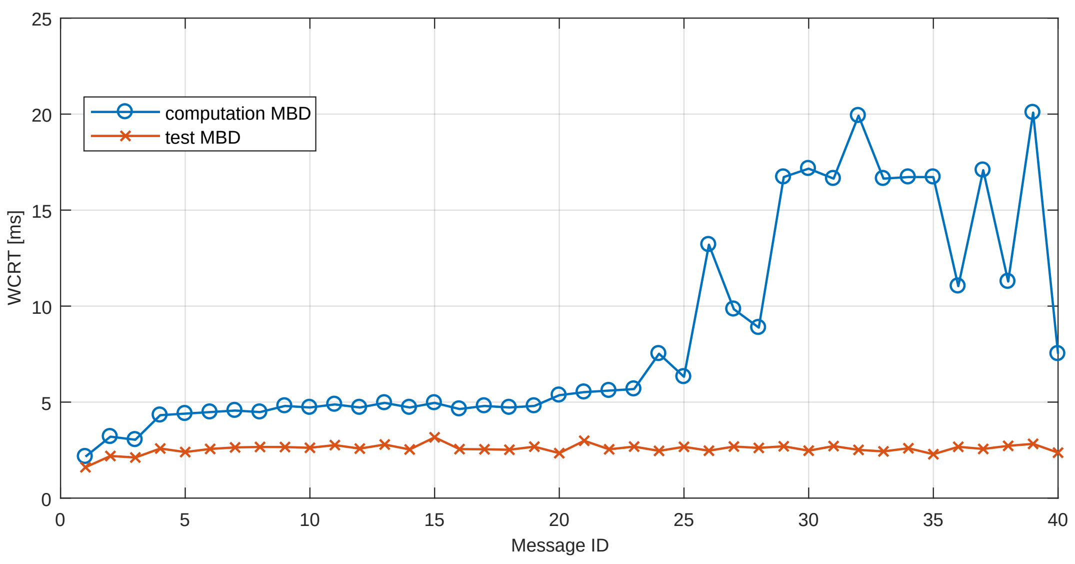


Figure 4.15: WCRT Comparison for Offset Scheduling by MBD

very restrictive, as they result in very high calculated WCRTs which could occur in theory, but with little probability in practice. In fact, the probability of missing a deadline could be as small as the probability of hardware failure [7]. However, there are many applications which are not time-critical and can tolerate a certain failure rate. Therefore a probabilistic response time analysis should be used along with the WCRT analysis, which leads us to the second contribution of the thesis.

Table4.1: Message Properties

M	N_M	P_M	T_M	L_M	$O_M(LD)$	$O_M(MBD)$
M_1	N_3	1	10 ms	3 B	4	4
M_2	N_1	2	20 ms	5 B	9	9
M_3	N_1	3	20 ms	3 B	14	19
M_4	N_2	4	20 ms	6 B	9	9
M_5	N_3	5	50 ms	4 B	9	9
M_6	N_1	6	50 ms	6 B	21	4
M_7	N_1	7	50 ms	7 B	41	14
M_8	N_1	8	50 ms	6 B	4	24
M_9	N_3	9	50 ms	7 B	19	19
M_{10}	N_2	10	50 ms	5 B	19	4
M_{11}	N_3	11	50 ms	8 B	29	29
M_{12}	N_3	12	50 ms	6 B	39	39
M_{13}	N_1	13	50 ms	8 B	25	34
M_{14}	N_1	14	50 ms	5 B	45	45
M_{15}	N_2	15	50 ms	6 B	39	14
M_{16}	N_3	16	100 ms	5 B	49	49
M_{17}	N_2	17	100 ms	4 B	59	39
M_{18}	N_2	18	100 ms	3 B	79	79
M_{19}	N_3	19	100 ms	7 B	97	99
M_{20}	N_3	20	100 ms	3 B	1	2
M_{21}	N_2	21	100 ms	3 B	95	97
M_{22}	N_3	22	100 ms	5 B	6	6
M_{23}	N_3	23	100 ms	7 B	11	12
M_{24}	N_2	24	100 ms	3 B	4	21
M_{25}	N_3	25	100 ms	8 B	16	16
M_{26}	N_1	26	100 ms	4 B	61	1
M_{27}	N_3	27	100 ms	8 B	21	22
M_{28}	N_3	28	100 ms	7 B	26	26
M_{29}	N_1	29	100 ms	7 B	82	17
M_{30}	N_1	30	100 ms	4 B	65	21
M_{31}	N_1	31	100 ms	7 B	17	37
M_{32}	N_2	32	100 ms	3 B	14	34
M_{33}	N_1	33	100 ms	4 B	37	41
M_{34}	N_1	34	100 ms	6 B	57	57
M_{35}	N_1	35	100 ms	3 B	78	61
M_{36}	N_3	36	100 ms	8 B	31	32
M_{37}	N_1	37	100 ms	5 B	85	77
M_{38}	N_3	38	100 ms	8 B	36	36
M_{39}	N_2	39	100 ms	7 B	24	44
M_{40}	N_3	40	100 ms	4 B	41	42

CHAPTER 5

RESPONSE TIME DISTRIBUTION

The experimental findings in the preceding chapter revealed that the computational WCRT is not likely to happen in practice. For applications where hard deadline constraints exist, the network design can be implemented according to the theoretical WCRT values. However most of the applications do not have strict deadlines and scheduling the messages based on the WCRT values results in degraded utilization of the network. In such cases, knowing the probability that a message experiences a certain response time plays a crucial role. Moreover, the response time distribution is also useful for applications with hard deadline constraints where a certain failure rate is acceptable. In such applications the probabilistic response time analysis allows the designer to make a trade-off between reliability and timeliness [12], [20].

5.1 Definition

In order to obtain the probabilistic response time distribution, the aspects that affect the response time and cause variations have to be analyzed. For a given message set, there are two factors which are not constant during the run-time of the network and thus affect the response time of a CAN message:

- The message length is not constant since stuff-bits are inserted depending on the message content
- The phases between nodes change because CAN nodes are not synchronized

5.1.1 Stuff-Bits

Stuff-bits are additional bits added to the message by the CAN protocol. As six consecutive bits of the same polarity (111111 of 000000) are used to signal errors, the CAN protocol has a built in mechanism that removes these forbidden bit-patterns by inserting stuff-bits at specific positions. The worst-case scenario for bit stuffing is depicted in Fig. 5.1.

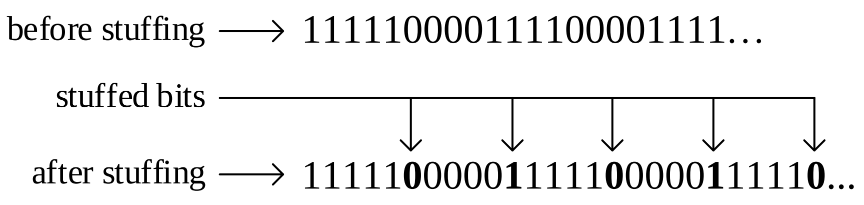


Figure 5.1: Worst-Case Bit Stuffing Scenario

This mechanism results in a variation in the length and thus the duration of a CAN message. The WCRT is computed by assuming the worst-case bit-stuffing condition. However, when evaluating the probabilistic response time distribution it is necessary to use the message duration distributions depending on bit-stuffing instead of the worst-case stuffed frame duration [13], [12].

The maximum duration of a CAN frame after bit-stuffing is given by [4]

$$C = (g + 8b + 13 + \lfloor \frac{g + 8b - 1}{4} \rfloor)\tau_{\text{bit}} \tag{5.1}$$

where $g = 34$ for standard format (11-bit identifier) and $g = 54$ for the extended format (29-bit identifier). b is the number of data bytes in the payload. The part of the function that uses the floor operator determines the number of worst-case stuff bits depending on the message payload.

The probability distribution of a certain number of stuff bits can be calculated by assuming equal probability of bit-value 0 and 1 among the bits that are exposed to bit-stuffing in a CAN frame [13]. However, this is an exhaustive work and is not in the scope of this thesis. In the computational analysis, we use the data provided in [13] which gives the probabilities for different frame sizes (number of bits).

5.1.2 Node Phases

As stated in the preceding sections, CAN is an asycnhronous bus, where there is no global clock mechanism. Each node has its own clock and queues its messages according to this clock. As a consequence, the contention among messages from different nodes is not deterministic. This results in response time variations in CAN messages since the relative generation times of messages vary. Fig. 5.2 illustrates the impact of different phases between two nodes. Here, node N_1 and N_2 generate messages M_1 and M_2, respectively.

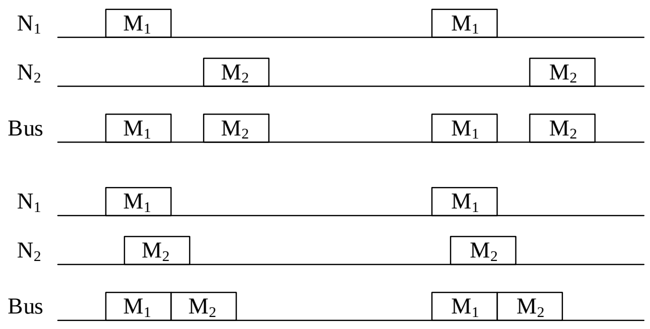


Figure 5.2: Impact of Phasing between Two Nodes

In the first case shown at the top of Fig. 5.2, the messages from the two nodes do not interfere with each other such that each message has immediate acces to the CAN bus. In the second case shown at the bottom, message M_2 from node N_2 is blocked by message M_1 from N_1, leading to a larger response time for M_2.

The WCRT is computed according to a so-called *critical instant* where the phases between all nodes are such that the interference of messages from other nodes on the target message is maximum. It is important to note that a single critical instant (worst-case node phases) that produces the WCRT can be computed with a straight-forward recurrence relation [4]. This computation does not need any information about potentially shorter response times obtained for different node phases.

The main objective of this chapter is computing the response time distribution for all messages. To this end, we will first compute the response time distribution for each phasing scenario depending on bit stuffing. Finally, we will obtain the overall response time distribution by averaging over all phasing scenarios by assuming that each scenario occurs with equal probability.

For a single phasing scenario, the computation of the response time distribution for a given target message is accomplished by the following steps:

- Transforming the message streams from different nodes into a single stream of messages
- Obtaining the message set that interferes with the target message and thus contributes to the response time distribution of the target message
- Computing the backlog distribution that is generated by the interfering message set
- Computing the response time distribution using the backlog distribution and the transmission time distribution of the target message

Note that the most important point of these steps is the computation of the *backlog computation* which is one of the main contributions of the thesis and has not been considered in the existing literature.

5.2 Analytical Evaluation of the Response Time Distribution

This section formalizes the described concept of the response time distribution for CAN messages and develops an original method for its computation.

5.2.1 System Model and Notation

We first introduce the required notation and the system model. As in Chapter 3, each message M_i is characterized by its period T_i, priority P_i, deadline D_i, offset O_i and transmitting node N_k. Differently, the execution time of M_i is now a discrete random variable denoted as $\mathcal{C}_i$ with a known probability mass function (*pmf*) $f_{\mathcal{C}_i} : \mathbb{N} \rightarrow [0, 1]$. $f_{\mathcal{C}_i}(c)$ defines the probability of an execution time c. Defining $C_{i,\text{min}} := \min_c\{f_{\mathcal{C}_i}(c) \neq 0\}$ and $C_{i,\text{max}} := \max\{f_{\mathcal{C}_i}(c) \neq 0\}$, a non-zero probability is obtained for each value $c \in [C_{i,\text{min}}, C_{i,\text{max}}]$.

Note that since the message execution time and node phase shift variation granularity is τ_{bit}, all probability distributions including execution times, response times and backlogs are expressed as discrete-time values that are multiples of τ_{bit}.

For each periodic activation of message M_i of some node N_k, we consider a message instance where the j-th instance of message M_i is denoted as $M_{i,j}$ with release time or start time expressed as $S_{i,j} = O_i + (j-1) \cdot T_i$ relative to the clock of N_k.

For each node N_k we define its hyperperiod H_k, as the least common multiple of the periods of its messages.

We denote $\Phi_{k,l}$ as the phase shift of node N_k relative to a remote node N_l. We assume that all nodes boot up at arbitrary times. Hence the probability function of $\Phi_{k,l}$ is uniformly distributed within the system hyperperiod with a granularity of τ_{bit} (bit time).

We further denote release times of message M_i belonging to the node N_k relative to the clock of a remote node N_l as

$$\lambda_{i,j} = \Phi_{k,l} + O_i + (j-1) \cdot T_i. \tag{5.2}$$

The response time for message M_i is a random variable denoted as $\mathcal{R}_i$ with *pmf* $f_{\mathcal{R}_i}(r) = \mathbb{P}\{\mathcal{R}_i = r\}$ and hence describes the probability that message M_i has the response time r.

5.2.2 Backlog Explanation

Definition: We define the *backlog* for message M_i as the sum of all execution times not yet serviced and hence causing a delay for the message M_i.

The components of the backlog are higher priority messages that are queued before or at the same time as M_i and interfere with M_i. Since CAN is a non-preemptive bus, when a low priority message starts transmission it cannot be interrupted by a higher priority message. Therefore, a *blocking* lower priority message can also contribute to the backlog. The backlog for message M_i is a random variable $\mathcal{B}_i$ with *pmf* $f_{\mathcal{B}_i}(b) = \mathbb{P}\{\mathcal{B}_i = b\}$.

As an example, consider the message set given in Table 5.1.

Table5.1: Message Properties

M_i	N_k	P_i	T_i	O_i	$f_{\mathcal{C}_i}$
M_1	N_1	1	25	0	$[5,6,7,8] \rightarrow [0.2,0.3,0.3,0.2]$
M_2	N_1	2	25	11	$[4,5,6] \rightarrow [0.2,0.4,0.4]$
M_3	N_2	3	25	0	$[4,5,6] \rightarrow [0.2,0.5,0.3]$

Assume that the phase shift of node N_2 relative to the node N_1 is $\Phi_{2,1} = 3$. In this case the queuing times of messages are $\lambda_1 = 0$, $\lambda_2 = 11$, and $\lambda_3 = 3$. Since the periods of the messages are equal to the hyperperiod $H = 25$, only one instance from each message is queued within a hyperperiod H. The positioning of the messages for a single hyperperiod is shown in Fig. 5.3.

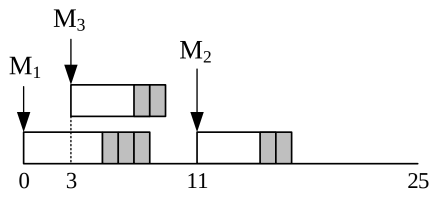


Figure 5.3: Initial View of the System

In Fig. 5.3, the gray boxes represent the variable part of the message with each box denoting a single stuff-bit. Assume that we want to compute the response time *pmf* for the message M_2 for this case. First we need to compute the backlog *pmf* for M_2. It is obvious that messages M_1 and M_3 starts transmission before M_2. In order to find out whether these messages interfere with M_2 and affect the response time of M_2, we first look at the worst case where M_1 and M_3 have the largest execution times. From Table 5.1 $C_{1,\text{max}} = 8$ and $C_{3,\text{max}} = 6$. In the worst case, M_1 and M_3 will finish their execution at time $t = 0 + 8 + 6 = 14$, which is after the release time of M_2. We conclude that these two messages create a probabilistic backlog for M_2 which needs to be calculated in order to obtain the response time *pmf* of M_2.

In order to find the backlog resulting from M_1 and M_3, we use the discrete-time *convolution* of two pmfs f and g as follows:

$$\texttt{conv}(f,g) = \sum_{i=-\infty}^{\infty} f(i)g(x-i). \tag{5.3}$$

where $\texttt{conv}()$ denotes the discrete-time convolution operator.

Proposition: When a new message M_j is to be added to an existing backlog and if the release time of the message is less than or equal to the minimum end time of the backlog, the resulting backlog *pmf* $f_{\hat{\mathcal{B}}_i}$ is the discrete-time convolution of the existing backlog *pmf* $f_{\mathcal{B}_i}$ and the execution time *pmf* $f_{\mathcal{C}_j}$ of the message M_j:

$$f_{\hat{\mathcal{B}}_i} = \texttt{conv}(f_{\mathcal{B}_i}, f_{\mathcal{C}_j}) \tag{5.4}$$

Therefore, the backlog *pmf* resulting from M_1 and M_3 is

$$f_{\mathcal{B}_2} = \texttt{conv}(f_{\mathcal{C}_1}, f_{\mathcal{C}_3}) = [9, 10, 11, 12, 13, 14] \to [0.04, 0.14, 0.26, 0.28, 0.2, 0.08]$$

The view of the system with the computed backlog is depicted in Fig. 5.4.

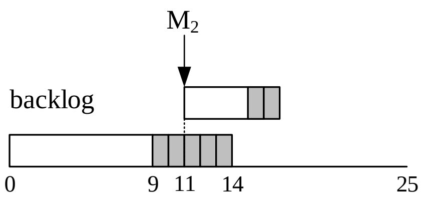


Figure 5.4: View of the System with the Computed Backlog

Since there are no more messages that will be added to the backlog, the response time *pmf* of the target message M_2 can be calculated. However, as can be seen from Fig. 5.4, the backlog starts from $t = 0$, whereas the release time of M_2 is $t = 11$. In order to calculate the backlog at $t = 11$, we use the *shrinking* operation [7].

Proposition: Given a backlog *pmf* starting from t, the backlog *pmf* at a later time t' can be calculated by shifting the *pmf* to the left by $t' - t$ time units and accumulating all the values for $b < 0$ at the origin (zero backlog) after the shift. This manipulation is called the shrinking operation and it represents progression in time. The idea behind accumulating the negative values in the origin is that all negative backlog values are seen as zero backlog. Formally, it holds that

$$\texttt{shrink}(f_{\mathcal{B}}(b), t') = \begin{cases} 0, & \text{for } b < 0 \\ \sum_{i=0}^{t'-t} f_{\mathcal{B}}(i), & \text{for } b = 0 \\ f_{\mathcal{B}}(b + t' - t), & \text{for } b > 0 \end{cases} \tag{5.5}$$

where $\texttt{shrink}()$ denotes the shrinking operator.

Using the shrinking operator, the backlog for M_2 at $t = 11$ is then calculated by

$f_{\hat{\mathcal{B}}_2} = \texttt{shrink}(f_{\mathcal{B}_2}, 11) = [0, 1, 2, 3] \rightarrow [0.44, 0.28, 0.2, 0.08]$.

Fig. 5.5 depicts the situation after shrinking.

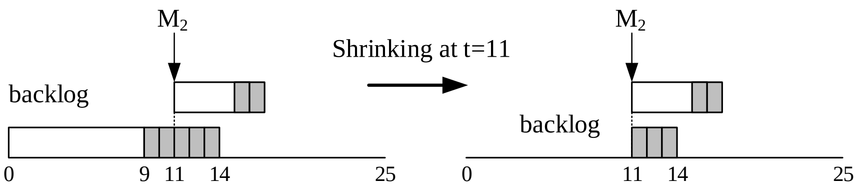


Figure 5.5: View of the System after Shrinking

As soon as the backlog at the message release instant is computed, the response time *pmf* can easily be calculated using the convolution operation as

$$f_{\mathcal{R}_i} = \texttt{conv}(f_{\mathcal{B}_i}, f_{\mathcal{C}_i}) \tag{5.6}$$

Using (5.6), the response time *pmf* of M_2 is calculated as

$f_{\mathcal{R}_2} = \texttt{conv}(f_{\mathcal{B}_2}, f_{\mathcal{C}_2}) = [4, 5, 6, 7, 8, 9] \rightarrow [0.088, 0.232, 0.328, 0.208, 0.112, 0.032]$.

That is, in this example and with the given node phases, message M_2 has a minimum response time of 4 with a probability of 0.088 and a maximum response time of 9 with a probability of 0.032. Note that the distribution in this case only depends on the execution time distributions of the messages. Using an analogous computation, the response time distributions for M_2 can be computed for all possible node phases.

5.2.3 Backlog Computation

As stated previously, backlog computation is the most important task for determining the response time *pmf*. If the backlog distribution *pmf* is obtained, the response time distribution *pmf* can be computed using (5.6). In this section, the steps required to compute the backlog *pmf* for a target message M_i will be described. We assume that the message stream that iteratively generates the backlog for the message M_i is already given. This message stream will be denoted as $\mathbb{S}$. It contains information about the successive release times $\lambda_{i,j}$ of all messages M_i.

The backlog distribution computation is an iterative process which starts with the first message (message with the highest priority at the smallest release time) in $\mathbb{S}$ and progresses with addition of the highest priority message of the messages that are queued just before the backlog ends. This process iterates until there are no more messages to add, or the target message is the message that gains access to the bus. In this case the backlog computation terminates.

In the example in the previous section, we show that when adding a new message to an existing backlog, the resultant backlog *pmf* is obtained by using convolution. However, this is only valid if the release time of the new message is less than or equal to the minimum end time of the existing backlog. The other condition leads to different cases which need to be evaluated separately.

As an illustration, consider the message queuing case shown in Fig. 5.6 with execution time *pmf*s in Table 5.2.

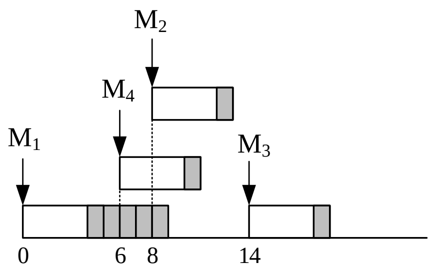


Figure 5.6: Example Message Queuing

Table5.2: Message Properties

M_i	P_i	f_{C_i}
M_1	1	$[4,5,6,7,8,9] \rightarrow [0.1,0.2,0.2,0.2,0.2,0.1]$
M_2	2	$[4,5] \rightarrow [0.5,0.5]$
M_3	3	$[4,5] \rightarrow [0.4,0.6]$
M_4	4	$[4,5] \rightarrow [0.7,0.3]$

Suppose that we want to compute the backlog for the message M_3. Since the stream starts with M_1, the initial value of the backlog is the execution time *pmf* of M_1. Later, M_2 and M_4 will join the backlog. However, as can be seen, messages M_2 and M_4 are released at the variable portion of the message M_1. Due to the priority-based non-preemptive nature of CAN, the transmission order of the messages varies depending on the possible end times of the backlog. For instance, if the message M_1 finishes its execution earlier than $t = 8$, M_4 will be transmitted after M_1. However if M_1 finishes at or later than $t = 8$, the higher priority message M_2 will be transmitted before M_4.

In order to handle such different conditions, we divide the backlog into sub-portions where each part ends before the next critical instant when at least one message is queued. We treat each portion as a separate backlog and make the computations for all possible backlog portions. The *pmf* of the resultant backlogs is defined as:

Definition: We define the *partial backlog* $f_{\mathcal{B}}[b_s, b_e]$ as the part of the backlog $f_{\mathcal{B}}$ that only takes the values between b_s and b_e. Formally the *pmf* of the partial backlog *pmf* is expressed as,

$$f_{\mathcal{B}}[b_s, b_e] = \begin{cases} f_{\mathcal{B}}, & \text{for } b_s \leq b \leq b_e \\ 0, & \text{otherwise} \end{cases} \tag{5.7}$$

Consider $f_{\mathcal{B}} : [5, 6, 7, 8] \rightarrow [0.2, 0.3, 0, 3, 0, 2]$. Then, $f_{\mathcal{B}}[6, 7] = [6, 7] \rightarrow [0.3, 0.3]$.

There are two critical instants $t = 6$ and $t = 8$ in the variable portion of the backlog caused by M_1. This leads to three possible partial backlogs as shown in Fig. 5.7 with *pmf*s $f_{\mathcal{B},1} = [4, 5] \rightarrow [0.1, 0.2]$, $f_{\mathcal{B},2} = [6, 7] \rightarrow [0.2, 0.2]$ and $f_{\mathcal{B},3} = [8, 9] \rightarrow [0.2, 0.1]$.

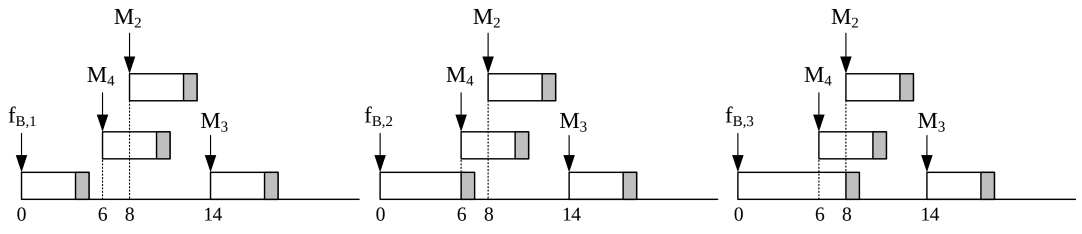


Figure 5.7: Partial Backlogs of the Backlog due to M_1

The backlog is computed progressively starting from each partial backlog and adding a new message to the backlog which is the highest priority of the messages released between the beginning and ending times of the backlog. If we look at the first partial backlog in Fig. 5.7, we see that there is no such message. Thus, this partial backlog is observed as zero backlog to the next message M_3. Therefore this probability information should be transferred to the beginning of the next partial backlog which is $t = 6$ which is also observed as zero backlog by M_3. At this time we need a new definition.

Definition: We define the *partially-accumulated backlog* $\texttt{accum}(f_{\mathcal{B}}[b_{\mathrm{s}}, b_{\mathrm{e}}], b_{\mathrm{e}} + 1)$ as the modified backlog where the probability at $b_{\mathrm{e}} + 1$ is replaced by the sum of probability values of *pmf* $f_{\mathcal{B}}$ from $b = b_{\mathrm{s}}$ to $b = b_{\mathrm{e}} + 1$. The resulting *pmf* is expressed as,

$$\texttt{accum}(f_{\mathcal{B}}[b_{\mathrm{s}}, b_{\mathrm{e}}], b_{\mathrm{e}} + 1)(b) = \begin{cases} 0 & \text{for } b < b_{\mathrm{e}} + 1 \\ \sum_{i=b_{\mathrm{s}}}^{b_{\mathrm{e}}+1} f_{\mathcal{B}}(i) & \text{for } b = b_{\mathrm{e}} + 1 \\ f_{\mathcal{B}}(b) & \text{otherwise} \end{cases} \tag{5.8}$$

Consider $f_{\mathcal{B}} : [5, 6, 7, 8] \to [0.2, 0.3, 0, 3, 0, 2]$. Then, $\texttt{accum}(f_{\mathcal{B}}[5, 6], 7) = [5, 6, 7, 8] \to [0, 0, 0.8, 0.2]$.

The partially-accumulated backlog is used for portions of the backlog that result in an empty bus. This is the case for the first partial backlog in Fig. 5.7. Since no new message is generated before time 6, a backlog of 6 is observed with probability $f_{\mathcal{B}}(4) + f_{\mathcal{B}}(5) + f_{\mathcal{B}}(6) = \texttt{accum}(f_{\mathcal{B}}[4, 5], 6)(6) = 0.1 + 0.2 + 0.2 = 0.5$.

We are now ready to state our original algorithm for the computation of the backlog distribution in Algorithm 7.

The algorithm is a recursive algorithm that calls itself as long as a new message is added to the backlog stream. Its arguments are the overall sum of backlog distributions $f_{\mathcal{B}}$ which accumulates the resultant backlogs at each condition, the local backlog distribution before division into partial backlogs $f_{\hat{\mathcal{B}}}$, the start time of the current backlog t_0, the message stream $\mathbb{S}$, a priority queue of waiting messages $\mathbb{W}$ (message generation time is greater than or equal to t_0 and less than or equal to the current backlog end time) and a done list of processed messages $\mathcal{D}$. In each function call, the highest-priority message in $\mathbb{W}$ becomes the current message, is removed from $\mathbb{W}$ and added to $\mathbb{D}$. Then, the convolution of the current backlog $f_{\hat{\mathcal{B}}}$ and the message execution time is performed to obtain the new backlog (line 3) with its minimum time t_{min} and maximum time t_{max} (line 4). As discussed in Fig. 5.7, different situations need to be considered regarding the resulting backlog and the backlog needs to be separated into portions. These portions depend on the respective next message generation times. That is, the generation time of the next message is determined and the next portion is defined as the time between t_{min} and the next generation time or the end of the current backlog t_{max} (line 6). Then, each portion of the backlog is considered in the while

```
Function $f_{\mathcal{B}} = \texttt{ComputeBacklog}(f_{\mathcal{B}}, f_{\hat{\mathcal{B}}}, t_0, \mathbb{S}, \mathbb{W}, \mathbb{D})$
    $\hat{M} = \mathbb{W}$.pop() and $\mathbb{D} = \mathbb{D} \cup \{\hat{M}\}$
    $f_{\hat{\mathcal{B}}} = \texttt{conv}(f_{\hat{\mathcal{B}}}, f_{C_{\hat{M}}})$
    $t_{\text{min}} = t_0 + \min_b\{f_{\hat{\mathcal{B}}}(b) \neq 0\}$ and $t_{\text{max}} = t_0 + \max_b\{f_{\hat{\mathcal{B}}}(b) \neq 0\}$
    $t_{\text{next}}$ is next message generation time after $t_{\text{min}}$ (use $t_{\text{next}} = \infty$ if there is no
      more message generation after $t_{\text{min}}$)
    $t_{\text{s}} = t_{\text{min}}$ and $t_{\text{e}} = \min\{t_{\text{next}} - 1, t_{\text{max}}\}$, $b_{\text{s}} = t_{\text{s}} - t_0$ and $b_{\text{e}} = t_{\text{e}} - t_0$
    while $t_e \leq t_{max}$ do
        Add all messages with generation times $\leq t_{\text{e}}$ to $\mathbb{W}$, $\mathbb{W} = \mathbb{W} \setminus \mathbb{D}$
        if $\mathbb{W} \neq \emptyset$ then
            $\hat{M} = \mathbb{W}$.peek()
            if $\hat{M}$ is the target message then
                $f_{\mathcal{B}} = f_{\mathcal{B}} + f_{\hat{\mathcal{B}}}[b_{\text{s}}, b_{\text{e}}]$
            else
                $f_{\mathcal{B}} = \texttt{ComputeBacklog}(f_{\mathcal{B}}, f_{\hat{\mathcal{B}}}, t_0, \mathbb{S}, \mathbb{W}, \mathbb{D})$
        else
            if $t_e \neq t_{max}$ then
                $f_{\hat{\mathcal{B}}} = \texttt{accum}(f_{\hat{\mathcal{B}}}[b_{\text{s}}, b_{\text{e}}], b_{\text{e}} + 1)$
            else
                Add all messages with generation times $= t_{\text{next}}$ to $\mathbb{W}$
                $\hat{M} = \mathbb{W}$.peek()
                if $\hat{M}$ is the target message then
                    $f_{\mathcal{B}} = f_{\mathcal{B}} + f_{\hat{\mathcal{B}}}$
                else
                    $t_0 = t_{\text{next}}$ and $f_{\hat{\mathcal{B}}} = \texttt{shrink}(f_{\hat{\mathcal{B}}}[b_{\text{s}}, b_{\text{e}}], t_{\text{next}})$
                    $f_{\mathcal{B}} = \texttt{ComputeBacklog}(f_{\mathcal{B}}, f_{\hat{\mathcal{B}}}, t_0, \mathbb{S}, \mathbb{W}, \mathbb{D})$
        if $t_e == t_{max}$ then
            return $f_{\mathcal{B}}$
        $t_{\text{next}}$ is next message generation time after $t_{\text{e}}$
        $t_{\text{s}} = t_{\text{e}} + 1$ and $t_{\text{e}} = \min\{t_{\text{next}} - 1, t_{\text{max}}\}$, $b_{\text{s}} = t_{\text{s}} - t_0$ and $b_{\text{e}} = t_{\text{e}} - t_0$
```

Algorithm 7: Backlog Distribution Computation.

loop starting from line 7. First, the list of waiting messages is updated (line 8). Then, different cases are possible. If at least one message is waiting (line 9 to 14), the waiting highest-priority message $\hat{M}$ is determined (line 10). If this message is the target message, it means that the target message will be transmitted after the current portion of the backlog, therefore this partial backlog is added to the total backlog (here, addition is not convolution but one-to-one addition such that $f_{\mathcal{B}}(i) = f_{\mathcal{B}}(i) + f_{\hat{\mathcal{B}}}(i)$) (line 11,12). Otherwise, the function is called recursively to determine the backlog for the current portion. If no message is waiting (line 16 to 29), it is checked if the last portion is reached. If no (line 17), the partial backlog is accumulated to the beginning of the next portion since no message is waiting (line 18). If yes this means that the backlog discontinues here. In this case, the list of messages released at the next generation time is obtained (line 20). Then the highest priority $\hat{M}$ is determined (line 21). If this message is the target message, this means the backlog terminates, therefore this partial backlog is added to the total backlog (again, this addition is not convolution but one-to-one addition) (line 22,23). Otherwise, the backlog computation continues with the next message. The starting time is updated as t_{next} and the partial backlog is shrinked at the new starting time t_{next} (line 25). Then the function is called recursively to determine the backlog for this portion (line 26). If the end of the current backlog is reached, (line 30), the function terminates by returning the sum of the computed backlogs $f_{\mathcal{B}}$. Otherwise, the next portion of the backlog is prepared and the while loop continues.

The complexity of the algorithm is $\mathcal{O}(k^m)$ where k is the maximum number of stuff-bits in the considered message set and m is the number of messages. The complexity is expressed according to the worst-case number of calls of the recursive function. Supposing each call is invoked by each partial backlog, in the worst-case there will be one generated message at each stuff-bit and this will lead to k partial backlogs. Also assuming that all of the messages are added to the backlog, this leads to an iteration of depth m leading to k^m calls for the recursive function. This can be seen as a high complexity but in reality the number of calls will be much smaller than this value. In fact, in computations performed for a message set composed of 9 CAN messages and 10 stuff-bits, the average number of calls for the function revealed to be 12 which is incomparable to the worst-case assumption 10^9.

We next illustrate Algorithm 7 by the previous example given in Fig. 5.6. In the first call of the function `ComputeBacklog`, the local backlog is computed by convolving the starting backlog $f_{\hat{\mathcal{B}}}(0) = 1$ with the execution time *pmf* of the first message M_1. Recall that backlog due to M_1 had three partial backlogs as shown in Fig. 5.8.

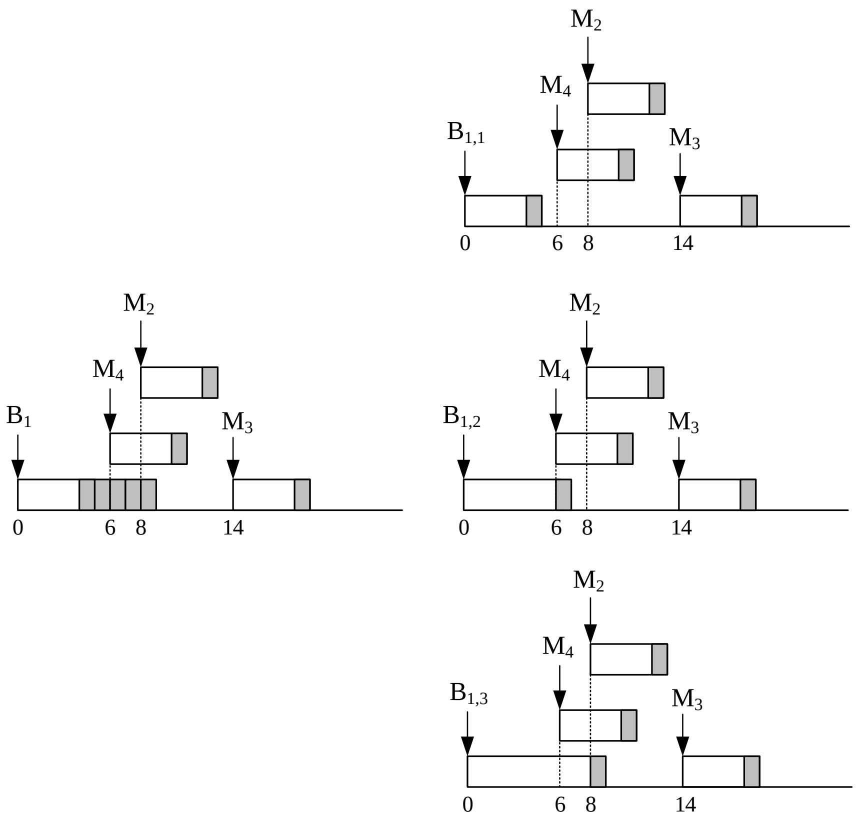


Figure 5.8: Partial Backlogs of the Backlog due to M_1

In order to track the flow of the algorithm easily, local backlog found by each additional message is denoted by $B_{\mathrm{i,j}}$ (corresponding to $f_{\hat{\mathcal{B}}}$ in the algorithm) where i denotes the number of calls of the function `ComputeBacklog` and j denotes the partial backlog number of B_{i}.

For the first partial backlog $B_{1,1}$, the algorithm ends in line-18 since this backlog ends with an empty bus. In this case the partial backlog is accumulated to the beginning of the next partial backlog by `accum` operator.

For the second partial backlog $B_{1,2}$, the only pending message is M_4, therefore the algorithm ends in line-14 and calls itself recursively. The partial backlog $B_{1,2}$ is transferred as the current backlog to the next call. In the next call, the new local backlog B_2 is computed by convolving the current backlog $B_{1,2}$ with the execution time *pmf* of M_4. This operation is depicted in Fig. 5.9.

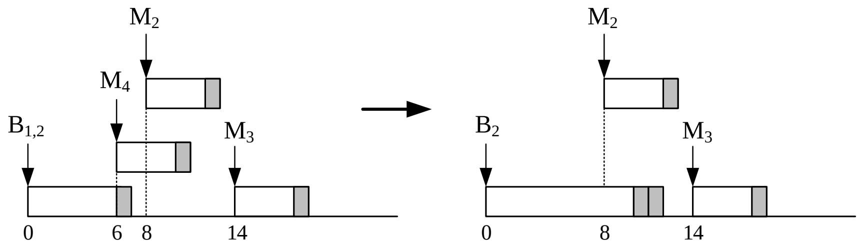


Figure 5.9: Obtaining the Local Backlog B_2

Since there are no messages released in the variable portion of B_2, no partial backlog is produced. The only pending message is M_2, therefore the algorithm ends in line-14 and calls itself recursively. The backlog B_2 is transferred as the current backlog to the next call. In the next call, the new local backlog B_3 is computed by convolving the current backlog B_2 with the execution time *pmf* of M_2. This operation is depicted in Fig. 5.10.

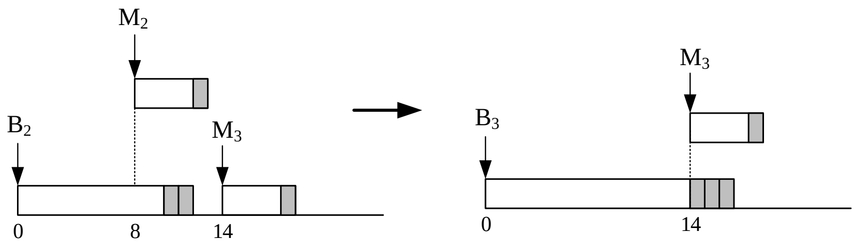


Figure 5.10: Obtaining the Local Backlog B_3

Since there are no messages released in the variable portion of B_3, no partial backlog is produced. The only pending message is the target message M_3, therefore the algorithm ends in line-12 and stores the partial backlog (the entire backlog B_3 in this case) to B_{total} (corresponding to $f_{\mathcal{B}}$ in the algorithm). Since there are no more partial backlogs for this local backlog, the function returns to the previous recursive call with the last updated B_{total}. Returning back to B_2, since there are no more partial backlogs also for B_2, the function returns to the first call of `ComputeBacklog` ending up with the local backlog B_1.

The next partial backlog for B_1 is $B_{1,3}$. There are two messages released before the backlog ends: M_4 and M_2. These messages are added to the priority queue and the function calls itself recursively in line-14. In the new call, the highest priority message is popped from the priority queue, which is M_2. Then the local backlog B_4 is computed by convolving the current backlog $B_{1,3}$ with the execution time *pmf* of M_2. The resulting backlog is depicted in Fig. 5.11.

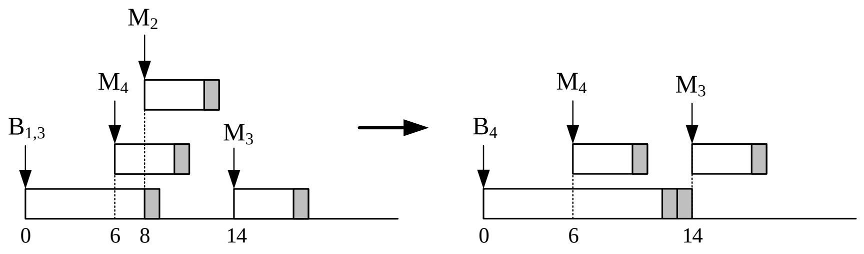


Figure 5.11: Obtaining the Local Backlog B_4

At this time, there are two partial backlogs because the target message M_3 is released just at the end time of the local backlog B_4. The partial backlogs are depicted in Fig. 5.12.

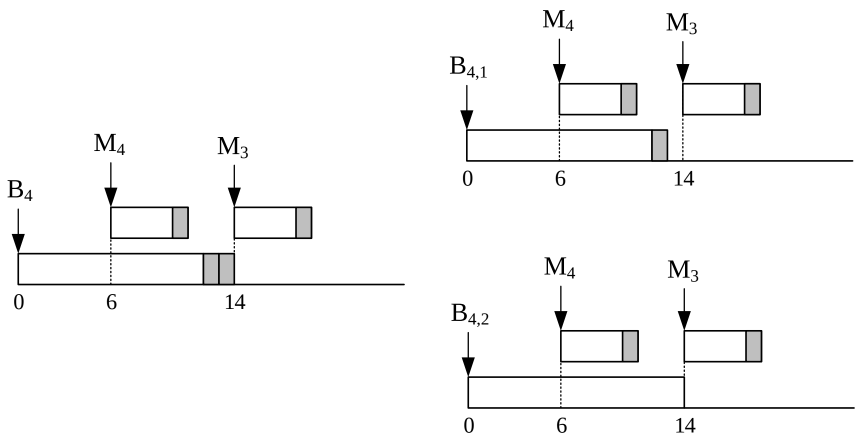


Figure 5.12: Partial Backlogs of the Local Backlog B_4

For the first partial backlog $B_{4,1}$, the only pending message is M_4, therefore the algorithm ends in line-14 and calls itself recursively. Here, the new local backlog B_5 is computed by convolving the current backlog $B_{4,1}$ and the execution time *pmf* of M_4. The resulting backlog is depicted in Fig. 5.13.

In B_5, there are no partial backlogs and the only pending message is the target message. Therefore, the function ends in line-18 storing the current backlog B_5 to B_{total} and returns with the stored B_{total} value to B_4.

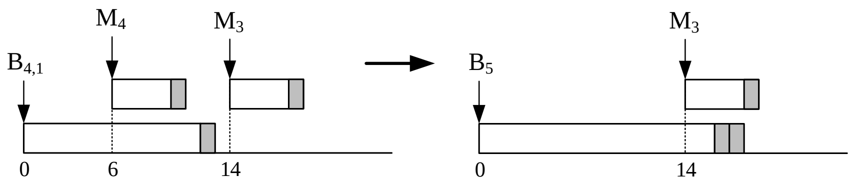


Figure 5.13: Obtaining the Local Backlog B_4

In the second partial backlog $B_{4,2}$, the pending messages are M_4 and M_3. Since the highest priority message is the target message, the function ends in line-12, storing the partial backlog $B_{4,2}$ to B_{total}. Then the function returns to the first call B_1. Since there are no more partial backlogs in B_1, the function returns for the last time with the B_{total} that stores the sum off all computed backlogs.

In this example, the function `ComputeBacklog` is called 5 times.

5.2.4 Response Time Distribution Computation for Given Node Phases

As stated before, the computation of the backlog distribution is the main contribution towards the computation of the response time distribution of a given target message. We next use the computed backlog distribution in Section 5.2.3 in order to determine the response time distribution in two steps given a message stream $\mathbb{S}$.

First, we identify the part of the message stream that interferes with the transmission of the target message. To this end, we first note that such interference can be determined using the longest execution time of each message since a non-zero interference probability is already given if all messages before the generation of the target message assume there longest execution time. To illustrate this fact, consider the example message queuing shown in Fig. 5.14.

Suppose that we want to find the message set interfering with message M_3. The backlog resulting from the best-case and the worst-case execution times for the earlier messages M_1, M_2 and M_4 is shown in Fig. 5.15.

As can be seen, adding the shortest execution times, the total execution finishes at $t = 14$, before the release time of the target message M_3 causing no interference on M_3. On the other hand, adding the longest execution times, the total execution finishes

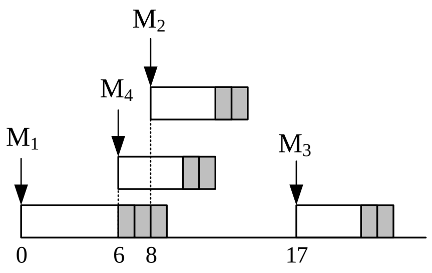


Figure 5.14: Example Message Queuing

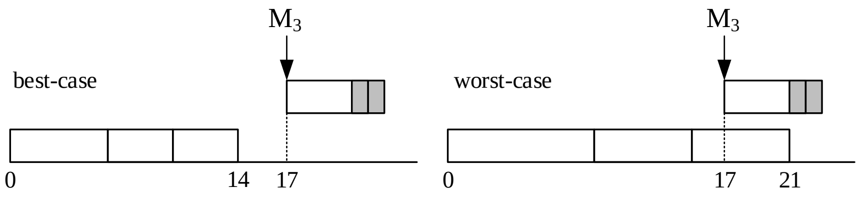


Figure 5.15: Best-Case and Worst-Case Execution Time Scenarios

at $t = 21$, after the release time of the target message M_3 causing interference on M_3. We conclude that potential interference is obvious from the longest execution times.

Accordingly, the messages in $\mathbb{S}$ interfering with the target messages are computed with the following Algorithm 8.

In this algorithm, the message set which accumulates and interferes with the target message M_T is determined. Starting from the first starting time $t_0 = 0$, the maximum size of the next message is added to the variable t_{inter} which accumulates the observed interference (line 10). In case, the interference does not extend to the next waiting message (line 11), a new starting time for the interference is defined (line 12). The algorithm terminates if the target message is the next message to gain access to the bus. The time t_0 obtained at the end of the algorithm is the time from where the response time distribution computation starts.

Using Algorithm 8 and 7, it is now possible to compute the response time distribution of a given target message M_T.

Algorithm 9 first determines the generation time t_0 of the first message interfering with the target message M_T. Then, the backlog distribution is computed starting from t_0 and the resulting backlog is convolved with the execution time distribution of message M_T.

```
1 Function $t_0 = \mathtt{ComputeStartTime}(\mathbb{S}, M_T))$
2     $t_0 = 0$, $t_{\text{inter}} = 0$, $\mathbb{D} = \emptyset$
3     $\hat{M}$ is the highest-priority message at $t_0$
4     $t_{\text{inter}} = C_{\hat{M}}$, $\mathbb{D} = \mathbb{D} \cup \{\hat{M}\}$
5     $t_T$ is the generation time of the target message
6     while End of $\mathbb{S}$ is not reached do
7         $\hat{M}$ is highest-priority message waiting in message stream after $t_0$ with
           $\hat{M} \notin \mathbb{D}$
8         $t_{\hat{M}}$ is set as generation time of $\hat{M}$
9         if $\hat{M}$ is not the target message then
10            if $t_{inter} \geq t_{\hat{M}}$ then
11                $t_{\text{inter}} = t_{\text{inter}} + C_{\hat{M}}$ and $\mathbb{D} = \mathbb{D} \cup \{\hat{M}\}$
12            else
13                $t_0 = t_{\hat{M}}$ and $t_{\text{inter}} = C_{\hat{M}}$ and $\mathbb{D} = \mathbb{D} \cup \{\hat{M}\}$
14            if No more message is waiting in $\mathbb{S}$ then
15                return $t_0$
16        else
17            return $t_0$
```

Algorithm 8: Determining the interfering messages.

```
1 Function $\mathcal{R}_{M_T} = \mathtt{ComputeRTD}(\mathbb{S}, M_T))$
2     $t_0 = \mathtt{ComputeStartTime}(\mathbb{S}, M_T)$
3     $f_{\mathcal{B}} = 0$
4     $f_{\hat{\mathcal{B}}} = [0] \rightarrow [1]$
5     $\mathbb{W} = \mathbb{D} = \emptyset$
6     Add all messages with generation times $= t_0$ to $\mathbb{W}$
7     $f_{\mathcal{B}} = \mathtt{ComputeBacklog}(f_{\mathcal{B}}, f_{\hat{\mathcal{B}}}, t_0, \mathbb{S}, \mathbb{W}, \mathbb{D})$
8     $f_{\mathcal{B}} = \mathtt{shrink}(f_{\mathcal{B}}, t_T)$
9     $\mathcal{R}_{M_T} = \mathtt{conv}(f_{\mathcal{B}}, \mathcal{C}_{M_T})$
10    return $\mathcal{R}_{M_T}$
```

Algorithm 9: Response Time Distribution Computation.

As an illustration, consider the example in Fig. 5.3 where the backlog was found as

$f_{\mathcal{B}_2} = [9, 10, 11, 12, 13, 14] \rightarrow [0.04, 0.14, 0.26, 0.28, 0.2, 0.08]$

which is the return value of `ComputeBacklog` function. The response time *pmf* of M_2 is then calculated by Algorithm 9 as

$f_{\hat{\mathcal{B}}_2} = \texttt{shrink}(f_{\mathcal{B}_2}, 11) = [0, 1, 2, 3] \rightarrow [0.44, 0.28, 0.2, 0.08].$

$f_{\mathcal{R}_2} = \texttt{conv}(f_{\mathcal{B}_2}, f_{\mathcal{C}_2}) = [4, 5, 6, 7, 8, 9] \rightarrow [0.088, 0.232, 0.328, 0.208, 0.112, 0.032].$

The computation of the response time distribution is repeated for all instances $M_{T,j}$ of the target message M_T generated within the hyperperiod H_k of node N_k which generates the target message. Starting from the first instance, the response time distribution $\mathcal{R}_{M_{T,j}}$ is computed for the j-th instance with generation time evaluated by (5.2). Then the average response time distribution of M_T is obtained as

$$\mathcal{R}_{M_T} = \frac{1}{H_k/T_{M_T}} \cdot \sum_{j=1}^{H_k/T_{M_T}} \mathcal{R}_{M_{T,j}}. \tag{5.9}$$

5.2.5 General Response Time Computation

The computation of the response time distribution in Section 5.2.4 is based on the assumption of a given message stream $\mathbb{S}$. Given an offset assignment for messages of each node, the composition of this message stream depends on the node phases. That is, for each combination of node phases, a different message stream with different generation times of messages is obtained based on the evaluation of (5.2).

Considering a CAN bus with n nodes and a hyperperiod H_k for each node $1 \leq k \leq n$, there are

$$\prod_{k=1}^{n-1} H_k$$

combinations of node phases. Here, node n was considered as the reference node. It is further the case that each of the described combinations potentially leads to a different response time distribution since the message stream is different. In the literature, it is considered that each of the node phases is equally likely such that the overall response time distribution of a target message M_T is obtained by averaging over the response time distributions for the different node phases [20]. Let $\mathbb{P}$ be the set of all possible node phases and denote the response time distribution for certain node phases $p \in \mathbb{P}$ as $\mathcal{R}^p_{M_T}$. Then, the overall response time distribution for message M_T is

$$\mathcal{R}_{M_T} = \frac{1}{|\mathbb{P}|} \cdot \sum_{p \in \mathbb{P}} \mathcal{R}^p_{M_T}. \tag{5.10}$$

The complexity of the overall response time distribution computation can be expressed as $\mathcal{O}(H^n \cdot k^m)$, where H denotes a bound on the hyperperiods of all nodes.

As an illustration, consider the previous example message queuing case shown in Fig. 5.6 such that messages M_1 to M_4 have the properties given by Table 5.3.

Table5.3: Message Properties

M_i	N_k	P_i	T_i	O_i	$f_{\mathcal{C}_i}$
M_1	N_1	1	30	0	$[4,5,6,7,8,9] \to [0.1,0.2,0.2,0.2,0.2,0.1]$
M_2	N_2	2	30	2	$[4,5] \to [0.5,0.5]$
M_3	N_1	3	30	14	$[4,5] \to [0.4,0.6]$
M_4	N_2	4	30	0	$[4,5] \to [0.7,0.3]$

Since all messages have a period of 30, the hyperperiod of each node is 30. This leads to 30 possible phasing scenarios to compute. One of these scenarios is depicted in Fig. 5.16 for $\Phi_{2,1} = 6$ which is exactly the queuing case shown in Fig. 5.6.

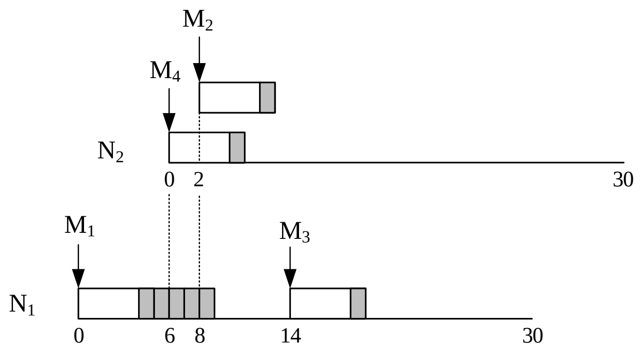


Figure 5.16: Message Queuing Scenario for $\Phi_{2,1} = 6$

The overall response time distribution for the message M_3 in terms of cumulative distribution function (*cdf*) is shown in Fig. 5.17. The response time *cdf* for two specific phases $\Phi_{2,1} = 13$ and $\Phi_{2,1} = 28$ is also given in the same figure.

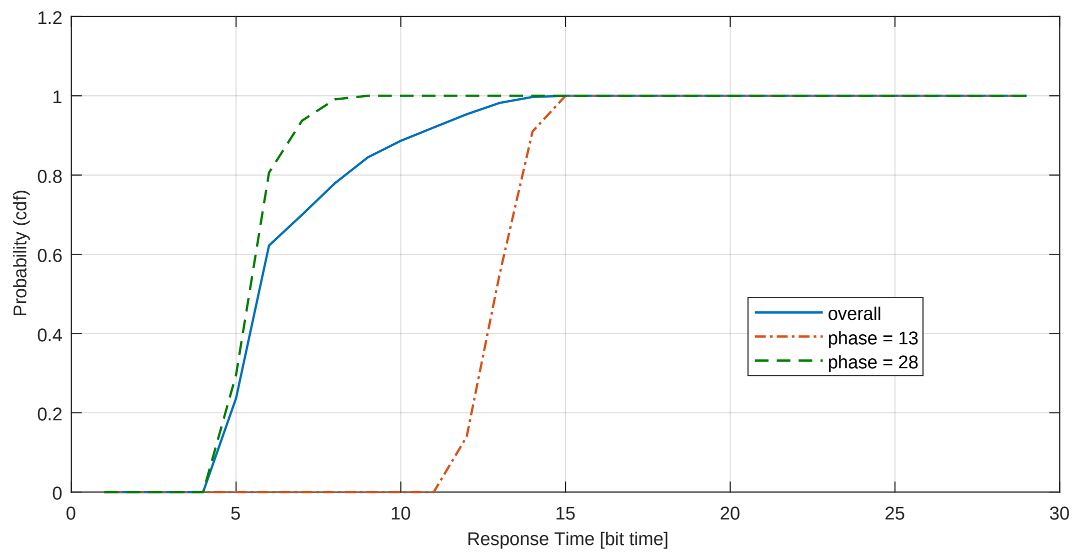


Figure 5.17: Response Time *cdf*s of M_3

5.3 Hardware Measurements and Discussion

We performed response time distribution experiments for a CAN bus with 3 nodes with a data rate of 125 kbit/s using the same test setup as in Chapter 4. In addition, we implemented the response time distribution computation algorithm in MATLAB and compared the experimental results with the computational ones.

The response time measurement experiments are run for 10 minutes resulting in transmission of approximately 60000 instances of each message. For each experiment, the response time is measured as described in Section 4.3. Then the response time *cdf* is obtained by using a parsing script for the CAN bus log file.

The main observation of these initial experiments was that the computed response time distribution was not observed in the measurement. An example of the observed mismatch for an experiment with 3 nodes and 9 messages is given in Fig. 5.18 with message properties as in Table 5.4. In this case message M_9 was chosen as the target message.

As can be seen, the computational *cdf* has three steps where the response time differs much. However in the measured *cdf* the middle step is not observed.

At this point, it could be argued that the proposed response time distribution compu-

Table5.4: Message Properties

M	N_M	P_M	T_M	L_M	O_M
M_1	N_1	1	10 ms	8 B	0
M_4	N_1	4	10 ms	8 B	0
M_7	N_1	7	10 ms	8 B	0
M_2	N_2	2	10 ms	8 B	0
M_5	N_2	5	10 ms	8 B	0
M_8	N_2	8	10 ms	8 B	0
M_3	N_3	3	10 ms	8 B	0
M_6	N_3	6	10 ms	8 B	0
M_9	N_3	9	10 ms	8 B	0

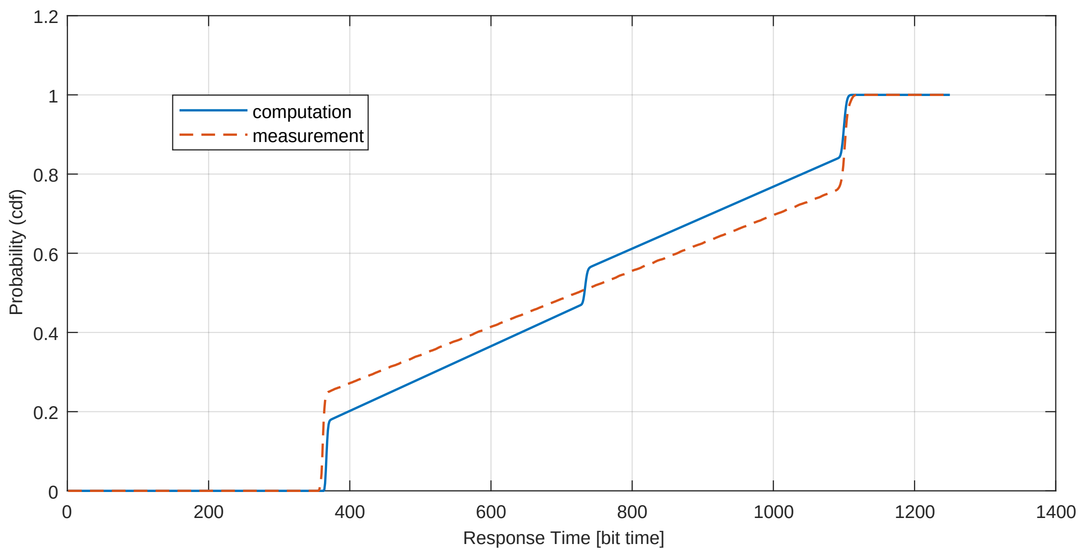


Figure 5.18: Response Time *cdf* Comparison for M_9

tation is incorrect. Nevertheless, a more detailed analysis of the stated assumptions reveals that one of the assumptions for the response time distribution computation is not valid in practice: the phases between nodes do not change arbitrarily but depending on the drift between the clocks of different nodes. That is, locally (for a long enough period of time in the order of tens of minutes) phases remain similar. As a result, the response time distribution changes over time according to the changes in the node phases.

We first illustrate this situation and then propose a simple modification of the response time distribution computation that allows a close match between computation and measurements. We choose the message set given in Table 5.5. Then we identify two extreme phasing scenarios for the target message. Then the nodes are forced to start with the chosen phasing scenarios. The target message is chosen as the lowest priority message which is M_9.

Table5.5: Message Properties

M	N_M	P_M	T_M	L_M	O_M
M_1	N_1	1	10 ms	8 B	0
M_4	N_1	4	10 ms	8 B	1
M_7	N_1	7	10 ms	8 B	2
M_2	N_2	2	10 ms	8 B	0
M_5	N_2	5	10 ms	8 B	1
M_8	N_2	8	10 ms	8 B	2
M_3	N_3	3	10 ms	8 B	1
M_6	N_3	6	10 ms	8 B	2
M_9	N_3	9	10 ms	8 B	0

The first scenario is a "bad case" for message M_9 in which the phasings of the nodes are arranged such that M_9 is blocked by all other messsages in the bus as shown in Fig. 5.19.

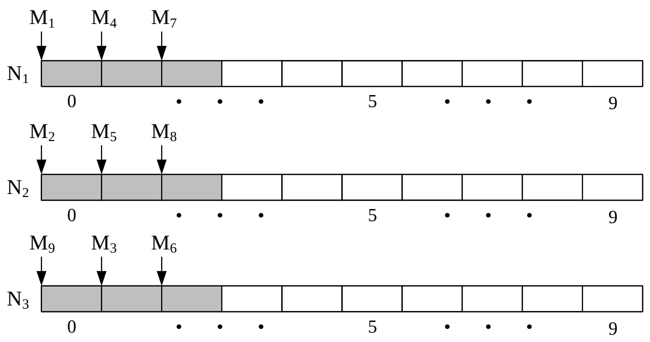


Figure 5.19: Bad Phasing Scenario for M_9

The second scenario is a "good case" for message M_9 in which the phasings of the nodes are arranged such that M_9 is not blocked by any message on the bus. This scenario is depicted in Fig. 5.20.

The response time measurement experiments are run for 10 minutes starting from the phase arrangements shown in Fig. 5.19 and Fig. 5.20. For each experiment, the response time for M_9 is measured as described in Section 4.3. Then the response time *cdf* is obtained by using a parsing script for the CAN bus log file. The measurement results for both bad and good cases along with the exact computation result is shown

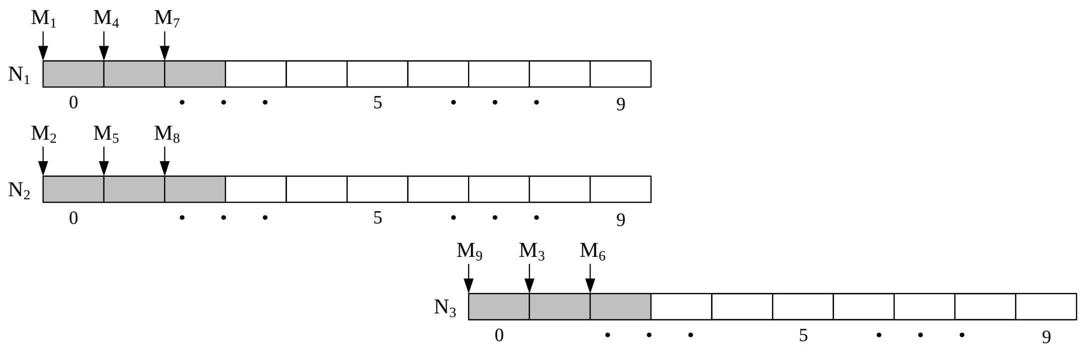


Figure 5.20: Good Phasing Scenario for M_9

in Fig. 5.21.

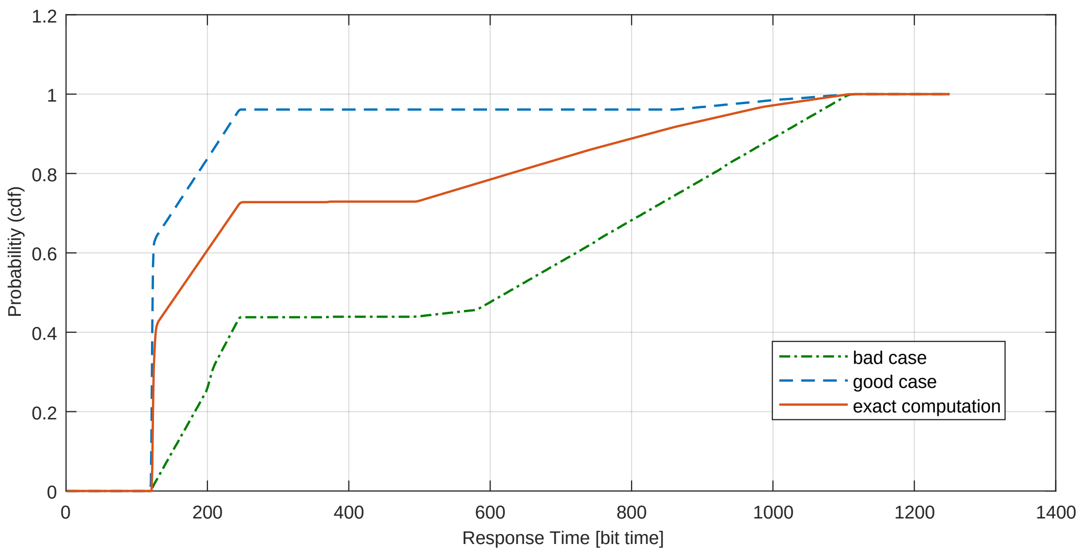


Figure 5.21: Response Time *cdf* Comparison for M_9

As can be seen, the computed *cdf* stays right in the middle of the two *cdf*s obtained from the measurements. In the bad phasing scenario, the probability that M_9 experiences large response times is higher compared to the computational results. This is an expected result since the nodes are forced to start with the worst-case phasing scenario for M_9. The phase shift due to clock drift among the nodes results in smaller response times for M_9. However, this drift is so slow and particular that even for a time period of 10 minutes, a very specific response time *cdf* that is far away from the computational *cdf* is obtained. This is also true for the good phasing scenario in which again a specific distribution is obtained with higher probabilities for smaller response times compared to the computational results. In general, it holds for any phasing scenario that each starting situation will result in a specific *cdf* that lies between the *cdf*s obtained in the bad and good cases.

5.4 Local Response Time Distribution

5.4.1 Local Response Time Distribution

In the previous section, it was identified from hardware measurements that the node phases do not change arbitrarily. As a result, the computed response time distribution does not agree well with the measured response times. This section proposes the new idea of a local response time distribution as a remedy for the observed issue.

5.4.2 Local Response Time Distribution Computation

(5.10) in Section 5.2.5 evaluates the response time distribution which is averaged over all possible node phases. According to the observation in the previous section, we now suggest to only consider node phases that are close to the initial node phase at system startup. As explained before, this modification captures the fact that the node clocks drift slowly and gradually. Consider that the initial node phases are given by $\Phi_{k,l}^0$ for a reference node N_k and the remaining nodes $l \neq k$. Then, we define a range $\Delta\Phi$ that captures the possible deviation of each node phase from the initial value and consider only node phases between $\Phi_{k,l} - \Delta\Phi$ and $\Phi_{k,l} + \Delta\Phi$ for each node l. Writing $\mathbb{L}$ for the set of local node phase combinations, the local response time distribution is computed as

$$\mathcal{R}_{M_T} = \frac{1}{|\mathbb{L}|} \cdot \sum_{p \in \mathbb{L}} \mathcal{R}_{M_T}^p. \tag{5.11}$$

5.4.3 Evaluation

In order to evaluate the modified algorithm, we consider the same example as in Section 5.3. The comparison of a local response time *cdf* computation with the measurement is given in Fig. 5.22 and Fig. 5.23 for the bad and good cases, respectively. In computations, node N_3 is selected as the reference node and the window of phases for nodes N_1 and N_2 is considered as $\Delta\Phi = 300\tau_{\text{bit}}$ for both sides relative to N_3. In comparison, the hyperperiod contains $10ms/8\mu s = 1250\tau_{\text{bit}}$ since the hyperperiod duration $H = 10ms$ and $\tau_{\text{bit}} = 8\mu s$ at 125 kbit/s data rate.

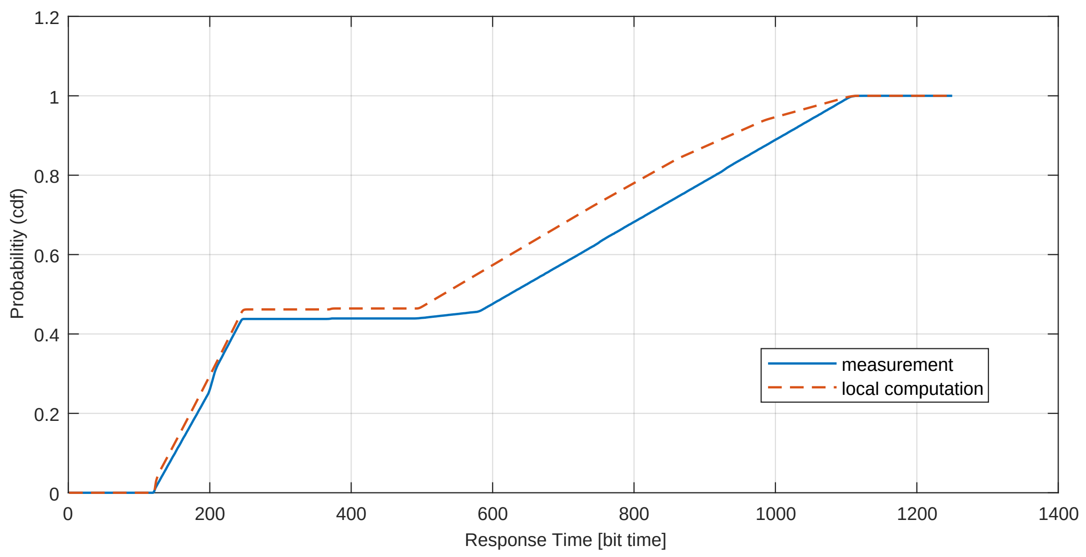


Figure 5.22: Comparison of Local Response Time *cdf* Computation and Measurement in Bad Case

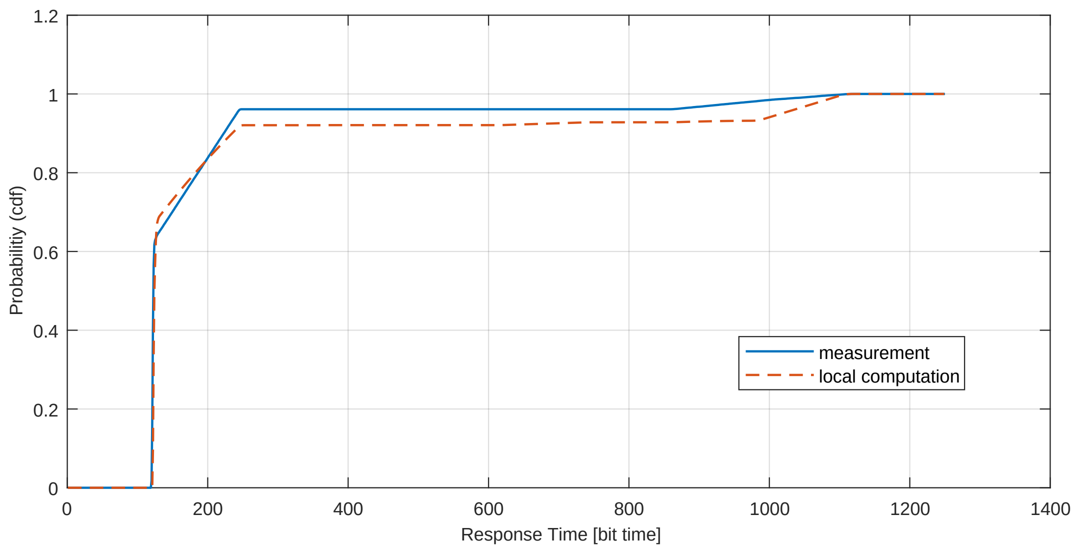


Figure 5.23: Comparison of Local Response Time *cdf* Computation and Measurement in Good Case

It can be seen that computations with bounded phase shift centered at the initial phase scenario results in computational results that are very close to the measurements.

5.4.4 Discussion of the Local Response Time Distribution

In the previous section we have shown that the local response time distribution very closely captures the actual response time distribution from measurements. This means that it is possible to computationally evaluate response time distributions on CAN for a certain time interval if the initial node phases are known. Although this is a very positive result that was not known in the existing literature, it has two major disadvantageous practical implications.

First, the obtained result shows that computing the overall response time distribution can lead to highly optimistic results. Just consider the example in Fig. 5.21. According to the figure, the exact response time distribution computation suggests that the response time is below 800 τ_{bit} with a probability of 90%. It is then possible that a system designer assumes that this probability is sufficient for safety such that the example CAN network is deemed safe. Nevertheless, looking at the "bad case", it is possible for some time (locally) that message response times are larger than 800 τ_{bit} with a probability of more than 30%. That is, a CAN network that is considered safe might be unsafe locally.

Second, although the computation of the local response time distribution would solve the described problem, this computation is based on the knowledge of the current node phases. Since there is no synchronization among CAN nodes, the node phases are actually unknown such that it is not possible to effectively compute the local response time distribution during system run-time.

In the next section, we propose to apply the ides of weak synchronization between CAN nodes in order to be able to apply the idea of the local response time distribution.

5.5 Weak Synchronization

5.5.1 Main Idea and Motivation

As was discussed in the previous section, it is necessary to obtain knowledge about the initial phase shift and the boundaries in which the phases between the nodes vary. As a solution to this problem, we implement weak synchronization between the nodes, where all nodes on the CAN bus are synchronized with a weak precision. The advantage of the suggested method is that both the identified issues are addressed and only a single CAN message that is sent infrequently is added to the CAN bus.

5.5.2 Description of the Method

The implemented weak synchronization method is the one proposed by Gergeleit and Streich (1994) [9] since it has minimal additional load on the bus. The method is based on a master-slave concept where one of the nodes in the system is designated as the master and all other nodes as slaves. The synchronization among the nodes is achieved by a single synchronization (sync) message that is sent periodically by the master node as shown in Fig. 5.24 where each gray arrow represents the transmission of a sync message.

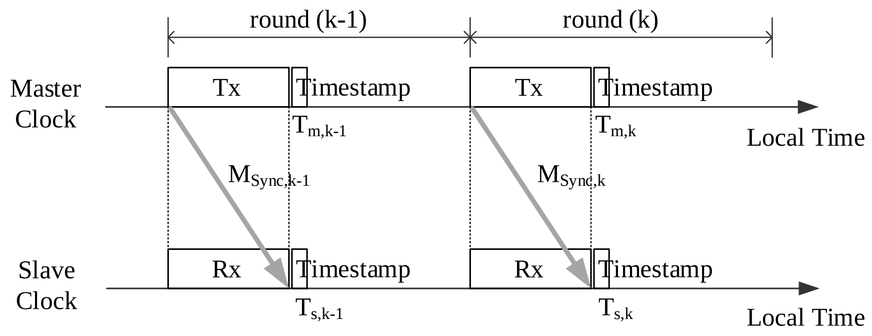


Figure 5.24: Clock Synchronization Method

From Fig. 5.24, it can be seen that local clock timestamp is taken on the master node when a sync message is transmitted and on the slave nodes when a sync message is received. Successful transmission of a message in CAN is synchronous to the reception of the message on the receiving nodes. Using this property, it is assured that the timestamps are taken at the same instant on the master and the slaves. The timestamp taken on the master is sent by the next sync message as a reference time to the slaves. For instance, in round k in Fig. 5.24, the master broadcasts a sync message $M_{\text{sync,k}}$ containing its timestamp taken at $T_{\text{m,k-1}}$ when the previous sync message $M_{\text{sync,k-1}}$ was transmitted to the slaves. Then, every slave in the network receives this message at $T_{\text{s,k-1}}$ and takes a timestamp right after the reception. Then each slave adjusts its local clock using the difference between the timestamps $T_{\text{s,k-1}}$ and $T_{\text{m,k-1}}$.

5.5.3 Algorithm

We implement the weak synchronization method described in the previous section on the Fujitsu SK-91465X-100PMC Evaluation Boards used as both master and slave nodes. The global clock is expressed as two time units called *tick* and *cycle* as shown in Fig. 5.25.

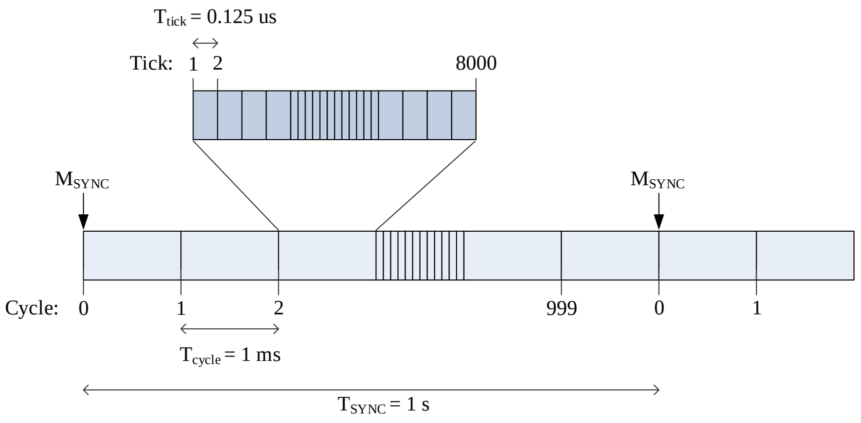


Figure 5.25: Global Clock Time Units

The ticks are generated by the microcontroller's own clock and have a duration of $T_{\text{tick}} = 0.125\mu$s. Each cycle is composed of 8000 ticks with a duration of $T_{\text{cycle}} = 1$ ms. The update of cycle number is triggered by a timer interrupt routine when the value set to the timer period register is elapsed, which is 8000 in our case. The cycle counter is a rolling counter which increments the cycle number at each timer interrupt call and rolls back to 0 after 999. This roll back creates the rounds for sending sync messages with a period of $T_{\text{sync}} = 1s$ such that when the cycle counter value is 0, a sync message transmission is invoked on the master.

The flow chart of the algorithm running in the master node is shown in Fig. 5.26. As can be seen, the master stays in an idle waiting loop until an interrupt is invoked either by the timer or the transmission of a sync message. In this case the master executes the related interrupt routine and returns to the idle loop. The timer period in the master node is constant during run-time as $T_{\text{cycle}} = 8000$ ticks resulting in an interrupt event at every 1 ms. When this period elapses, the CPU triggers a timer interrupt and executes the interrupt routine given in Algorithm 10. Here, the master updates its rolling cycle counter n_{cycle}. As stated, when $n_{\text{cycle}} = 0$ (line 5) a sync message is generated containing the timestamp which is a tuple $\{N_{\text{cycle}}, N_{\text{tick}}\}$. When the generated sync message is successfully transmitted, the CAN transmission interrupt is invoked by the CPU and the routine given by Algorithm 11 is executed. In this routine, the master's only task is to take the timestamp $\{N_{\text{cycle}}, N_{\text{tick}}\}$. Note that N_{tick} is obtained from the timer counter register of the microcontroller which counts and rolls independently.

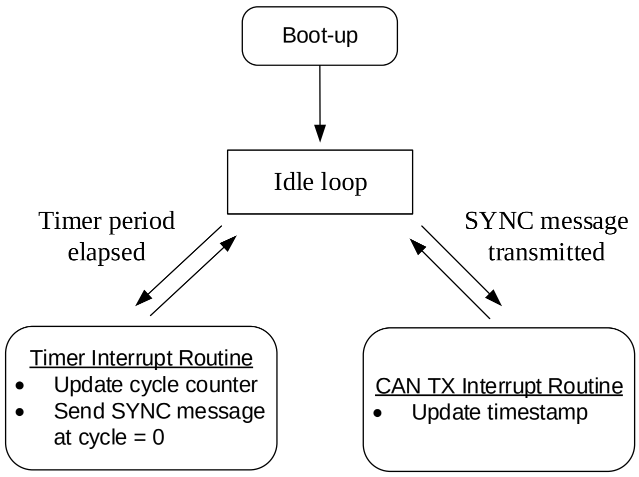


Figure 5.26: Flow Chart for the Master Node

```
1 Function TimerInterrupt()
2     increment $n_{\text{cycle}}$
3     if $n_{cycle} == 1000$ then
4         reset $n_{\text{cycle}}$
5         CANSendMessage($M_{\text{SYNC}}, \{N_{\text{cycle}}, N_{\text{tick}}\}$)
6     return
```

Algorithm 10: Timer Interrupt Routine for the Master Node.

```
1 Function CANTxInterrupt()
2     $\{N_{\text{cycle}}, N_{\text{tick}}\} = \{n_{\text{cycle}}, n_{\text{tick}}\}$
3     return
```

Algorithm 11: CAN Tx Interrupt Routine for the Master Node.

The flow chart of the algorithm running in a slave node is shown in Fig. 5.27. Similar to the master, the slave stays in an idle waiting loop until an interrupt is invoked either by the timer or the reception of a sync message. In this case the slave executes the related interrupt routine and returns to the idle loop. When a sync message is successfully received, the CAN reception interrupt is invoked by the CPU and the routine given by Algorithm 12 is executed. Here, the slave node stores its own timestamp taken at the previous call to $\{N^{\text{old}}_{\text{cycle,s}}, N^{\text{old}}_{\text{tick,s}}\}$ to be used in the calculation of the clock drift (line 2). Then it takes the current timestamp $\{N^{\text{new}}_{\text{cycle,s}}, N^{\text{new}}_{\text{tick,s}}\}$ (line 3). Later, it gets the master timestamp $\{N_{\text{cycle,m}}, N_{\text{tick,m}}\}$ by reading the payload of the received sync message (line 4). Finally it computes the clock difference relative to the master node in terms of the number of ticks as:

$$\Delta_{\text{tick}} = (N^{\text{old}}_{\text{cycle,s}} - N_{\text{cycle,m}}) \cdot 8000 + N^{\text{old}}_{\text{tick,s}} - N_{\text{tick,m}} \tag{5.12}$$

This value is used as the clock correction term in the slave node as follows. Different from the master, the timer period in the slave node is not constant during run-time. At start-up, it is initialized as $T_{\text{cycle}} = 8000$. Depending on the clock drift Δ_{tick} value, the timer period value is changed in the timer interrupt routine given in Algorithm 13. If Δ_{tick} is greater than 1000 ticks (line 5), this means that the slave clock is 1000 ticks ahead from the master clock. Therefore the timer period is adjusted as $T_{\text{cycle}} = 8001$ (line 6) such that the cycle counter is slowed down by 1 tick for the entire round resulting in slow-down of 1000 ticks at the end of the round to catch the master clock. Similarly if Δ_{tick} is smaller than -1000 ticks (line 7), this means that slave clock is 1000 ticks behind the master clock. Therefore the timer period is adjusted as $T_{\text{cycle}} = 7999$ (line 8) such that the cycle counter is speeded up by 1 tick for the entire round resulting in speed-up of 1000 ticks at the end of the round to catch the master clock. Lastly, if Δ_{tick} is between 1000 and -1000, the slave clock is considered in the synchronous range and no speed-up or slow-down action is applied. In this case, the timer period value is set as the usual value $T_{\text{cycle}} = 8000$.

Note that since nodes can boot-up at arbitrary times, in order to minimize the initial drift, the timer of each slave is started with the first received sync message.

To sum up, the algorithm tries to keep the clock drift of the slave clock in $[-1000, 1000]$ ticks range relative to the master clock. This results in $(1000) \cdot (0.125) = 125\mu$s drift time for either side. Therefore, the overall drift between any two nodes in the network

```
1 Function CANRxInterrupt()
2     $\{N^{\text{old}}_{\text{cycle,s}}, N^{\text{old}}_{\text{tick,s}}\} = \{N^{\text{new}}_{\text{cycle,s}}, N^{\text{new}}_{\text{tick,s}}\}$
3     $\{N^{\text{new}}_{\text{cycle,s}}, N^{\text{new}}_{\text{tick,s}}\} = \{n_{\text{cycle}}, n_{\text{tick}}\}$
4     $\{N_{\text{cycle,m}}, N_{\text{tick,m}}\} = \texttt{CANReadMessage}()$
5     $\Delta_{\text{tick}} = \texttt{ComputeDeltaTick}(\{N^{\text{old}}_{\text{cycle,s}}, N^{\text{old}}_{\text{tick,s}}\}, \{N_{\text{cycle,m}}, N_{\text{tick,m}}\})$
6     return
```

Algorithm 12: CAN Rx Interrupt Routine for a Slave Node.

```
1 Function TimerInterrupt()
2     increment $n_{\text{cycle}}$
3     if $n_{cycle} == 1000$ then
4         reset $n_{\text{cycle}}$
5     if $\Delta_{tick} > 1000$ then
6         $T_{\text{cycle}} = 8001$
7     else if $\Delta_{tick} < -1000$ then
8         $T_{\text{cycle}} = 7999$
9     else
10        $T_{\text{cycle}} = 8000$
11    return
```

Algorithm 13: Timer Interrupt Routine for a Slave Node.

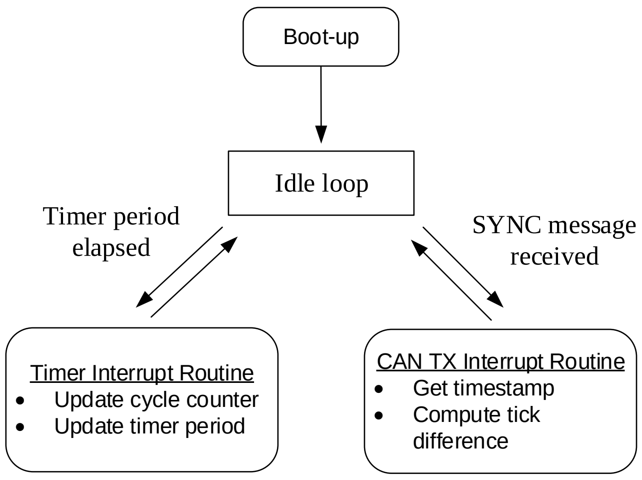


Figure 5.27: Flow Chart for a Slave Node

is assured to be less than $(2) \cdot (125) = 250\mu$s.

5.5.4 Hardware Measurements

We performed weak synchronization experiments for a CAN bus with 3 nodes, one master node and two slave nodes, with a data rate of 125 kbit/s using the same test setup as in Chapter 4. In order to measure the quality of the synchronization, the time drift Δ_{tick} is transmitted by a diagnostic CAN message on each slave node as a 4-byte data. We started the nodes at arbitrary times (first the master node) and run the setup for 1 hour. Later parsed the resultant log file to obtain the results. The variation of clock drift between the slave clock and the master clock is given in Fig. 5.28 for slave-1 and in Fig. 5.29 for slave-2.

As can be seen, the overall time swing is less than 250μs in both of the slaves. Also it is observed that this drift stays in the expected region even for long run-times as much as 1 hour. In order to see the benefit of the weak synchronization, we disabled the clock correction parts in the slave nodes and repeated the same experiment. The variation of the time difference when weak synchronization is disabled can be seen in Fig. 5.30 both for slave-1 and slave-2 for a test run of 20 minutes.

As can be seen, even when the slave clocks start synchronous to the master clock with the reception of the first sync message, the time drift increases dramatically since clock correction is disabled. The clock drift increases to the values over 10 ms

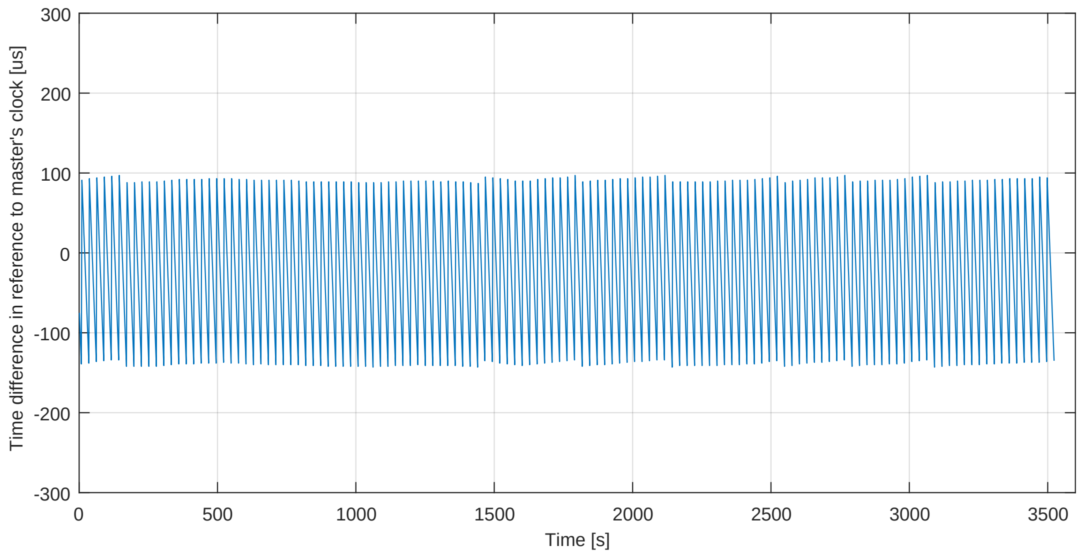


Figure 5.28: Variation of the Time Difference between Slave-1 and Master Clock under Weak Synchronization

in slave-1 and over 10 ms in slave-2 after 20 minutes. Also it is observed that slave-2 drifts faster compared to slave-1, which shows that prediction of the phase shift boundaries for a node without weak synchronization is not possible. By using weak synchronization, it is known that the phase shift boundaries are 125μs away from the initial phase shift.

5.5.5 Local Response Time Distribution Experiments under Weak Synchronization

In order to evaluate the response time distribution under weak synchronization, we consider the same example as in Section 5.3. The nodes are synchronized with weak synchronization where node N_1 is used as the master node and the other two nodes as the slaves. In computations, the master node N_1 is selected as the reference node and the window of phases for N_2 and N_3 is considered as $\Delta\Phi = 125\mu s/8\mu s \approx 15\tau_{\text{bit}}$ for both sides relative to node N_1 where $125\mu s$ is the clock synchronization precision and $\tau_{\text{bit}} = 8\mu s$ at 125 kbit/s data rate.

The comparison of a local response time *cdf* computation with the measurement under weak synchronization is given in Fig. 5.31 and Fig. 5.32 for the bad and good scenarios, respectively. In both cases, it can be seen that the resulting computed local

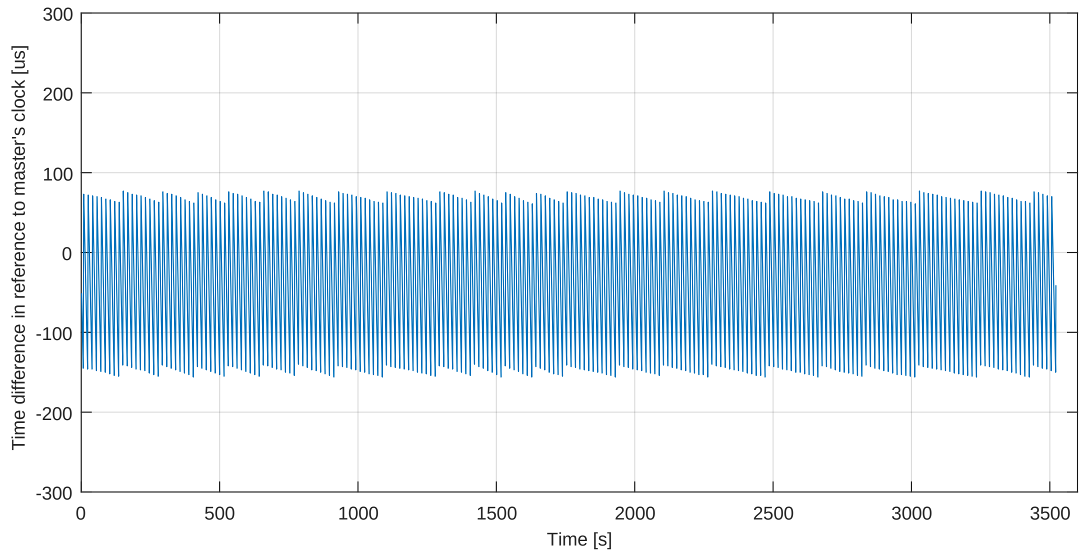


Figure 5.29: Variation of the Time Difference between Slave-2 and Master Clock under Weak Synchronization

response time distribution shows a very tight match with measured response time distributions. This is an expected result since the computations are made for phase shift boundaries which are the same as the node synchronization boundaries. Using weak synchronization, it is assured that the nodes do not drift from the initial phase scenario by more than the clock synchronization precision which is 125μs in this case.

The resulting computed local response time distribution shows a very tight match with measured response time distributions.

5.5.6 Discussion of Local Response Time Distribution and Weak Synchronization

In this section, we saw that weak synchronization provides a global clock that is synchronized with a weak precision which makes is possible to predict the phase shift boundaries between the nodes. Knowing the phase variation boundaries allows to compute local response time distributions that match the actual response time distributions.

Another important benefit of weak synchronization is that making response time distribution computations based on a bounded phase shift rather than computing over the entire period reduces the computational complexity and hence the computation times

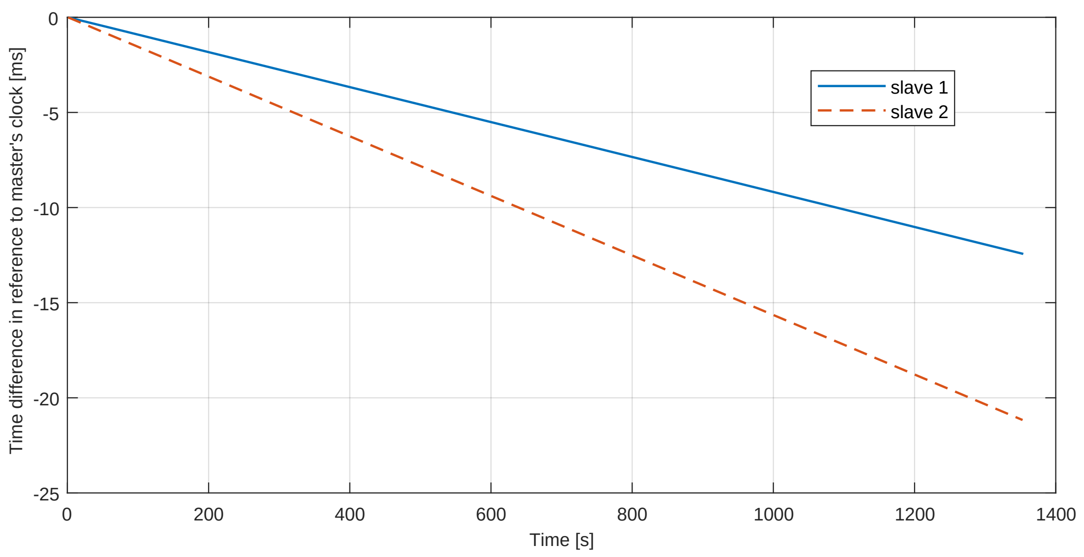


Figure 5.30: Variation of the Time Difference between Slave and Master Clock when Weak Synchronization is Disabled

considerably. In this case the complexity reduces from $\mathcal{O}(H^n \cdot k^m)$ to $\mathcal{O}(\Delta^n \cdot k^m)$ since the computation is done covering $2\Delta\Phi$ phase shifts rather than the entire hyperperiod. For instance, in the example considered in Section 5.5.5, there are 1250^2 node phasing scenarios leading to 1250^2 computations without synchronization whereas under weak synchronization this reduces to $(2\Delta\Phi)^2 = 30^2$. This also makes the computational complexity independent of the hyperperiod. As a result, the variable hyperperiod term is replaced by a constant and much smaller value.

In Chapter 3, we saw that offset scheduling is performed exclusively to each node since nodes are not synchronized. The existence of a global clock that is synchronized with a weak precision is expected to enable offset scheduling with smaller response times for the entire network which is left as a future work.

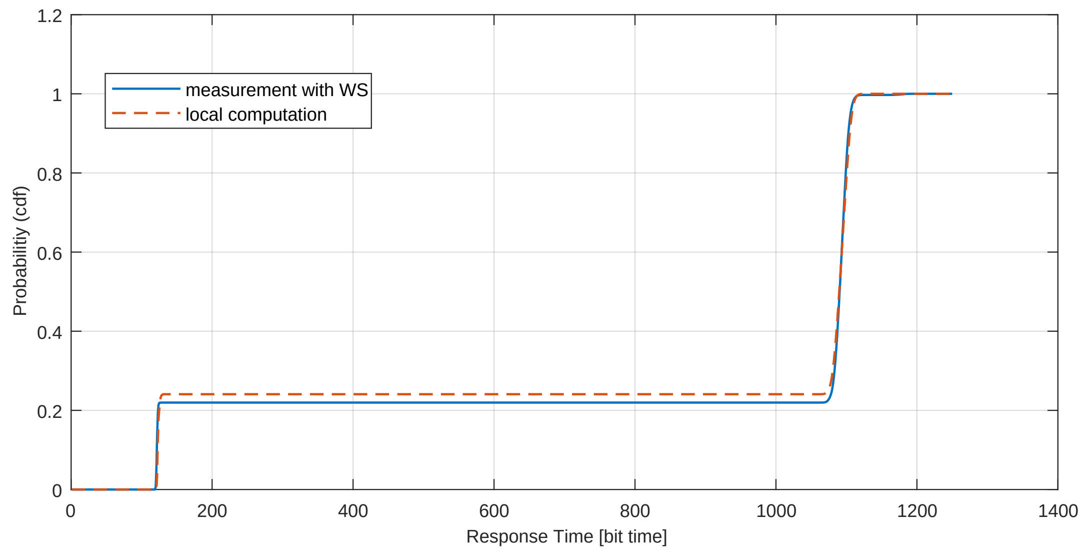


Figure 5.31: Comparison of Local Response Time *cdf* Computation and Measurement in Bad Case under Weak Synchronization

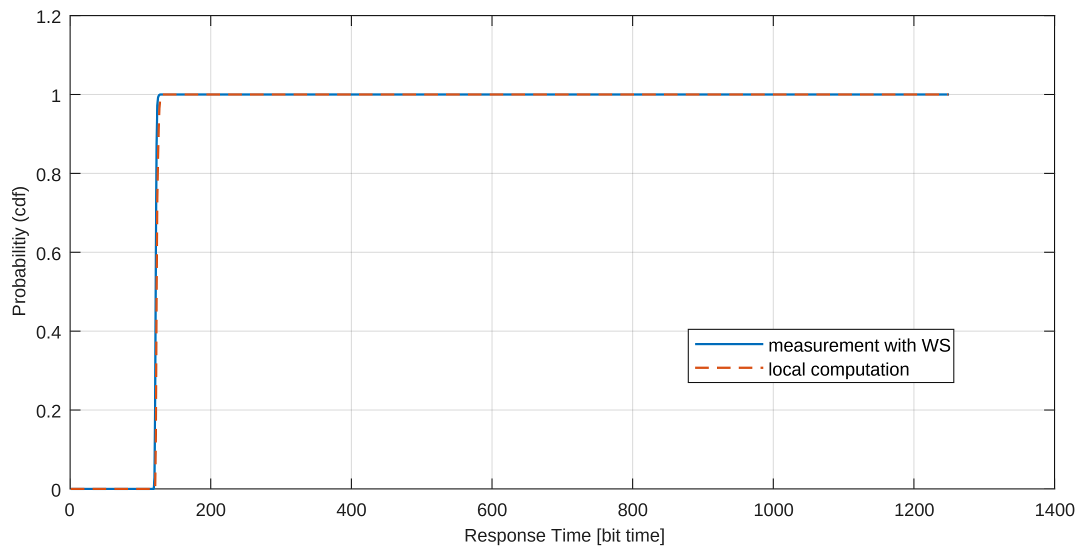


Figure 5.32: Comparison of Local Response Time *cdf* Computation and Measurement in Good Case under Weak Synchronization

CHAPTER 6

CONCLUSION

The subject of this thesis is the *offset scheduling* on the controller area network (CAN), which is the most popular in-vehicle network. The first aim of the thesis is the improvement of performance metrics such as worst-case response times (WCRTs), message slacks and the differences of WCRTs of messages with the same deadline, compared to an existing load distribution (LD) algorithm. The second aim of the thesis is the computation of the probabilistic response time distribution for messages transmitted on CAN. The response time distribution shows the variation of the message response time due to non-deterministic factors such as the message length and the changing phase difference between nodes due to clock drifts.

Regarding the first aim, the thesis proposes four new algorithms for the offset assignment on CAN. Three of the algorithms merely use information of individual network nodes and hence have computation times in the order of milliseconds. The fourth algorithm uses a neighborhood search method and leads to computation times of at most 30 min for realistic messages sets. Since offset assignments for CAN are computed offline, all algorithms are suitable for practical applications. The proposed algorithms are compared to the existing LD algorithm in comprehensive computational experiments using CAN networks with different numbers of nodes and bus loads. These experiment highlight that, although the existing LD algorithm already achieves good results, the proposed algorithms outperform this algorithm in most of the test cases.

Regarding the second aim, the thesis develops an original method for computing the response time distribution for non-preemptive systems such as CAN. The proposed method is based on the computation of a backlog distribution for the target message that is due to the interference of other messages and the execution time of the target message. Moreover, the thesis shows that the response time distribution has a local nature in the sense that it remains approximately constant for a short period of time (in the order of tens of minutes) but gradually changes due to clock drifts. As a novel contribution, the thesis defines the computation of a local response time distribution and shows that it matches well with hardware measurements. In addition, the thesis develops the new idea of enforcing a desired local response time distribution by weak synchronization of all CAN nodes. As the main result of this thesis, it is possible to keep the node phases within a small range and to perform a very exact computation of local response time distribution that does not change over time.

It has to be noted that the presented results regarding offset assignment are currently restricted to the consideration of the (deterministic) WCRT and are hence not directly applicable to probabilistic response time distributions. In turn, the results for the computation of the response time distribution assume a given offset assignment and choice of suitable node phases. In future work, a combination of the presented results is suggested. In particular, the final aim of this work is the computation of offset assignments and node phases in order to guarantee small message response times with a high probability.

REFERENCES

[1] A. Batur, K. W. Schmidt, and E. G. Schmidt. Improved load distribution for controller area network. In *2017 25th Signal Processing and Communications Applications Conference (SIU)*, pages 1–4, May 2017.

[2] A. Batur, K. W. Schmidt, and E. G. Schmidt. Offset assignment on controller area network: improved algorithms and computational evaluation. In *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–9, June 2017.

[3] Y. Chen. Real time scheduling and analysis for can messages with offsets. Master's thesis, Nagoya University, 2012.

[4] R. Davis, A. Burns, R. Bril, and J. Lukkien. Controller area network (CAN) schedulability analysis: Refuted, revisited and revised. *Real-Time Syst.*, 35(3):239–272, April 2007.

[5] R. I. Davis and A. Burns. Robust priority assignment for fixed priority real-time systems. In *IEEE International Real-Time Systems Symposium*, pages 3–14, 2007.

[6] R. I. Davis and A. Burns. Robust priority assignment for messages on controller area network (CAN). *Real-Time Syst.*, 41(2):152–180, Feb. 2009.

[7] J. L. Diaz and J. M. Lopez. Probabilistic analysis of the response time in a real-time system. Oct 2001.

[8] L. Du and G. Xu. Worst case response time analysis for CAN messages with offsets. In *Vehicular Electronics and Safety, IEEE International Conference on*, pages 41–45, 2009.

[9] M. Gergeleit and H. Streich. Implementing a distributed high-resolution real-time clock using the can-bus. *Proceedings of the 1st international CAN-Conference 94*, Sep 1994.

[10] M. Grenier, L. Havet, and N. Navet. Pushing the limits of CAN - scheduling frames with offsets provides a major performance boost. In *European Congress on Embedded Real Time Software*, 2008.

[11] M. Grenier, L. Havet, and N. Navet. Scheduling messages with offsets on Controller Area Network - a major performance boost. In *The Automotive Embedded Systems Handbook*, Industrial Information Technology Series. Taylor & Francis / CRC Press, 2008.

[12] T. Nolte, H. Hansson, and C. Norstrom. Probabilistic worst-case response-time analysis for the controller area network. In *The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings.*, pages 200–207, May 2003.

[13] T. Nolte, H. Hansson, C. Norstrom, and S. Punnekkat. Using bit-stuffing distributions in can analysis. *IEEE Real-Time Embedded Systems Workshop*, Dec 2001.

[14] K. W. Schmidt. Robust priority assignments for extending existing controller area network applications. *IEEE Transactions on Industrial Informatics*, 10(1):578–585, Feb 2014.

[15] I. Standard-11898. Road vehicles-interchange of digital information – Controller Area Network (CAN) for high-speed communication. *International Standards Organisation (ISO)*, 1993.

[16] K. Tindell, A. Burns, and A. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3:1163–1169, 1995.

[17] A. Valenzano and G. Cena. Controller area networks for embedded systems. In R. Zurawsk, editor, *Networked Embedded Systems*, pages 15–1–15–38. CRC Press, 2009.

[18] C. Yang, R. Kurachi, Z. Gang, and H. Takada. Schedulability comparisons between priority queue and FIFO queue for CAN messages with offsets. *International Journal of Automotive Engineering*, 4(4):75–82, 2013.

[19] P. Yomsi, D. Bertrand, N. Navet, and R. Davis. Controller area network (CAN): Response time analysis with offsets. In *Factory Communication Systems, IEEE International Workshop on*, pages 43–52, 2012.

[20] H. Zeng, M. Di Natale, P. Giusto, and A. Sangiovanni-Vincentelli. Stochastic analysis of can-based real-time automotive systems. *Industrial Informatics, IEEE Transactions on*, 5(4):388 –401, nov. 2009.