CLOCK SYNCHRONIZATION ALGORITHMS ON A SOFTWARE DEFINED CAN CONTROLLER: IMPLEMENTATION AND EVALUATION

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

BY

SERKAN YALÇIN

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

JANUARY 2020

Approval of the thesis:

CLOCK SYNCHRONIZATION ALGORITHMS ON A SOFTWARE DEFINED CAN CONTROLLER: IMPLEMENTATION AND EVALUATION

submitted by **SERKAN YALÇIN** 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. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. İlkay Ulusoy
Head of Department, **Electrical and Electronics Engineering** _____

Prof. Dr. Klaus Werner Schmidt
Supervisor, **Electrical and Electronics Engineering, METU** _____

Prof. Dr. Ece Güran Schmidt
Co-supervisor, **Electrical and Electronics Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Gözde B. Akar
Electrical and Electronics Engineering, METU _____

Prof. Dr. Klaus Werner Schmidt
Electrical and Electronics Engineering, METU _____

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

Prof. Dr. İlkay Ulusoy
Electrical and Electronics Engineering, METU _____

Prof. Dr. Ali Ziya Alkar
Electrical and Electronics Engineering, Hacettepe University _____

Date:

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, Surname: Serkan Yalçın

Signature :

ABSTRACT

CLOCK SYNCHRONIZATION ALGORITHMS ON A SOFTWARE DEFINED CAN CONTROLLER: IMPLEMENTATION AND EVALUATION

Yalçın, Serkan
M.S., Department of Electrical and Electronics Engineering
Supervisor: Prof. Dr. Klaus Werner Schmidt
Co-Supervisor: Prof. Dr. Ece Güran Schmidt

January 2020, 71 pages

Many advanced driver-assistance systems (ADAS) and in-vehicle applications require coordination for their safety-critical tasks. To achieve such a coordination, different electronic control units (ECUs) in the system should synchronize their clocks in order to share a global time. Although the controller area network (CAN) is the most widely used communication bus for the information exchange among ECUs, it does not support the required clock synchronization. Moreover, even several advanced clock synchronization methods for CAN have been suggested in the literature, they require modifications of the CAN driver, which is generally implemented in hardware and not accessible to modifications. The first aim of this thesis is the implementation of a software-defined CAN controller (SDCC) which enables modifications to the standard CAN driver. This SDCC is compatible to standard CAN controllers. The second aim of the thesis is the realization of new clock synchronization algorithms for CAN based on the SDCC including modifications to the classical CAN driver. The performance of the new algorithms is evaluated and compared to existing clock synchronization algorithms for CAN.

Keywords: Controller Area Network, Clock Synchronization, Software-Defined Controller

ÖZ

YAZILIM TANIMLI CAN DENETLEYİCİ TABANLI SAAT SENKRONİZASYONU ALGORİTMALARİ: UYGULAMA VE DEĞERLENDİRME

Yalçın, Serkan
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Bölümü
Tez Yöneticisi: Prof. Dr. Klaus Werner Schmidt
Ortak Tez Yöneticisi: Prof. Dr. Ece Güran Schmidt

Ocak 2020 , 71 sayfa

Pek çok gelişmiş sürücü destek sistemi ve araç içi uygulamalar, güvenlik açısından kritik görevler için koordinasyona ihtiyaç duymaktadır. Bu koordinasyonun sağlanması için farklı elektronik kontrol ünitelerinin (ECU) global bir saate sahip olmak amacıyla saatlerini senkronize etmeleri gerekmektedir. Her ne kadar Kontrol Alanı Ağı Veriyolu (CAN Bus), elektronik kontrol üniteleri arası bilgi alışverişinde en yaygın kullanılan iletişim yolu olsa da bahsi geçen saat senkronizasyonunu desteklememektedir. Literatürde birçok gelişmiş saat senkronizasyon metodu önerilmiş olmasına rağmen, bu metotlar sürücü değişikliklerini gerektirme olup, bu sürücüler çoğunlukla donanım tabanlı olduğundan bu değişikliklere izin vermemektedir. Bu tezin ilk amacı, konvansiyonel bir CAN sürücüsü üzerinde değişikliklere izin veren ve standart denetleyiciler ile uyumlu yazılım tanımlı bir CAN denetleyicisinin uygulanmasıdır. Tezin ikinci amacı ise, bahsedilen yazılım tanımlı CAN denetleyicisi üzerinde çalışan yeni saat senkronizasyon algoritmalarının geliştirilmesidir. Sonrasında ise, bu algoritmaların performansları değerlendirilecek ve mevcut CAN saat senkronizasyon algorit-

maları ile karşılaştırılacaktır.

Anahtar Kelimeler: Kontrol Alanı Ağı Veriyolu, Saat Senkronizasyonu, Yazılım Tanımlı Denetleyici

To my family

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my thesis supervisor, Prof.Dr. Klaus Werner Schmidt, and co-supervisor, Prof. Dr. Ece Güran Schmidt, for their continuous guidance and support. It was really an honor to work with them during this thesis.

I would like to acknowledge the support of my family and my fiance Selvi Deniz Dörttaş. This work would not have been possible without their valuable support and encouragement.

TABLE OF CONTENTS

ABSTRACT v

ÖZ vii

ACKNOWLEDGMENTS x

TABLE OF CONTENTS xi

LIST OF TABLES xiv

LIST OF FIGURES xv

LIST OF ABBREVIATIONS xvii

CHAPTERS

1 INTRODUCTION 1

2 BACKGROUND INFORMATION 5

2.1 Controller Area Network Protocol 5

2.2 Controller Area Network Node 7

2.3 Controller Area Network Bit Timing 9

2.4 Clock Drift 11

2.5 Clock Synchronization 13

2.6 Related Work 16

2.6.1 Current Clock Synchronization Algorithms for CAN 16

2.6.2 Software Defined CAN Controller 17

3 SOFTWARE DEFINED CAN CONTROLLER 19
3.1 Software-Defined CAN Controller Layers 19
3.1.1 Medium Access Control Layer 21
3.1.2 Physical Coding Sub-layer 27
3.1.3 Physical Medium Attachment Layer 27
3.2 Hardware Dependency of Software-Defined CAN Controller 29
3.3 Porting Software-Defined CAN Controller 30
3.3.1 Experimental Setup 30
3.3.2 Porting SDCC to Experimental Setup 31
3.3.3 Running SDCC on the Experimental Setup 33
4 SOFTWARE APPROACH TO CLOCK SYNCHRONIZATION ON CAN . 37
4.1 Gergeleit's Method 37
4.1.1 Description 37
4.1.2 Implementation on SDCC 39
4.2 ISCS 42
4.2.1 Description 42
4.2.2 Implementation on SDCC 43
4.3 Results 47
5 HARDWARE APPROACH TO CLOCK SYNCHRONIZATION ON CAN . 55
5.1 Hardware Perspective on CAN Clock Synchronization 55
5.2 Description of the PECS Algorithm 56
5.3 Implementation of the PECS Algorithm on SDCC 58
5.3.1 Results 59

6 CONCLUSION 65

REFERENCES 67

LIST OF TABLES

TABLES

Table 4.1 Clock Drift Measurement Without Clock Synchronization Algorithm 50

Table 4.2 Performance of ISCS and Gergeleit's Method on SDCC Under Different Bus Loads (with 81 ppm Clock Drift) 52

Table 5.1 Performance of Gergeleit's Method with PECS on SDCC Under Different Bus Loads (with 81 ppm Clock Drift) 61

Table 5.2 Performance of ISCS with PECS on SDCC Under Different Bus Loads (with 81 ppm Clock Drift) 61

LIST OF FIGURES

FIGURES

Figure 2.1 CAN Arbitration Loss Scenario 6

Figure 2.2 A Typical Controller Area Network Frame 7

Figure 2.3 A Typical CAN Node 8

Figure 2.4 CAN Bit Timing 9

Figure 2.5 Hard and Soft Synchronization in CAN Protocol 10

Figure 2.6 Phase Error Calculation Process During Bit Time 11

Figure 2.7 Bit Stuffing and Destuffing in CAN 11

Figure 2.8 Visualization of Clock Drift 12

Figure 2.9 Temperature Effect on Oscillator Frequency [1] 13

Figure 2.10 Offset Scheduling with Clock Synchronization 15

Figure 2.11 Offset Scheduling without Clock Synchronization 15

Figure 2.12 Main Flow of Software Synchronization Algorithms in CAN . . 15

Figure 3.1 Software-Defined CAN Controller Architecture 20

Figure 3.2 Software-Defined CAN Controller Primitives 21

Figure 3.3 SDCC MAC Layer RX Finite State Automata 22

Figure 3.4 SDCC MAC Layer TX Finite State Automata 25

Figure 3.5 Timings of PCS Layer Operations 28

Figure 3.6 Software-Defined CAN Controller Hardware Dependency 29

Figure 3.7 Experimental Setup Diagram 31

Figure 3.8 Observed SDCC Waveform 35

Figure 4.1 Gergeleit's Algorithm for Clock Synchronization on CAN 38

Figure 4.2 ISCS Algorithm Operation - Validity Flag Match 43

Figure 4.3 ISCS Algorithm Operation - Validity Flag Mismatch 44

Figure 4.4 Data Collection for Performance Measurement 49

Figure 4.5 Evaluation of the Clock Drift between the CAN nodes 51

Figure 4.6 Performance of ISCS and Gergeleit's Method on SDCC Under Different Bus Loads (with 81 ppm Clock Drift) 52

Figure 4.7 Clock Drift Comparison of ISCS and Gergeleit's Method on SDCC Under Different Bus Loads (with 81 ppm Clock Drift) 53

Figure 4.8 Clock Drift Histograms of ISCS and Gergeleit's Method on SDCC (with 81 ppm Clock Drift) 54

Figure 5.1 Valid Clock Correction Edges for PECS 57

Figure 5.2 Performance of PECS on SDCC Under Different Bus Loads (with 81 ppm Clock Drift) 62

Figure 5.3 Clock Drift Comparison of Gergeleit's Method with PECS on SDCC Under Different Bus Loads (with 81 ppm Clock Drift) 63

Figure 5.4 Clock Drift Comparison of ISCS with PECS on SDCC Under Different Bus Loads (with 81 ppm Clock Drift) 64

LIST OF ABBREVIATIONS

ADAS	Advanced Driver Assistance Systems
CAN	Controller Area Network
ISCS	Improved Software-Based Clock Synchronization
SDCC	Software-Defined CAN Controller
PECS	Phase Error-Based Clock Synchronization

CHAPTER 1

INTRODUCTION

Controller Area Network (CAN) [2] is a bus standard that is widely used in automotive applications. With more than 2 billion nodes sold [3], [4], CAN allows different electronic control units (ECUs) and components such as sensors and actuators to exchange information. After its development, it became the de facto standard in time-critical applications such as X-by-wire, advanced-driver-assistance systems and autonomous driving systems, since it guarantees known and bounded worst-case response time [5], [6]. Although the CAN bus is seemingly outdated with newer bus standards such as Flex-Ray [7], its popularity is still high due to its simplicity, low cost and reliability.

As stated before, CAN is very old compared to other bus standards. Nevertheless, its performance and functionality are still being improved with new enhancements and extensions such as [8–13]. However, since CAN controllers are implemented in hardware, adding new capabilities to existing controllers is not an easy task. In order to enable easier modifications and experiments on CAN, a "Software Defined CAN Controller (SDCC)" is proposed in [14]. SDCC is able to communicate with CAN devices through a physical bus and it is capable of real-time execution.

Many advanced driver-assistance systems and in-vehicle applications require coordination for their operations. This coordination is only possible if different electronic control units (ECUs) share a common global time perspective [15–17]. However, CAN does not support clock synchronization between the nodes. To add this capability to CAN, several clock synchronization methods are suggested. Widely known software-based methods such as [18] exploit the synchronous message reception of the CAN protocol. These methods suggest taking timestamps after the reception of

time-reference frames to ensure that timestamps are recorded concurrently. Differing from previous approaches, the Improved Software-Based Clock Synchronization (ISCS) method given in [19], makes use of each frame on the bus to improve the clock synchronization performance. Also suggested in [19], the hardware-based Phase Error-Based Clock Synchronization (PECS) uses the phase error correction mechanism of standard CAN hardware to provide more precise global clock for the nodes on the network.

In this context, it has to be noted that all of the clock synchronization methods (both software and hardware approaches) on CAN are implemented on CAN controllers that are realized in hardware up to now. Since most of the advanced clock synchronization methods require modifications on standard CAN controllers, their implementation relies on changes in the CAN controller hardware, which is time consuming and requires very specific expertise. Accordingly, implementing those algorithms on SDCC significantly increases the practicality of modifications on the CAN protocol. Specifically, the easier experimentation on CAN controller layers with SDCC will enable faster development of new clock synchronization algorithms.

The first aim of this thesis is to implement SDCC on different microcontrollers and establish the communication between them via a physical bus. The second aim of the thesis is the realization of clock synchronization algorithm using the SDCC. Hereby, the first step is to validate the functionality of the SDCC by implementing and evaluating the different software-based clock synchronization algorithms in [18] and [19]. The second step is the usage of the SDCC for modifications of the CAN protocol. To this end, the hardware-based PECS algorithm in [19] is implemented by using the bit-timing information of the SDCC. The correct functionality of all clock synchronization algorithms is established and the improvements of the ISCS algorithm (software-based) and PECS algorithm (hardware-based) are shown by practical experiments. In summary, the main contributions of the thesis are as follows:

- Implementation of the SDCC and deployment on different platforms,
- Realization of the software-based clock synchronization algorithms for CAN in [18] and [19],

- Realization of the hardware-based clock synchronization algorithm in [19],
- Experimental evaluation and comparison of the implemented clock synchronization algorithms.

The remainder of the thesis is organized as follows. In Chapter 2, the CAN protocol, the concept of clock drift and clock synchronization are explained to provide the necessary background. Following that, existing clock synchronization algorithms for CAN are examined. After that, the SDCC is explained in detail in Chapter 3 and the steps followed for porting it to the experimental setup are illustrated. Subsequently, in Chapter 4 and 5 the clock synchronization algorithms in [18], [19] are described and implemented on SDCC. Consequently, the performance of the implemented algorithms is evaluated and compared. Conclusions are given in Chapter 6.

CHAPTER 2

BACKGROUND INFORMATION

This section gives the necessary background information for this thesis. Section 2.1 provides a detailed description of the CAN protocol and Section 2.2 outlines the relevant components of a CAN node. In Section 2.3, the bit timing on the CAN bus is explained. Then, Section 2.4 introduces the concept of clock drift. The idea of clock synchronization is introduced in Section 2.5 and Section 2.6 gives an overview of related work for this thesis.

2.1 Controller Area Network Protocol

Controller Area Network (CAN) is a well-known bus standard that establishes shared medium communication between multiple controller units. It allows multi-master priority-based bus access, with non-destructive contention-based arbitration [2]. Although it can be used in any system for communication, it is widely used in safety-critical systems, especially in the automotive industry. Despite the fact that CAN is a very old communication protocol, its robustness and proven reliability still makes it a viable solution for in-vehicle and industrial networks [20–22].

Unlike many other communication protocols, CAN is not based on a master/slave structure. Instead, it defines a strict contention-based arbitration that governs the transmission and reception on the bus. To achieve this, the CAN protocol uses the identifier field that is embedded to all CAN frames. In addition to its use for recognition of the frames on the bus, the identifier field plays a big role in the arbitration of the frames. The CAN protocol states that, when multiple nodes start transmitting, the frame with the smaller identifier will be granted access to the medium. In order

to achieve this, all nodes monitor the bus while they are transmitting a frame. Since CAN defines the low voltage level as dominant and the high voltage level as recessive, the transmission of a low and high bit at the same time results in a low voltage level on the bus. Therefore, when multiple nodes are transmitting, the node with the smaller identifier will eventually set the voltage level to dominant, while the other nodes try to set it as recessive. Once nodes monitor this, the ones with the bigger frame identifier will stop transmitting and wait until the bus becomes idle again. With this arbitration method, prioritization of the frames is achieved without destroying the frames that lost the arbitration. The arbitration process is visualized in Figure 2.1 for better understanding.

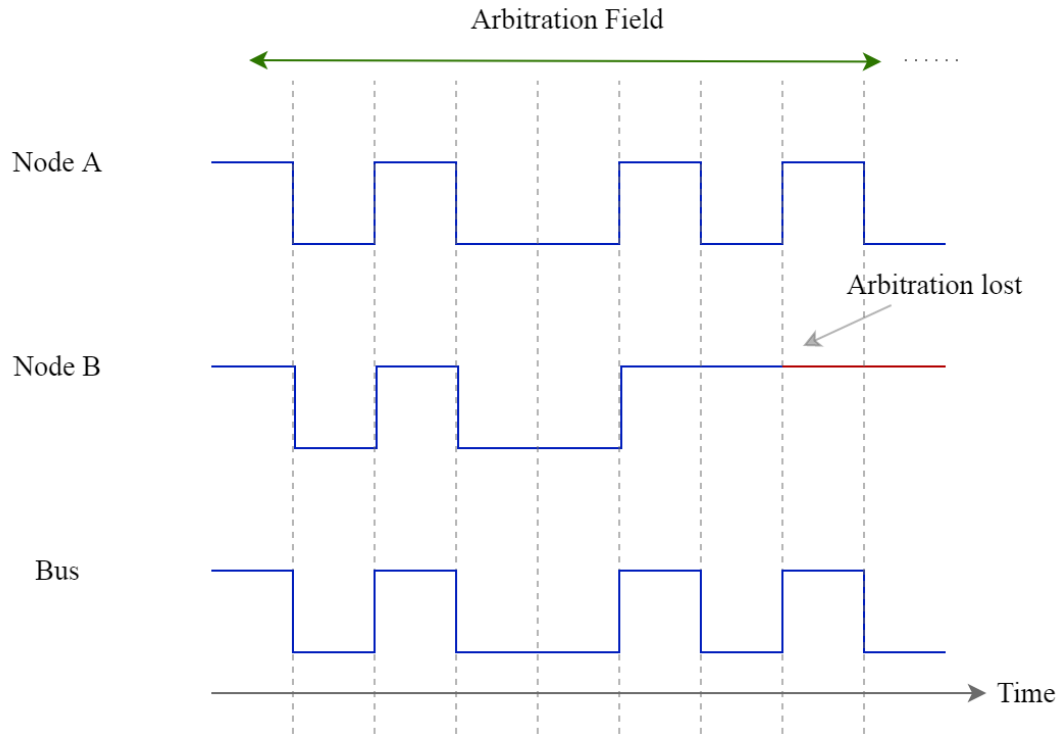

Figure 2.1: CAN Arbitration Loss Scenario

As is illustrated in Figure 2.2, a typical CAN frame starts with the Start-Of-Frame (SOF) field that is used for hard synchronization which will be discussed in following sections. Following the SOF bit, there is an 11 bit or 29 bit identifier which is used for recognizing the frames and performing the contention-based arbitration. The remote transmission request bit (RTR) is used for requesting a frame with a specific identifier. The identifier extension bit (IDE) is used for determining if 29 bit extended identifier

will be used instead of 11 bit. Following the reserved bit r, the data length code specifies the number of bytes in the payload that is placed in the data field of the frame. After the payload, the cyclic redundancy check (CRC) field is used for checking data integrity of the frame. Finally, the acknowledgement bit (ACK) is used for checking if the transmitted frame is received with success. To perform this, the transmitter sends a recessive bit during acknowledgement slot. When a receiving node decides that the frame is received with success without any error, it transmits a dominant bit which overrides the recessive bit sent by the transmitter. When the transmitter monitors the dominant level during this slot, it infers that the frame is received successfully at least by one node in the network. Here, it has to be noted that the transmitter cannot tell if all nodes received the frame successfully, since even just one dominant level on the bus will indicate acknowledgement of the frame. After the frame is completed, the End-Of-Frame (EOF) is transmitted and the bus becomes idle again. Nodes should wait for a predefined inter-frame space to start transmitting once again.

Figure 2.2: A Typical Controller Area Network Frame

2.2 Controller Area Network Node

Although CAN nodes can be considered as a single component from a high level perspective, it is possible to examine them in three different sub-layers. As it is visualized in Figure 2.3, a typical CAN node consists of a microcontroller, a CAN controller and a CAN transceiver. In this section, these layers will be described briefly.

As in many of the embedded computer systems, microcontrollers are the components where the main software is embedded. With application software, the microcontroller controls the underlying hardware to transmit and receive data from the CAN bus. Most of the time, the software block that performs CAN related operations is a small part of the application software, since the microcontroller controls many peripherals such as sensors and actuators to generate data to be transmitted via CAN frames.

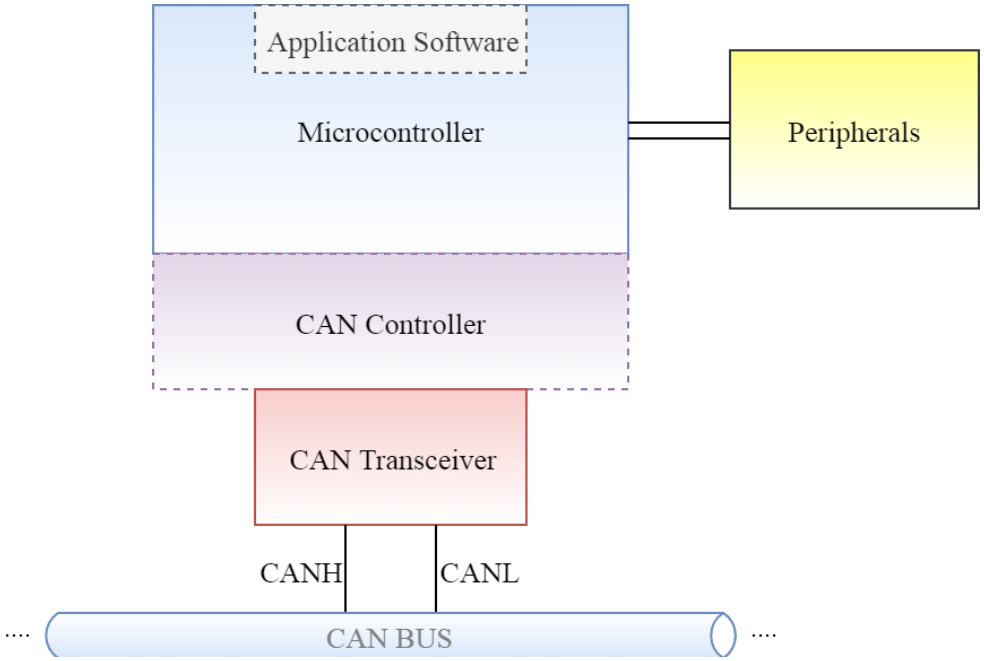

Figure 2.3: A Typical CAN Node

Microcontrollers interact with the CAN bus by using a CAN controller hardware. CAN controllers may be embedded into the microcontroller or they can be connected as a separate integrated circuit. Embedded controllers are memory mapped and can be accessed by the microcontroller directly, whereas external controllers can be accessed with serial communication protocols such as Inter-integrated Circuit (I^2C) or Serial Peripheral Interface (SPI). Whether a CAN controller is embedded or connected externally, its responsibility is to provide an interface for transmitting and receiving CAN frames. When application software wants to send a CAN frame, it writes the payload to a CAN controller register. Then, the CAN controller serializes the data and handles the transmission process. When the CAN controller receives a frame, it assembles the frame by performing de-serialization on the received bits. Depending on the application, it may trigger an interrupt or it provides a register that can be polled to decide if a frame is received. To perform transmission and reception, the CAN controller handles arbitration, bit-stuffing, edge detection, hard/soft synchronization and other protocol specific operations specified in [2].

The last layer of a CAN node is the CAN transceiver. As in the case with CAN controller, the CAN transceiver can be embedded into the microcontroller or it can be

connected externally. The CAN transceiver provides an interface between the CAN controller and the physical CAN medium (cable). It establishes proper signalling on the bus according to the data sent by the CAN controller. Also, when the node is receiving, it converts the CAN voltage levels into bits which can be processed by the controller.

2.3 Controller Area Network Bit Timing

According to [2], the CAN bit time is divided into four separate time segments, given in Figure 2.4. The synchronization segment (SYNC) is used for synchronization of local CAN clocks. This is the part, where edges are observed on the bus. The propagation segment (PROP) is used for tolerating the propagation delay on the bus and processing delay of the nodes. Phase segment 1 and 2 (PHASE1 and PHASE2) are used for soft synchronization, which will be covered in this section. These segments are lengthened or shortened depending on the calculated phase error by the nodes. Moreover, the data on the bus is sampled at the end of PHASE1.

Each segment consists of time quanta (TQ), which are constant time periods calculated according to the CAN system clock. Depending on the application and physical properties of the network, the number of time quanta in the segments can be modified. However, the synchronization segment is always 1 time quantum long and it cannot be changed.

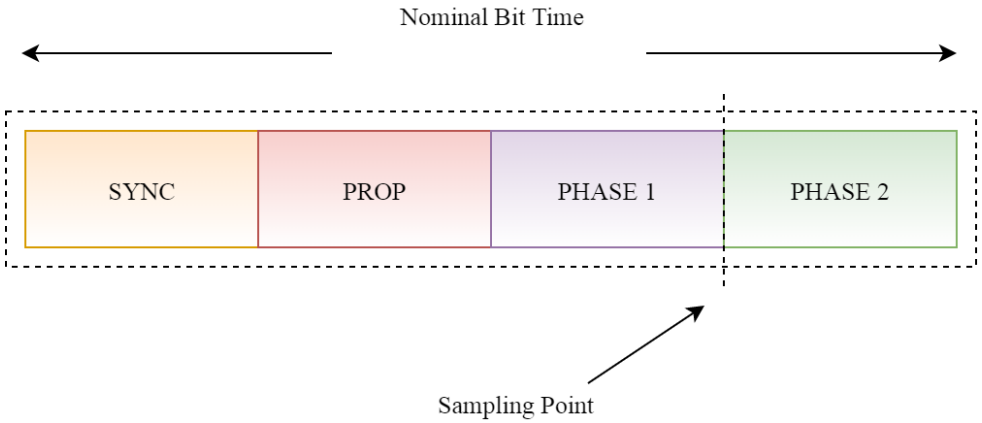

Figure 2.4: CAN Bit Timing

To compensate phase differences between CAN system clocks, phase error correction is performed as part of the CAN protocol. This correction is performed in two different ways. The first one is called hard synchronization which is performed after the Start-Of-Frame (SOF) bit is transmitted on the bus. When SOF is observed on the bus, each node resets its bit time. Hence, hard synchronization is performed at the beginning of each CAN frame.

The second synchronization method is called soft synchronization. Unlike hard synchronization, soft synchronization is performed while a frame is being received, given in Figure 2.5. When a receiving node detects a high to low transition (edge) on the bus, it calculates the phase error e_p as the difference between the quantum, where it observes the edge and time quantum of the synchronization segment (if the node's local clock would be perfectly synchronized with the sender CAN node, the edge should be detected right in the SYNC segment with phase error $e_p = 0$). Therefore, if the edge is observed before the SYNC segment, the phase error is calculated to be negative. If the edge is detected after the SYNC segment, then the phase error becomes positive. After phase error calculation illustrated in 2.6, Phase segment 1 is lengthened by e_p if e_p is positive and Phase segment 2 is shortened by e_p if e_p is negative.

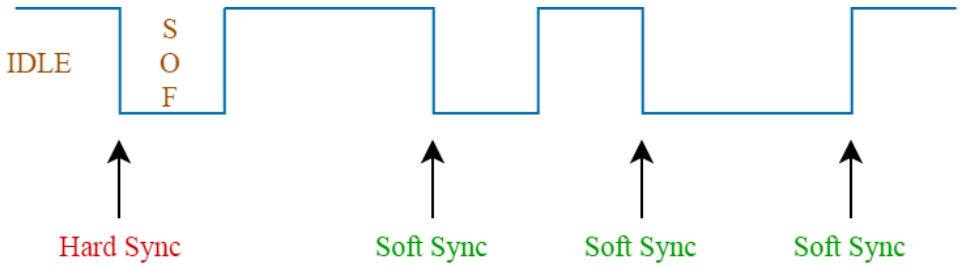

Figure 2.5: Hard and Soft Synchronization in CAN Protocol

To sustain correct bit timings, CAN nodes shall perform frequent soft synchronizations. Since soft synchronization requires a transition between opposite polarities, consecutive bits with the same polarity should be avoided. To prevent such situations, the CAN protocol uses the bit stuffing mechanism visualized in Figure 2.7. When a node is transmitting a CAN frame, it inserts (stuffs) a bit with opposite polarity after five consecutive bits with the same polarity are sent. On the receiver side, the stuffed

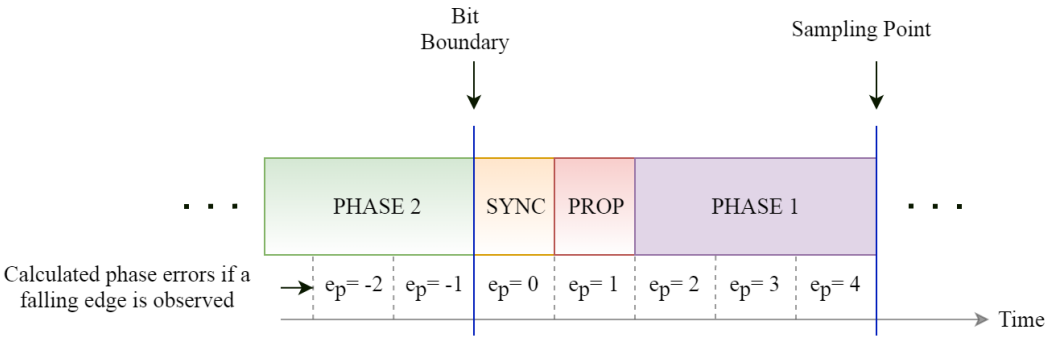

Figure 2.6: Phase Error Calculation Process During Bit Time

bits are discarded (de-stuffed) and in this way, the original frame is recovered.

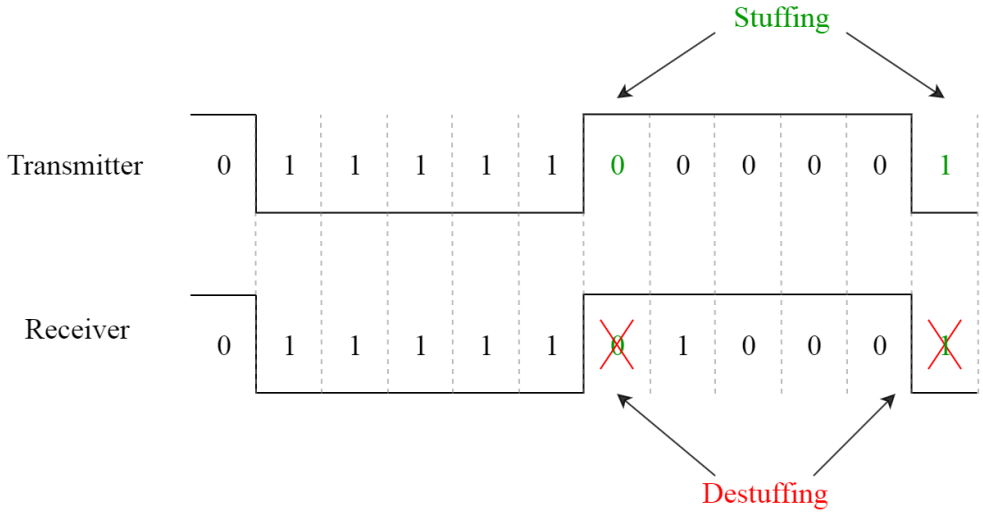

Figure 2.7: Bit Stuffing and Destuffing in CAN

2.4 Clock Drift

It is widely known that clocks are essential building blocks of electronic systems. They are used for performing synchronous operations and calculations within the electronic control units. The simplest hardware design of a clock consists of an oscillator and a counter [23]. Since frequencies of these oscillators can easily be affected by external and internal factors, even clocks built with same oscillators deviate from each other over time. As it is visualized in Figure 2.8, this phenomenon is called clock drift and it can be observed in all electronic systems with clocks.

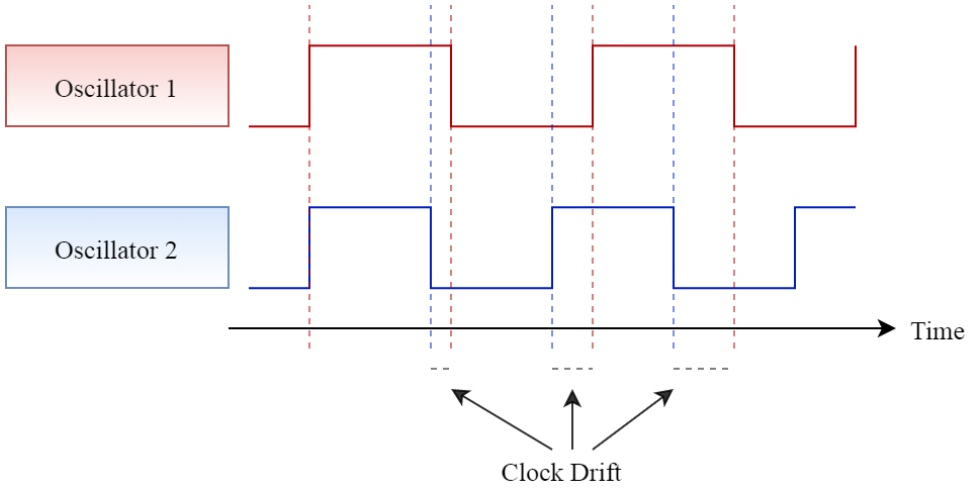

Figure 2.8: Visualization of Clock Drift

Denoting the time value of a clock as C and real time as t, if $dC/dt < 1$, the clock can be considered as slow whereas if $dC/dt > 1$, it is said to be fast compared to the real time [24]. Therefore, oscillators with different values of dC/dt, that is, different frequencies, deviate from each other over time. This difference between oscillator frequencies is due to fabrication tolerance, aging, temperature, humidity and vibration [25]. The most common of these reasons is the temperature, because temperature change is inevitable during operation of an electronic system. Since most of the time this change does not happen homogeneously, frequencies of the oscillators within the system deviate from each other. The effect of temperature on the frequencies of several oscillators is visualized in Figure 2.9.

As it is given in Figure 2.3, CAN nodes are operated by microcontrollers. Since these controllers use hardware clocks, CAN nodes on a network experience clock drift. Even if the same controllers are used and environmental conditions that affect clock frequency are eliminated, clock drift between CAN nodes cannot be avoided. This is because it is not possible to produce identical oscillators with current production methods. For example, quartz oscillators used in automotive applications commonly have tolerances of ± 50 ppm (parts per million) for fabrication and ± 5 ppm per year for aging [25]. That means, quartz crystals belonging to identical microcontrollers may deviate about 50 μs per second and this deviation may increase by 5 μs per year. Moreover, additional ± 150 ppm of clock drift may be observed depending on the

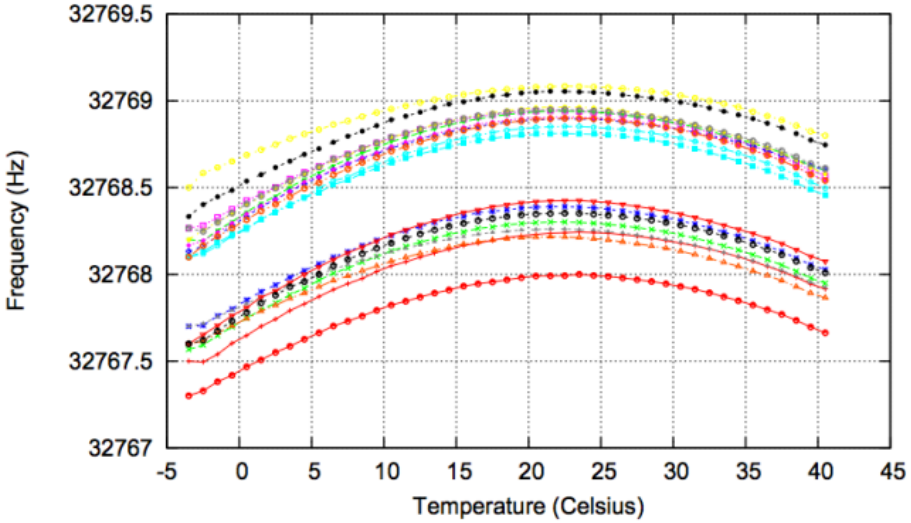

Figure 2.9: Temperature Effect on Oscillator Frequency [1]

temperature in the operating temperature range (-40/125°C) [25].

Although CAN provides a phase error correction mechanism within the bit timing as described in Section 2.3, CAN does not support clock synchronization between the nodes. Hence, as a result of imperfections in the system clock and lack of clock synchronization in CAN protocol, nodes lose the common perception of time after long periods of time, even if they are initially synchronized. It may seem that it does not create a problem since classical CAN networks are event-triggered. However, considering the recent developments in the automotive technology including advanced driving assistance systems (ADAS) and autonomous driving applications, synchronized CAN nodes are needed to ensure the performance and safety of such systems.

2.5 Clock Synchronization

In a system where more than one node is present, to calculate the time of occurrence of events and the duration between them, a common time reference is needed [26]. In order to establish such common time reference, a global timebase with bounded precision should be present in the system. Although providing a global timebase seems easy, because of the clock drift phenomenon explained in Section 2.4, addi-

tional methods shall be deployed.

The common solution for providing a global timebase is to use a hardware clock to create a virtual clock that each node accesses locally and then synchronize all virtual clocks in the system [27], [28]. By this way, without providing an actual clock that operates globally, each node possesses a virtual clock that serves as a global clock.

Based on this general idea, clock synchronization methods can be divided into two categories: external synchronization and internal synchronization. External clock synchronization maintains the difference between processor and external time reference within a bounded interval, whereas internal clock synchronization performs the same for the difference between node processor clocks [29]. The goal in both external and internal synchronization algorithms is to design an agreement protocol that keeps the clock as close as possible to real time or among all nodes by issuing periodical time value exchange [30]. To have such an exchange in the system, each node shall issue periodic events where they share their local virtual times. Consequently, the nodes correct their clocks according to the synchronization protocol deployed in the system.

As it is stated in Section 2.4, advanced driving applications require time-triggered communication on CAN. In addition, there are protocol extensions of CAN that benefit from synchronized clocks. For example, offset scheduling [8] introduces the idea of adding offsets to frame transmissions to reduce response times. However, using such offsets for scheduling requires synchronized clocks which CAN does not support. As it can be seen from Figure 2.10 and 2.11 clock synchronization is crucial in such situations. Therefore, it can be stated that global timebase is mandatory in certain CAN applications or extensions.

Clock synchronization methods in CAN can be examined in two approaches: software and hardware. Software approaches implement clock synchronization algorithms on software that controls CAN hardware, whereas hardware approaches modify the actual CAN controller to add synchronization capabilities. Although both hardware and software based clock synchronizations on CAN are challenging in some respects, synchronous reception and atomic broadcast properties of CAN are very useful [31]. When transmission of a CAN frame is completed, all nodes including the

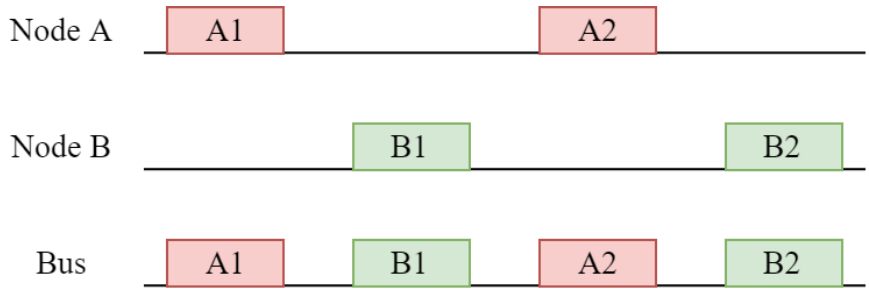

Figure 2.10: Offset Scheduling with Clock Synchronization

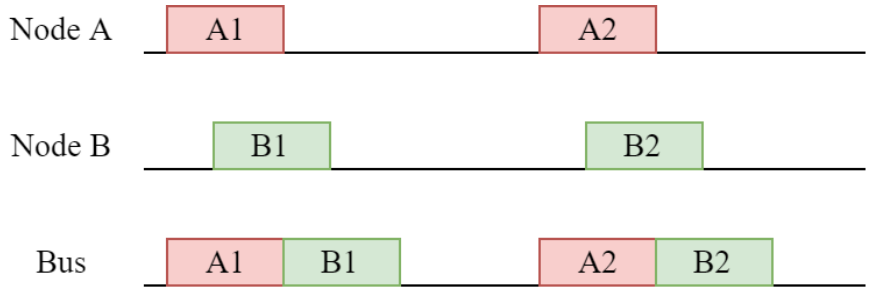

Figure 2.11: Offset Scheduling without Clock Synchronization

transmitter (since it performs bus monitoring) receive the frame simultaneously. Additionally, CAN frames are considered to be either received successfully by all nodes or they are not received by any of the nodes. Hence, most of the existing hardware and software solutions exploit these two properties.

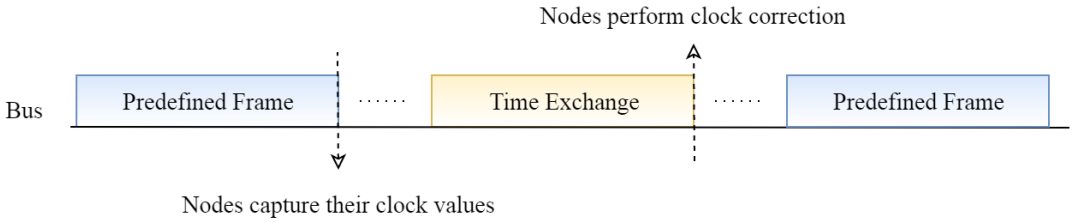

Figure 2.12: Main Flow of Software Synchronization Algorithms in CAN

Although the implementation and algorithm change in software based CAN clock synchronization solutions, the main idea visualized in Figure 2.12 stays the same. Triggered by the reception of a periodically recurring predefined frame, each node captures its clock value. Subsequently, these values are exchanged with a CAN frame among the nodes according to the synchronization algorithm. Following that, each node corrects its clock accordingly. Regarding hardware solutions for clock synchro-

nization, it has to be noted that even though the same idea can be used, it is hard to find a generalized approach since direct modification of the controller enables different methods when hardware approach is taken.

2.6 Related Work

2.6.1 Current Clock Synchronization Algorithms for CAN

Although the simplest idea for clock synchronization on CAN is taking a timestamp and broadcasting it right away, it cannot be considered as a viable solution. This is because the exact time between frame generation and end of transmission cannot be determined in CAN since a frame may be blocked by a higher priority frame. Also, although the payload size can be kept fixed, stuffed bits change the length of the CAN frame constantly. Therefore, to add time synchronization support to standard CAN protocol by avoiding such situations, several software and hardware approaches are suggested in the literature.

One of the best known software approach given in [18], uses the synchronous reception and atomic broadcast property of CAN protocol. It introduces a Time Master that periodically sends time reference frames to Time Slaves that are present in the network. Moreover, it ensures all nodes to record their local times at the same instant by suggesting taking a timestamp after each reference frame. Implementation of this method on a 32-bit microcontroller in [32], shows that average of 4.5 microsecond precision can be achieved. Note that this precision is achieved by using very small bandwidth by sending only 10 reference frames in one second. In order to make this method fault-tolerant, [33] suggests assigning different masters in each round instead of having one dedicated master. By this way, clock synchronization can still be provided in some extent even if one of the time masters fails. However, system will not be able to perform clock synchronization during the period that faulty node is expected to act as a time master. To solve this problem, [34] suggests a new master selection technique that involves partition of nodes into three groups: substitutes, master candidates and slaves. In this method, when one of the master candidate is faulty another master is selected from the substitutes.

Differing from the method in [18], [19] suggests using all frame receptions for collecting timestamps instead of taking timestamps after reference frames only. Additionally it introduces a flag that is used to determine the validity of the reference frame for the situations where the reference frame is blocked by another CAN frame. It is stated that this method increases the performance by 40% compared to the method in [18] without increasing the frequency of reference messages. Another software approach given in [35] is built on the AUTOSAR platform. It suggests a two-step mechanism where two different frames are used for calculating relative time in Time Slave nodes.

In addition to software approaches, some hardware approaches that are based on modifying the standard CAN controller are present in the literature. For example, in [31] additional hardware modules called clock units are implemented and integrated into a CAN node. These clock units consist of global clock, synchronization and CAN modules where each one of them has different responsibilities to provide a clock synchronization in the network. Similarly, [36] implements different clock units in hardware which is able to achieve 10 microsecond accuracy with low overhead. Additionally, TTCAN level 2 [9] introduces a frequency divider which is fed by clock drift information obtained from the network according to time value exchange between time master and slaves. To provide fault tolerance, it suggests master selection by using priority based arbitration of CAN protocol. Here, it has to be noted that the drawback of most hardware approaches is that either they require complex hardware extensions, or they are not compatible with standard CAN controllers. This is the reason why software approaches are preferred more frequently.

2.6.2 Software Defined CAN Controller

Since CAN controllers are most commonly implemented in hardware, they are not accessible to modifications. Therefore, even a small extension to the current CAN protocol or an experimentation of a new feature requires tedious work on the hardware design because of the complexity of CAN controllers. To be able to eliminate hardware dependency and provide easier experimentation on CAN, a software defined CAN controller (SDCC) [14] is implemented in C programming language [37].

The design in [14], consists of three different layers: medium access control layer

(MAC), physical coding sub-layer (PCS) and physical medium attachment (PMA). The low-level layer of SDCC, PMA provides the required interface to general purpose I/O pins (GPIO) and a timer module which will be used as the system clock of the software defined controller. As a result of that, it is possible to port [14] to any system with a microcontroller that contains a timer and controllable GPIOs.

A possible drawback of the SDCC is that most of the functions provided to the application layer software are time-critical. Hence, algorithms that require long computations will block lower level functions that are responsible for bus synchronization. Therefore, execution of complex algorithms on the application layer may end up with a loss of bus synchronization which can lead to erroneous reception or transmission of a frame. Although this situation can be solved by decreasing the CAN bit rate by modifying SDCC software, it is undesired since bit rate achieved by SDCC is already low compared to standard hardware CAN controllers.

CHAPTER 3

SOFTWARE DEFINED CAN CONTROLLER

This chapter describes the software-defined CAN controller (SDCC) adopted in this thesis. Section 3.1 gives an overview of the layers used in the SDCC and Section 3.2 points out which parts of the SDCC depend on the hardware where the SDCC is deployed. The method of porting the SDCC to the specific hardware used in this thesis is explained in Section 3.3.

3.1 Software-Defined CAN Controller Layers

SDCC [14] is a Controller Area Network controller that is implemented in software instead of hardware that most controllers are based upon. It is designed to establish communication between nodes that are connected by the CAN bus without any hardware controller support. SDCC uses a layered software approach, where each layer is responsible for different operations that are well-defined in the CAN standard [2]. Those layers can be listed as:

- Medium Access Controller Layer (MAC)
- Physical Coding Sub-Layer (PCS)
- Physical Medium Attachment Layer (PMA)

The SDCC layers shown in Figure 3.1 provide a consistent interface, where upper and lower layers are linked together to share information related to CAN frames that are transmitted or received. Also, they share common information for their operations with the help of data structures where each layer accesses states, variables and errors

etc. Additionally, unlike hardware controllers, it is possible to access this valuable information from the application software.

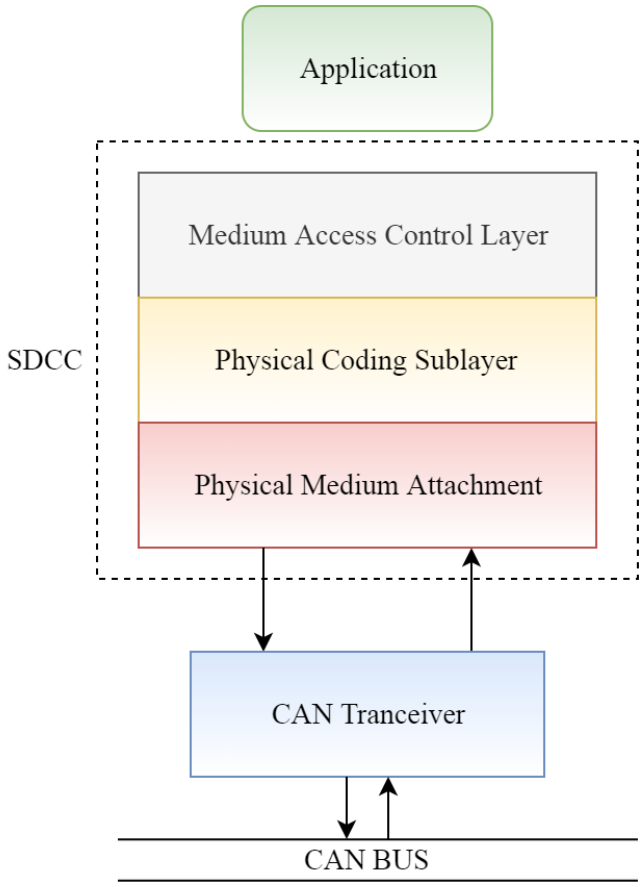

Figure 3.1: Software-Defined CAN Controller Architecture

As can be seen from Figure 3.2, SDCC uses indications, confirmations and requests called "primitives" to invoke or notify upper and lower layers. Indications are sent from lower layers to upper layers in case of node clock ticks and data reception. Confirmations and requests are sent to lower layers for transmission request and data confirmation. Note that the application using SDCC can use requests for sending frames, indications and confirmations for frame reception and notification. As a result, the user can send and receive CAN frames without going into much detail. This is the advantage of the layered software design.

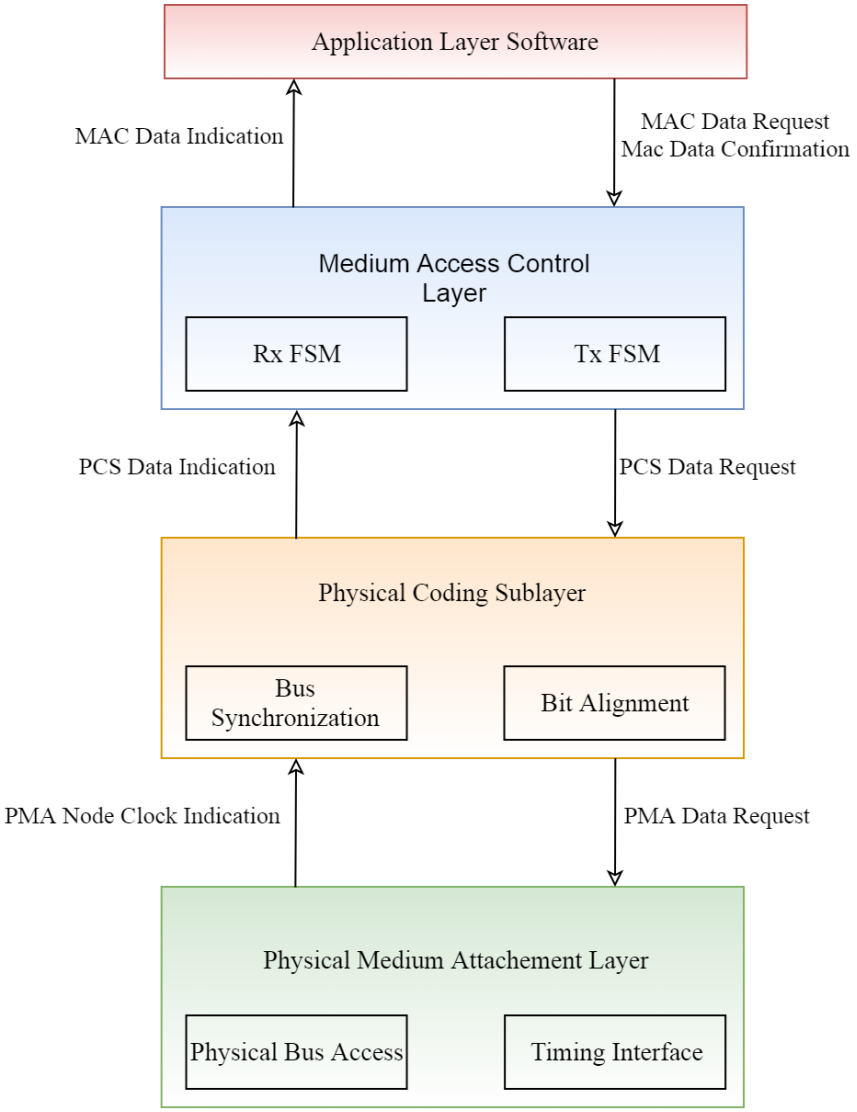

Figure 3.2: Software-Defined CAN Controller Primitives

3.1.1 Medium Access Control Layer

The Medium Access Control Layer performs frame level operations such as serialization & deserialization, bit stuffing & destuffing, bus arbitration, CRC control and error detection. Although it is the most complex of the three layers, because of its closeness to the application layer, it is easier to understand from user perspective. Since CAN reception and transmission have states that change the operation of the controller layers, SDCC implements two finite state machines called RX_FSM and TX_FSM for this purpose. These state machines, use the indications coming from the PCS layer to advance between states. As can be seen from Figure 3.3 and 3.4, most

of these states directly correspond to CAN frame fields.

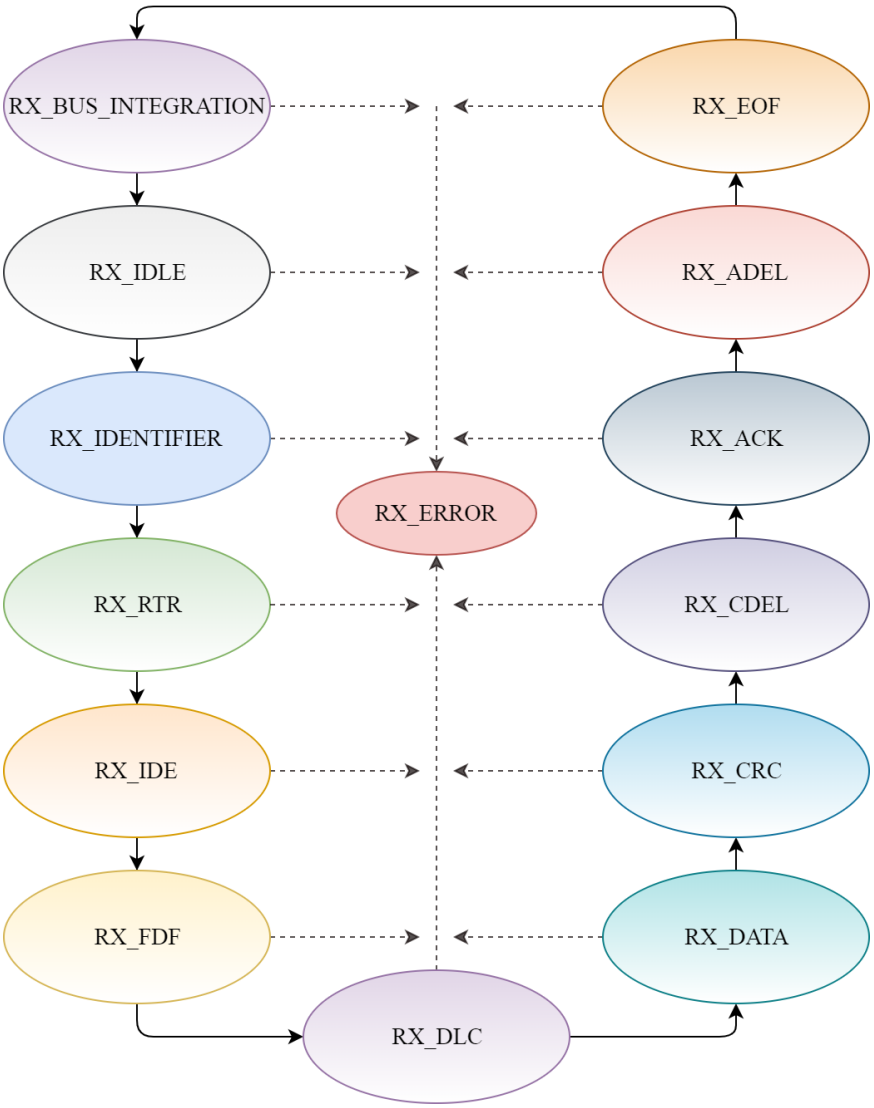

Figure 3.3: SDCC MAC Layer RX Finite State Automata

RX_FSM is invoked when PCS generates a data indication for the MAC layer after the bus level is sampled at the sampling point. MAC processes the bus level according to the current RX_FSM state, performs necessary state transition and waits until the next PCS data indication. The procedures followed in each RX_FSM state are summarized below:

- **RX_BUS_INTEGRATION**: In this state, bus integration is performed. To achieve this, the state machine simply counts 11 consecutive recessive bits before declaring the bus as idle. If a dominant bit is observed in this state, the

recessive bit counter is reset. When bus integration is completed, the RX_FSM state is updated as RX_IDLE. Note that this state is necessary to establish a frame level synchronization.

- **RX_IDLE**: RX_FSM stays at this state until a dominant bit representing SOF is received. When SOF is observed on the bus, the de-stuffing routine is initialized to discard future stuffed bits. Following that, CRC calculation is started and RX_FSM state is updated as RX_IDENTIFIER.

  In this context, note that the MAC layer performs de-stuffing by counting consecutive bits with the same polarity. When this counter reaches five, the MAC layer expects the arrival of a bit with opposite polarity. If it receives a bit with the same polarity, RX_FSM state is set as RX_ERROR.

- **RX_IDENTIFIER**: In this state, incoming bits are shifted in for constructing the identifier field. When 11 identifier bits are received, the state is set as RX_RTR. The identifier value is also stored as an integer, to be used when MAC data indication is generated.

- **RX_RTR**: SDCC does not support RTR frames. Hence in this state, received bit is unchecked and stored directly, then state is updated as RX_IDE.

- **RX_IDE & RX_FDF**: Since SDCC supports only classical base format frame, bits received during these states are checked to be dominant. If a recessive bit is received, the state is updated as RX_ERROR. Otherwise, the state is advanced to RX_DLC.

- **RX_DLC**: During this state, DLC bits are shifted in until all 4 DLC bits are received by the MAC layer. According to the constructed DLC field, the size of the data field is calculated and the counter for the data field is initialized accordingly. Following that, the state is updated as RX_DATA.

- **RX_DATA**: This is the state where payload is received from the bus. The counter that is set during RX_DLC state is used to receive the whole payload. The received bits are shifted in and each time a byte is completed, it is copied to a byte array to be passed to the user as a pointer once the frame is received successfully.

- **RX_CRC**: During RX_CRC state, CRC calculation is not stopped. Instead, the calculation is continued until all CRC bits are received. Once all CRC bits are received, the calculated CRC is checked to be zero in order to be sure that the CRC field is correct. By this way, CRC is not stored and checked on the calculation process itself. If the CRC is found to be correct, the state is updated as RX_CDEL. Otherwise, a transition to RX_ERROR is performed.

- **RX_CDEL**: The bit received during this state is checked to be recessive. If it is not, a transition to RX_ERROR is triggered. Otherwise, the acknowledgement phase is started by advancing to RX_ACK state.

- **RX_ACK**: The acknowledgement bit is checked to be dominant to be sure that nodes in the network received the frame successfully. If the frame is received successfully, the state is advanced to RX_ADEL, otherwise state is changed to RX_ERROR.

- **RX_ADEL**: As in the case with CDEL, the bit received during this state is checked to be recessive and the state is advanced accordingly.

- **RX_EOF**: Finally, RX_EOF state is the state where 7 recessive bits indicating the end of frame are expected to be received. If a dominant bit is observed during this period, a transition to RX_ERROR performed. If the EOF field is completed without an error, it means that a CAN frame is received successfully. In that case, a MAC data indication is sent to the upper layer to notify that a frame is received. This indication contains the identifier, DLC and format fields of the received frame as well as the payload.

- **RX_ERROR**: The error state is responsible for releasing the bus by calling a PCS data request with a recessive bit and resetting RX_FSM and TX_FSM states to RX_BUS_INTEGRATION and TX_ERROR respectively.

As in the case with RX_FSM, TX_FSM is also invoked by the PCS data indication. After the MAC layer receives a bit from PCS and handles RX_FSM accordingly, it starts to advance TX_FSM for performing frame transmission. The steps followed in each TX_FSM state are given below:

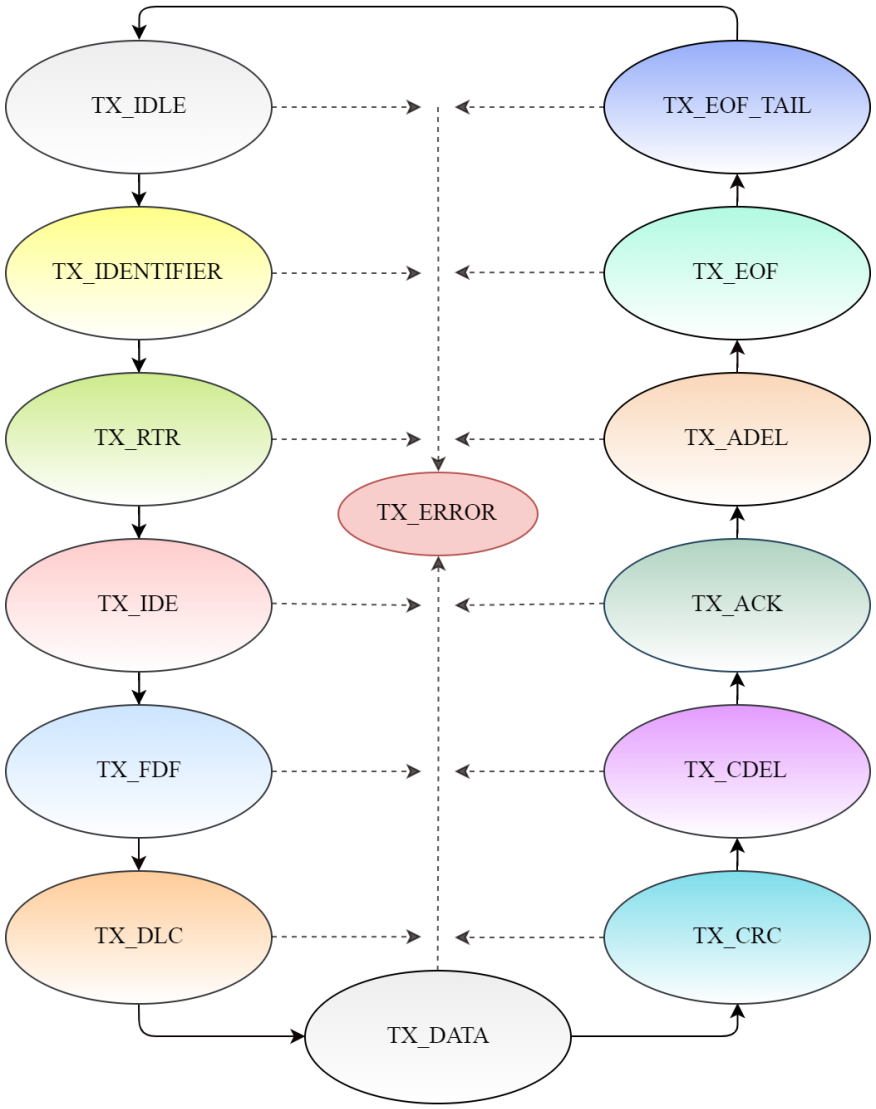

Figure 3.4: SDCC MAC Layer TX Finite State Automata

- **TX_IDLE**: In this state, the MAC layer checks if a data transmission is pending by reading a flag which is set during data request calls. If there is a pending request and RX_FSM state is RX_IDLE, which means the node is not receiving a frame, PCS data request is triggered to send a dominant bit that corresponds to SOF. Following that, the state is set as TX_IDENTIFIER.

- **TX_IDENTIFIER**: In this state, the identifier of the frame is shifted and sent one by one with PCS data requests. After the identifier field is finished, the state is advanced to TX_RTR.

  Here, we note that after a node starts transmitting the identifier, the MAC starts

stuffing bits when it is necessary. As in the de-stuffing case in RX_FSM, it uses a counter to count consecutive bits with the same polarity. Once this counter becomes five, it immediately generates a PCS data request to send a bit with opposite sign. Until the stuffed bit is transmitted, TX_FSM state is preserved and no operation related to the current frame is performed.

- **TX_RTR, TX_IDE, TX_FDF**: Since SDCC does not support remote frames and it uses classical base format only, it sends a dominant bit during each of these three states.
- **TX_DLC**: In this state, length of the data is sent. Because the payload is passed as a pointer in the application layer, the length of the data is determined according to this field.
- **TX_DATA**: In this state, the payload of the frame is transmitted. In each cycle, the payload is shifted out and sent by a PCS data request.
- **TX_CRC**: The CRC field is shifted out and sent with data requests. Here, the CRC is not calculated during transmission. Instead it is calculated as soon as a transmission is requested from the application layer.
- **TX_CDEL, TX_ACK, TX_ADEL**: In all of these states, a recessive bit is sent.
- **TX_EOF, TX_EOF_TAIL**: Once the frame fields are completed, TX_FSM sends 7 recessive bits for 1 bit EOF and 6 trailing bits by invoking the PCS data request primitive. After this state is completed, the MAC data confirmation primitive is invoked and TX_FSM is reset to TX_IDLE since the frame transmission is finished.
- **TX_ERROR**: This state only becomes accessible once an error in RX_FSM is observed. Since TX_FSM cannot detect transmission errors by itself, it uses RX_FSM to monitor the bus during transmission. Once the state is set as TX_ERROR, it is reset to TX_IDLE and retransmission is started.

By implementing the two different automata given in Figure 3.3 & 3.4 and using the services of PCS & PMA layers, the MAC layer performs high level operations related to frame transmission and reception. Additionally, it implements an interface for lower layers to communicate with application layer software.

3.1.2 Physical Coding Sub-layer

The Physical Coding Sub-layer operates as a mid layer between MAC and PMA. It is responsible for maintaining bus synchronization and bit alignment. It uses the indications coming from PMA and provides service to the MAC layer. PCS periodically receives node clock indications with current bus levels from the PMA layer. Once it receives them, it directly performs edge detection by comparing the previous node clock bus level with the current one. If it detects a rising edge, it checks its local time quantum (TQ) counter. Since the length of each bit segment is determined by the user during the initialization phase of SDCC, PCS can easily determine the current segment when an edge is detected. If it determines that the current TQ is not in the SYNC segment, it calculates its relative position to the sampling point by simply subtracting the current position from sampling point. In this way, the phase error is calculated according to [2]. Following that, it accesses the variable that holds the MAC state to determine if the SOF bit is being transmitted on the bus for deciding if hard synchronization can be performed. According to the MAC state, it resets the TQ counter to perform hard synchronization or it adds the phase error to the TQ counter to lengthen PHASE1 or shorten PHASE2.

Following the phase correction, PCS generates a PCS data indication to invoke the MAC layer if the TQ counter reaches the sampling point. This indication is used to advance the reception and transmission state machines in the MAC layer. Finally, if the TQ counter reaches the bit boundary, it forwards the output buffer of the latest PCS data request to the PMA layer to trigger transmission. The timings of these operations are visualized in Figure 3.5.

3.1.3 Physical Medium Attachment Layer

The Physical Medium Attachment layer is the layer that is responsible for controlling the underlying hardware where SDCC runs. It uses the hardware dependent timer to generate a PMA node clock indication after each node clock for signalling PCS layer. To perform that, it directly reads the time value from the timer hardware and assigns it to a local time variable. Following that, it accesses the GPIO to sample the current bus

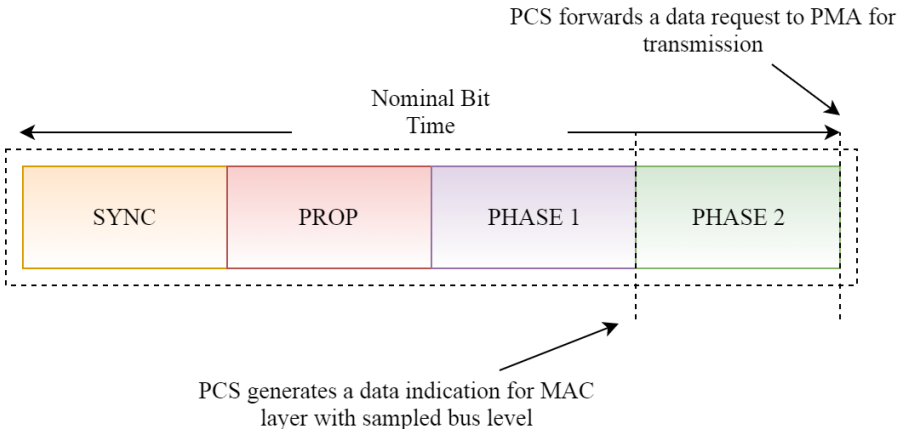

Figure 3.5: Timings of PCS Layer Operations

level. Finally, it passes this bus level to the PCS layer by generating a PCS node clock indication. PCS uses this information to track the sampling point and forward the bus level to the MAC layer if necessary. Here, the user can also register an application layer node clock indication to be called whenever a node clock is incremented. By using this functionality, the application layer can customize this indication to request a frame transmission or prepare the data for future transmissions. Additionally, it can monitor the bus in real-time.

After both PCS and application indication routines are generated and they are processed by the upper layers, PMA reads the current hardware time and compares it with the local time variable. If they are the same, PMA concludes that the next node clock is not missed and there is no cycle overflow. If the local time variable and current hardware time are different, PMA detects that a node clock is missed. Here, it has to be noted that there is no error correction mechanism for cycle overflow. PMA just notifies the user by a red LED in case of node clock miss. Specifically, a node clock miss indicates that the CAN bit rate was set too fast for the microcontroller hardware.

The last operation handled by the PMA is performing necessary GPIO operations according to PMA data requests sent by the PCS layer. Once PMA receives this data request, it sets the GPIO pin that corresponds to the transmit pin of CAN to the desired bus level. PMA performs this immediately since it relies on PCS to establish the necessary bit alignment. As it is stated before, PMA performs these low level operations on the actual hardware, therefore its behaviour is hardware dependent.

Hence, to clarify the interface between the underlying hardware and PMA layer, the hardware dependent part of the SDCC will be explained in following sections.

3.2 Hardware Dependency of Software-Defined CAN Controller

As it is given in Figure 3.1, SDCC consists of three layers with different tasks. However, it can be examined in two parts by considering hardware dependency. SDCC is implemented in the form of separate hardware dependent and hardware independent modules, so that the upper controller layers are abstracted from the hardware. As it is visualized in Figure 3.6, MAC and PCS layers are hardware independent and they use the services of the hardware dependent PMA layer.

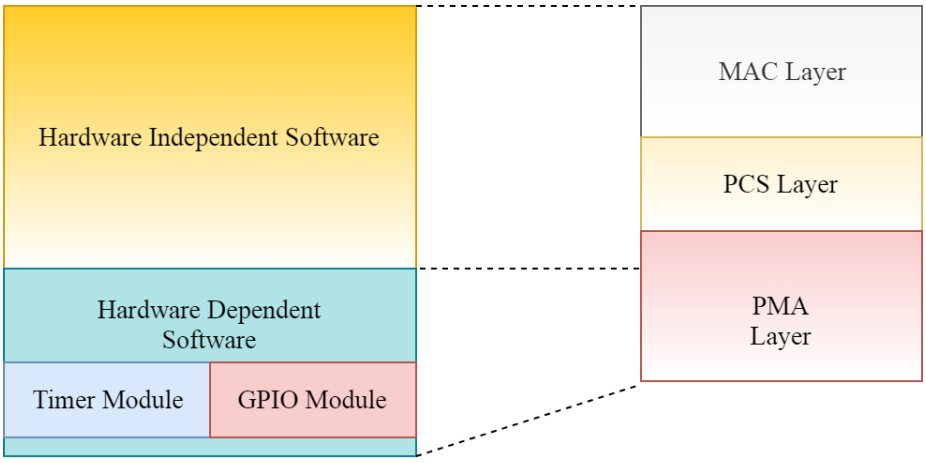

Figure 3.6: Software-Defined CAN Controller Hardware Dependency

In order to use SDCC on a custom board, the hardware dependent modules have to be ported to the target. To achieve this, the user should implement two different interfaces: timer module and general-purpose input/output (GPIO) module. Note that GPIO and timer interfaces are used by the PMA layer to provide services to PCS and MAC layers.

3.3 Porting Software-Defined CAN Controller

The main aim of this thesis is to employ the SDCC for the convenient implementation and experimental evaluation of software-based and hardware-based clock synchronization algorithms. That is, as the first important contribution of this thesis, we develop a suitable experimental setup and port the SDCC to the hardware components of this setup.

3.3.1 Experimental Setup

To deploy SDCC and to evaluate the clock synchronization algorithms which will be discussed in the next chapter, the experimental setup shown in Figure 3.7 is prepared. As CAN nodes, STM32F4 [38] and STM32F429I Discovery Evaluation Kits [39] are used.

In order to establish the connection between the nodes, the STM32F4 Discovery board is connected to a MPC2551 CAN transceiver [40], since this evaluation kit does not contain a built-in CAN transceiver. Differently, STM32F429I has a built-in CAN transceiver that is connected to the GPIO pins and this is why there is a single external transceiver in the setup. Please note that the same setup can be used without CAN transceivers by simply removing MPC2551, disabling built-in CAN transceiver on STM32F429I and connecting GPIO pins of the boards directly. However, since a real CAN network would require such transceivers, experimental setup is built with CAN transceivers to comply with CAN bus voltage characteristics given in [2].

Other than these two CAN nodes, there is an Arduino Mega [41] board connected to the CAN nodes via a single wire. Although it does not play an active role in CAN communication, it will be required for synchronously collecting data from the clock synchronization algorithms discussed in following chapters. Additionally, a logic analyzer is connected to the CANH and CANL channels to probe the signals and parse CAN frames if it is necessary.

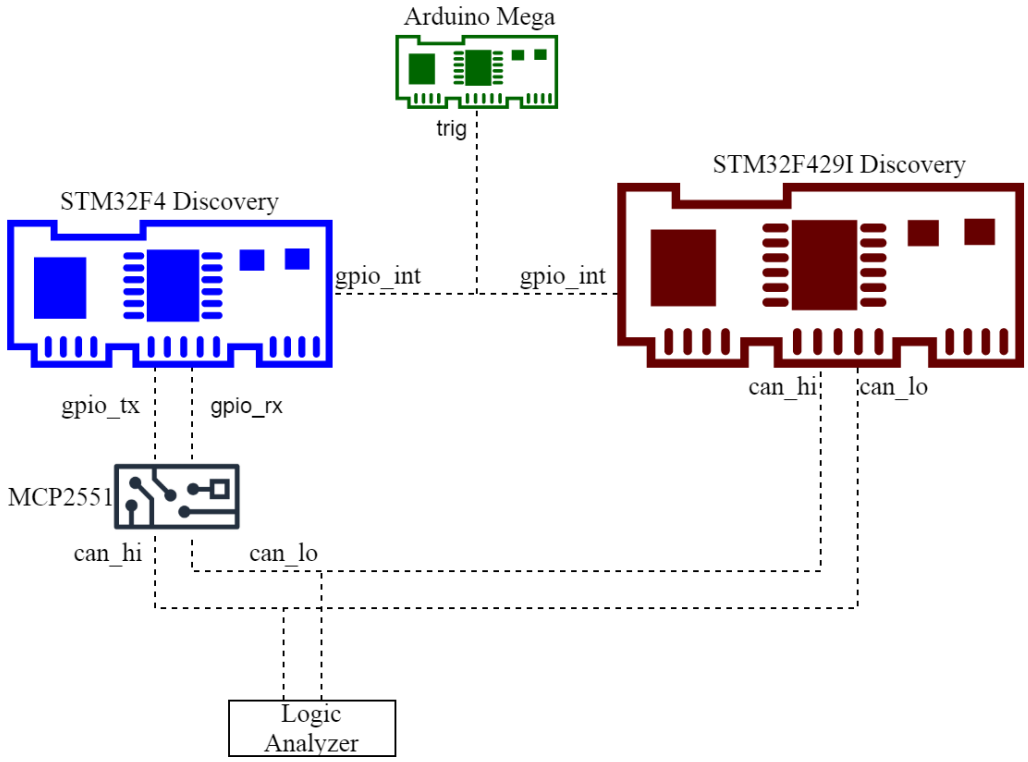

Figure 3.7: Experimental Setup Diagram

3.3.2 Porting SDCC to Experimental Setup

As shown in Figure 3.1, SDCC contains two hardware dependent modules for a hardware timer and GPIO pins. Therefore, in order to establish a communication between two nodes with SDCC, these two modules should be ported to the target. Since experimental setup consists of STM32F4 Discovery and STM32F429I Discovery boards, the GPIO and timer functions need to be adapted accordingly. In this section porting steps of these functions will be explained in detail.

The hardware dependent timer module of SDCC consists of two functions: $setup\_ts$ and $read\_ts$. Function $setup\_ts$ is responsible for initializing the hardware timer that will be used for clocking the CAN controller. It takes a prescaler value from the application layer, and initializes the timer registers accordingly. Since STM32F429I and STM32F4 boards are very similar, this function is common for both boards, as it can be seen from the Source Code 1. The function first enables the TIM5 clock from Reset and Clock Control Register. Then, it sets the prescaler value and enables

Source Code 1 Hardware Dependent Timer Functions

```
1  #if defined(GCC_ARM_STM32429I) || defined(GCC_ARM_STM32F4)
2  static void setup_ts(uint16_t prescaler)
3  {
4      /* TMR5 */
5      /* Enable timer clock */
6      RCC_APB1ENR |= 0x8;
7
8      /* Set prescaler */
9      TIM5_PRESCALER |= prescaler;
10
11     /* Clear counter */
12     TIM5_CNT = 0;
13
14     /* Enable counter */
15     TIM5_CR1 |= 0x1;
16
17     /* Update generation to flush registers */
18     TIM5_EGR |= 0x1;
19
20     /* Init virtual clock (TMR2) */
21     init_virtual_clock();
22 }
23 #define read_ts() TIM5_CNT
```

the counter. The function called $read\_ts$ is implemented as a one line definition that accesses TIM5 counter register directly. Note that this is the very function that PMA uses to infer the node clock.

In addition to the initialization of node clock that drives the CAN controller, there is another initialization function $init\_virtual\_clock$ called in line 21 of Source Code 1. This initialization will be required for the time synchronization methods that will be implemented on the SDCC. Differing from the timer that generates node clock, the prescaler of this timer is set as zero for providing better measurement accuracy for the next chapters. Note that function details are omitted, since it is very similar to the function $setup\_ts$. The only difference is that instead of TIM5, TIM2 timer module is used.

The second hardware dependent module GPIO consists of five functions given in Source Code 2 and 3. The first function $init\_gpio$ initializes two GPIO pins that will be used for transmitting (TX) and receiving (RX) CAN frames. It enables related GPIO port clock, then configures the RX pin as input and TX pin as output. It also configures push-pull state and slew rate of both pins. Finally, after initial-

Source Code 2 Hardware Dependent GPIO Functions Part 1

```
#ifdef GCC_ARM_STM32429I
static void init_gpio(void)
{
    /* Enable Clock for GPIOA */
    RCC_AHB1ENR |= 0x1;

    /* Set PA12 output and P11 input*/
    GPIOA_MODER |= 0x01000000;

    /* Select output push-pull for PA11/PA12 */
    GPIOA_OTYPER &= ~(0x00001800);

    /* Select output speed to low for PA11/PA12 */
    GPIOA_OSPEEDR &= ~(0x03C00000);

    /* Select no pull-up/pull-down for PA11/PA12 */
    GPIOA_PUPDR &= ~(0x03C00000);

    /* Select mode as 0 for PA11/PA12 */
    GPIOA_AFRH |= (0x00000000);

    /* Set tx as recessive*/
    gpio_tx_rec();
}

#define gpio_tx_rec() (GPIOA_BSRR |= PA12_SETMASK)
#define gpio_tx_dom() (GPIOA_BSRR |= PA12_CLRMASK)
#define gpio_tx_pin() ((GPIOA_ODR & PA12_MASK) >> 12)
#define gpio_rx_pin() ((GPIOA_IDR & PA11_MASK) >> 11)

#endif /* GCC_ARM_STM32429I */
```

ization is completed, the transmit pin is set as recessive to set the bus as idle. The functions $gpio\_tx\_rec$ and $gpio\_tx\_dom$ are used for setting the TX pin value, while $gpio\_tx\_pin$ and $gpio\_rx\_pin$ are used for retrieving the current levels of the RX and TX pins. These functions are implemented as definitions since they only read/write one register in the related GPIO port. Note that on STM32F429I, PA11 and PA12 pins are used for accessing the bus whereas PB5 and PB6 are used on STM32F4.

3.3.3 Running SDCC on the Experimental Setup

After the experimental setup is built, SDCC is ported to STM32F4 and STM32429I by following the steps given in Section 3.3.2. Then, in order to establish the communication between the two boards, several example projects with SDCC are created. After they are compiled and uploaded to the boards, it is observed that cycle over-

Source Code 3 Hardware Dependent GPIO Functions Part 2

```
#ifdef GCC_ARM_STM32F4
static void init_gpio(void)
{
    /* Enable Clock for GPIOA */
    RCC_AHB1ENR |= 0x2;

    /* Set PB6 output and PB5 input*/
    GPIOB_MODER |= 0x00001000;

    /* Select output push-pull for PB5/PB6 */
    GPIOB_OTYPER &= ~(0x00000060);

    /* Select output speed to low for PB5/PB6 */
    GPIOB_OSPEEDR &= ~(0x00003C00);

    /* Select no pull-up/pull-down for PB5/PB6 */
    GPIOB_PUPDR &= ~(0x00003C00);

    /* Select mode as 0 for PB5 and PB6 */
    GPIOA_AFRH |= (0x00000000);

    /* Set tx as recessive*/
    gpio_tx_rec();
}

#define gpio_tx_rec() (GPIOB_BSRR |= PB6_SETMASK)
#define gpio_tx_dom() (GPIOB_BSRR |= PB6_CLRMASK)
#define gpio_tx_pin() ((GPIOB_ODR & PB6_MASK) >> 6)
#define gpio_rx_pin() ((GPIOB_IDR & PB5_MASK) >> 5)

#endif /* GCC_ARM_STM32F4 */
```

flows are present. The reason for this is that the SDCC software misses CAN node clocks since the CAN clock is configured to be too fast. This is why the function $setup\_ts$ has a prescaler input that can be used to slow CAN clock down.

To achieve the maximum bit rate without any cycle overflows, all printout/debug functions of SDCC are disabled, since most of the controller functions are time-critical. Following that, in order to eliminate cycle overflows, the prescaler value is calculated using the relation given in Equation 3.1. By using this equation, it is observed that if the CAN bit rate is selected above 35kHz, node clocks start to be missed and the communication ends up with cycle overflows. Therefore, the maximum CAN bit rate that can be achieved with the experimental setup is calculated as 35kHz. Finally, since our system clock is 180MHz, the prescaler is selected as 643 to set the CAN bit rate

to maximum.

$$can\_clock\_prescaler = system\_clock/(can\_bit\_rate \cdot can\_bit\_quanta) \quad (3.1)$$

Following the solution of the cycle overflow problem, it is observed that CAN frames are transmitted/received successfully by the nodes. Observing the physical CAN frames given in Figure 3.8 enabled the implementation of clock synchronization algorithms that will be discussed in the following chapters. Since software solutions for clock synchronization over CAN mostly work on the application layer and they are relatively easier compared to hardware solutions, they will be discussed first and will be followed by hardware approaches.

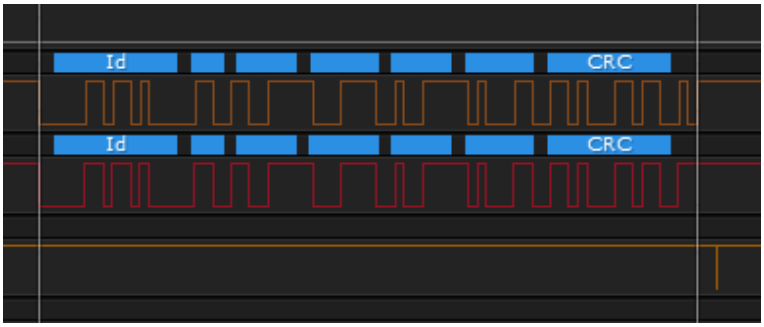

Figure 3.8: Observed SDCC Waveform

CHAPTER 4

SOFTWARE APPROACH TO CLOCK SYNCHRONIZATION ON CAN

As described in Section 2.6, there are software and hardware approaches for providing clock synchronization on CAN. Software-based approaches are realized as applications that operate on top of the CAN controller, while hardware-based approaches tend to modify or extend the existing CAN controller to add clock synchronization support. Software-based approaches seem to be more popular compared to hardware-based approaches, since hardware solutions require tedious work even for a small extension or modification to existing CAN controller.

This chapter focuses on software-based clock synchronization using the SDCC. To this end, Section 4.1 and 4.2 give details about the implementation of two specific methods: Gergeleit's method in [18] and the improved software-based clock synchronization (ISCS) method in [19]. These methods are then evaluated using the experimental setup and the results are presented in Section 4.3.

4.1 Gergeleit's Method

4.1.1 Description

The initial approach to clock synchronization on CAN was sending the timestamp taken by one node to the others on the network. However, latency of the transmission path is not deterministic due the possibility of blocking, arbitration loss and difference in transmission speed [18]. Additionally, software running on the receiving nodes is also critical, since it may change the instant, where each node processes the received timestamp. Hence, it can be stated that it is not possible to provide precise clock

synchronization by simply broadcasting time information.

In order to eliminate this non-deterministic latency, [18] proposes a method that exploits the synchronous reception property of CAN frames. This method uses the concept of a Time Master (TM) that periodically sends reference frames and Time Slaves (TS) that correct their clock accordingly. Although a reference frame is nothing but a timestamp taken by TM, sending it right away is not viable as it is discussed in previous paragraph. Hence, [18] suggests that after a reference frame is received, each node on the network takes and stores a local timestamp. By this way, it is ensured that all nodes record their local time at the same time instant. Subsequently, when the TM decides to send a reference frame, it sends the timestamp that is taken after the previous reference frame. In order to make it clear, this process is visualized in Figure 4.1.

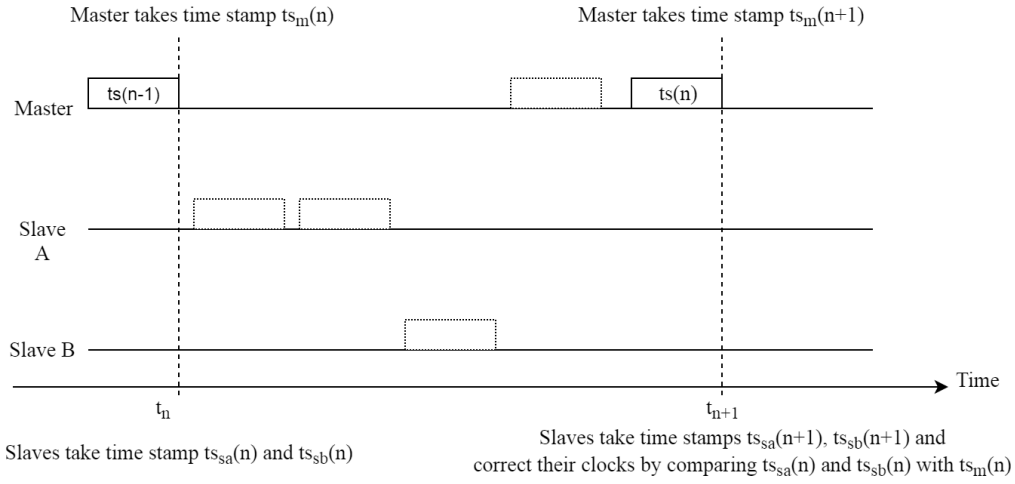

Figure 4.1: Gergeleit's Algorithm for Clock Synchronization on CAN

After the TM sends the reference frame $ts(n-1)$ and it is received by every node on the network at time t_n, the TS nodes take the timestamps $ts_{sa}(n), ts_{sb}(n)$. At the same instant, TM takes the timestamp $ts_m(n)$. That is, this is the time instant, where [18] exploits the simultaneous reception property of CAN. In this way, it is ensured that each node on the network takes a local timestamp at the same time. Subsequently, when the reference frame period is reached, the TM sends the latest timestamp that it stored, $ts_m(n)$. After this frame is received at t_{n+1}, each node takes a timestamp again. Consequently, differing from TM node, the TS nodes correct their local clocks by comparing their latest timestamps $ts_{sa}(n)$ and $ts_{sb}(n)$ with $ts_m(n)$. For example,

TS A can correct its local clock by adding $ts_m(n) - ts_{sa}(n)$ to its current value.

4.1.2 Implementation on SDCC

As it is stated before, the method in [18] is a software-based clock synchronization method. It is designed for the software layer that is operating the CAN controller for sending and receiving CAN frames. Hence, it does not require any changes in the standard CAN controller. Knowing that time reference frames are just CAN frames with unique identifiers, this method can directly be integrated into the application layer software. Therefore, it is directly implemented on the application layer of the SDCC, without changing any controller layers.

From the SDCC point of view, Gergeleit's method may be integrated into the application layer by using data indications and application node clock indications provided by the MAC and PMA layers respectively. As they are explained in Section 3.1.1 and 3.1.3, MAC data indications notify the application software when a frame is received, while PMA application node clock indication is triggered on every nominal period of CAN clock. Please note that the functions which are called after these indications should be implemented by the user and registered to the SDCC during initialization of the driver. SDCC stores them as function pointers and calls the appropriate routine when it is necessary.

In addition to the data and node clock indications, a data confirmation function can be written and registered to the SDCC during initialization in order to observe transmission errors on the Time Master side. As stated in Section 3.1, data confirmation is triggered when a frame transmission is completed. In this way, the transmission status of reference messages can be monitored. The implementation of Gergeleit's method on SDCC is outlined in Source Code 4 to 7.

Function $time\_master\_data\_ind$ given in Source Code 4 is implemented for the tasks the TM has to perform after a reference frame is received. According to Gergeleit's method, the TM shall take a timestamp right after a reference frame is received. This operation is implemented in a MAC data indication since they are triggered when a CAN frame is received successfully. However, it has to be deter-

Source Code 4 Time Master Data Indication For Gergeleit's Method

```
1  void time_master_data_ind(
2      struct CAN_XR_LLC *llc, unsigned long ts, uint32_t identifier,
3      enum CAN_XR_Format format, int dlc, uint8_t *data)
4  {
5    /* Gergeleit Method */
6    if (gergeleit_enabled && (identifier == TIMESYNC_CAN_ID) &&
7        (dlc == sizeof(tx_gerg_ref_msg))
8    {
9      prev_ts = read_virtual_clock();
10   }
11
12   /* Processing can be done here for non-reference frames */
13   /* ... */
14 }
```

mined if the received frame is a reference frame, because data indications are invoked after each frame. This filtering can be done by using the $identifier$ and dlc fields provided by the data indication. As can be seen in lines 6-7 of Source Code 4, the identifier of the received frame is compared to the predefined reference frame identifier `TIMESYNC_CAN_ID`. Additionally, the length of the frame is compared to the size of the reference message, which is 4 bytes since we are using a 32-bit timer. After it is ensured that a reference frame is received successfully, a timestamp is taken and stored in a global variable $prev\_ts$, to be sent in the next reference frame. After this block is executed, additional application layer processing on the received frame can be done, if necessary.

Source Code 5 Time Slave Data Indication For Gergeleit's Method

```
1  void time_slave_data_ind(
2      struct CAN_XR_LLC *llc, unsigned long ts, uint32_t identifier,
3      enum CAN_XR_Format format, int dlc, uint8_t *data)
4  {
5    /* Gergeleit Method */
6    if (gergeleit_enabled && (identifier == TIMESYNC_CAN_ID) &&
7        (dlc == sizeof(tx_gerg_ref_msg))
8    {
9      cur_ts = read_virtual_clock();
10     set_virtual_clock(read_virtual_clock() + (*(uint32_t*)data - prev_ts));
11     prev_ts = cur_ts;
12   }
13
14   /* Processing can be done here for non-reference frames */
15   /* ... */
16 }
```

The function called *time_slave_data_ind* illustrated in Source Code 5 defines the behaviour of time slaves when a frame is received. As for the TM, this function is called after the reception of each CAN frame. Therefore, in lines 6-7 of Source Code 5, reference frame is filtered by checking size and identifier of the frame. After it is ensured that a reference frame is received, a timestamp is taken immediately. Following that, clock correction is performed in line 10, by simply adding the difference between the previous local timestamp and the received one. After correction is completed, the most recent local timestamp is transferred to *prev_ts* for the next synchronization. After this operation is completed, the slave can perform additional processing depending on its application.

Source Code 6 Time Master Node Clock Indication For Gergeleit's Method

```
1  void time_master_nodeclock_ind(struct CAN_XR_PCS *pcs, int bus_level)
2  {
3    if (gergeleit_enabled && ref_msg_ready &&
4        !pcs->mac.state_data_req_pending)
5    {
6      /* Send the reference frame */
7      CAN_XR_MAC_Data_Req(&mac, TIMESYNC_CAN_ID, CAN_XR_FORMAT_CBFF,
8                          sizeof(tx_gerg_ref_msg), (uint8_t *)&prev_ts);
9      ref_msg_ready = false;
10   }
11
12   /* Processing can be done here for non-reference frames */
13   /* ... */
14 }
```

As given in Source Code 6, *time_master_nodeclock_ind* is used for triggering transmission of any kind of CAN frame. In Gergeleit's method, this is needed by the TM to send the reference frame by issuing a data request provided by the MAC layer of SDCC. Since a node clock indication is issued on every nominal period of the CAN clock and reference frames are periodic messages, the TM has to wait for this period. As can be seen in lines 3 and 4 of Source Code 6, it is checked with a boolean flag that is set to *true* with the help of the hardware timer when a frame is ready to be sent. Additionally, the current state of MAC is checked to see if there is a pending data request. If the period of the reference frame is reached and there is no frame pending, the TM sends the latest timestamp by issuing a 32-bit data transmission request.

In order to observe reference frame transmission errors during communication, the

Source Code 7 Time Master Data Confirmation For Gergeleit's Method

```
void time_master_data_conf(
    struct CAN_XR_LLC *llc, unsigned long ts, uint32_t identifier,
    enum CAN_XR_MAC_Tx_Status transmission_status)
{
  if (gergeleit_enabled && (identifier == TIMESYNC_CAN_ID)
  && (transmission_status != CAN_XR_MAC_TX_STATUS_SUCCESS))
    error_cnt++;
}
```

function called $time\_master\_data\_conf$ given in Source Code 7 is implemented. For filtering errors related to reference frames, identifier and transmission status are checked. In addition to that, a static integer called $error\_cnt$ is incremented with every error to monitor while experimental results are being examined. Note that, if a transmission is not successful, SDCC issues a re-transmission automatically.

4.2 ISCS

4.2.1 Description

In order to improve the performance of existing software-based clock synchronization algorithms on CAN, the Improved Software-Based Clock Synchronization (ISCS) is introduced in [19]. [19] states that after timestamps are taken, the slave node clocks continue to drift and this degrades the performance of the clock synchronization. To solve this, timestamps should be taken closer to the next reference frame. Although it can be achieved by increasing the frequency of the reference frames, this will create an additional load on the bus. Hence, in order to solve this problem without increasing the frequency of the reference messages, ISCS suggests making use of the reception of each frame on the bus instead of only reference frames to take timestamps.

Differing from [18], after each frame reception, nodes take a timestamp and store the previous one. Additionally, the reference frame has another field that is called "validity flag" which nodes store for verifying that a reference frame is valid. This flag is crucial when the TM issues the transmission of a reference frame but the bus is busy, because an another node is transmitting a frame. Although the reference

frame will be transmitted after the bus becomes idle, other nodes will already have taken another timestamp. In this case, the reference frame should be interpreted as invalid since a slave node correcting its local clock according to this frame will cause an increase in the clock difference between nodes. Therefore, each node toggles its local validity flag when a timestamp is taken after a frame is received. When the TM issues the transmission of a reference frame but the bus is busy, nodes on the bus will toggle their validity flag after the frame that currently being transmitted. Hence, the reference frame validity flag will become different from local validity flag of the nodes. However, this frame will not be discarded, instead when TS nodes observe a validity flag mismatch, they correct their local clocks according to the difference between received timestamp and the previous timestamp that they stored. If the flags match, they use the current timestamp instead of the previous one. Flag match and flag mismatch cases of ISCS algorithm are visualized in Figure 4.2 and 4.3.

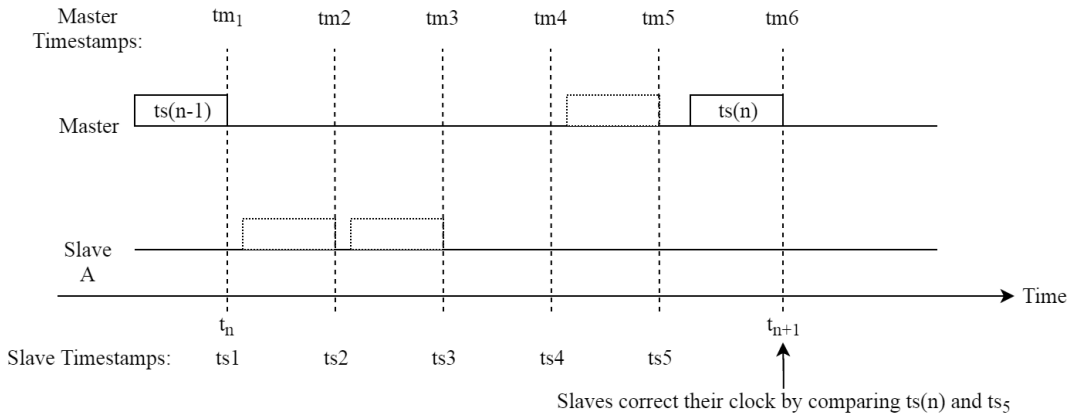

Figure 4.2: ISCS Algorithm Operation - Validity Flag Match

4.2.2 Implementation on SDCC

Like Gergeleit's method, ISCS is purely based on software. Hence, without any modification of the SDCC, it will be implemented in the application layer by using data indications and application node clock indications provided by SDCC. Data indications will be used for receiving the reference frame, taking a timestamp and clock correction while application node clock indications will be used for issuing the transmission of a reference frame when its period is reached. For debugging and monitor-

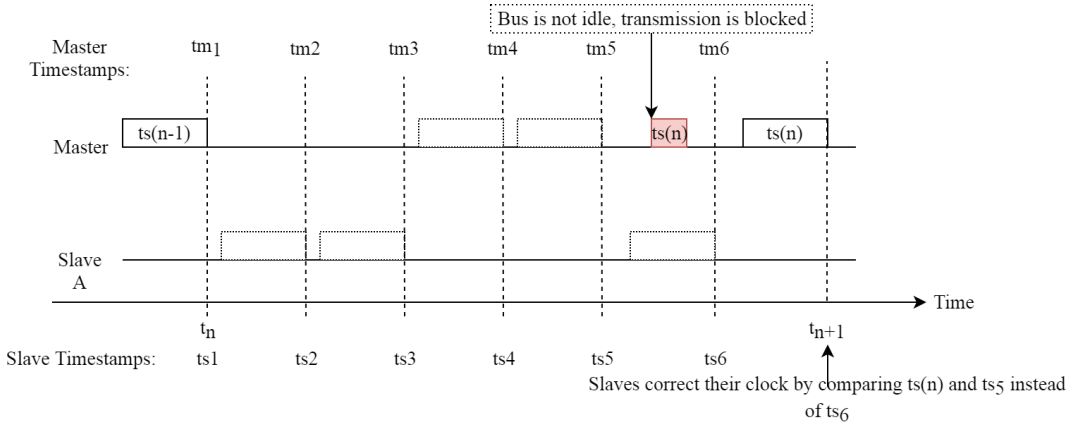

Figure 4.3: ISCS Algorithm Operation - Validity Flag Mismatch

ing purposes, the data confirmation mechanism of SDCC will also be used.

Differing from Gergeleit's algorithm, the ISCS reference frame has an additional 1 bit field for the validity flag. Appending this validity flag after the 32-bit timestamp is not a practical solution, because payload size of a CAN frame should be multiple of a byte. Hence, transmitting 33-bit value will require 5 bytes. This unwanted situation can be solved by sacrificing the last bit of the 32-bit timestamp and placing the validity flag in that last bit. With this modification, the reference frame will fit into 4 bytes of payload without any additional overhead. The data structure used for this purpose is given in Source Code 8.

Source Code 8 Data Structure for ISCS Reference Frame

```
/* Definition of ISCS reference Message */
typedef struct _tx_ref_msg
{
  union {
    struct {
      uint32_t vFlag : 1;
      uint32_t TS    : 31;
    };
    uint32_t msg;
  };
}__attribute__((packed))tx_ref_msg;
```

Illustrated in Source Code 9 to 12, ISCS algorithm is implemented on SDCC with 4 different routines that are responsible for different tasks such as processing/generating reference frames, managing timestamps and validity flags and performing clock

correction.

Source Code 9 Data Indication For ISCS Method

```
void all_nodes_data_ind(
    struct CAN_XR_LLC *llc, unsigned long ts, uint32_t identifier,
    enum CAN_XR_Format format, int dlc, uint8_t *data)
{
  /* ISCS Method */
  if (iscs_enabled)
  {
    clockTS(identifier, dlc, data);
  }

  /* Processing can be done here for non-reference frames */
  /* ... */
}
```

The function $all\_nodes\_data\_ind$ given in Source Code 9 is used for handling frame receptions. It checks if ISCS is enabled and calls the lower level function $clockTS$ given in Source Code 10. Note that this function is common for all nodes and it is registered to the SDCC to be called whenever slave and master nodes receive a frame.

Source Code 10 ISCS clockTS Function

```
void clockTS(uint32_t msg_id, int dlc, uint8_t *data)
{
  /* Check if this is a valid time reference*/
  if(msg_id == TIMESYNC_CAN_ID && dlc == (sizeof(tx_ref_msg)))
  {
    syncClock((tx_ref_msg *)data); // call the synchronization serivce
  }

  vFlag = !vFlag; // Toggle vFlag
  prevTS = curTS;

  /* take new timestamp */
  curTS = read_virtual_clock();
}
```

The function $clockTS$ given in Source Code 10, is responsible for calling the clock correction routine, storing local timestamps and managing the validity flag. First, it determines if the received frame is a reference frame by checking the frame identifier. Then, it checks if the data size matches the size of a reference frame to perform a sanity check. If it ensures that a reference frame is received, it calls the function $syncClock$ given in Source Code 11 to perform a clock correction. After that, it toggles the validity flag which will be used to check the validity of the next reference

frame. Finally, it stores the latest timestamp and takes another one. Since this function is called after every frame reception, the timestamp transmitted by the TM will be closer to the reference frame, and this is the main advantage of ISCS method.

Source Code 11 ISCS syncClock Function

```
1  void syncClock(tx_ref_msg* Rx_M)
2  {
3    int32_t clockDiff;
4
5    if ((Rx_M->vFlag) == vFlag)
6    {
7      clockDiff = Rx_M->msg - curTS;
8    }
9    else
10   {
11     clockDiff = Rx_M->msg - prevTS;
12   }
13   /* Reset local timestamp and validity flag */
14   prevTS = 0;
15   curTS  = 0;
16   vFlag  = 0;
17
18   /* Correct local clock */
19   set_virtual_clock(read_virtual_clock() + clockDiff);
20 }
```

The function $syncClock$ is called from the function $clockTS$ whenever a reference frame is received. The main purpose of this function is to process the reference frame and correct the local clock accordingly. As is given in Section 4.2.1, clock correction starts with the check of the validity flag of the received frame. Since the validity bit is placed in the last bit of the reference frame, the local validity flag and the last bit of the received frame are compared. If the flags match, the clock difference is determined by the difference between received timestamp and current local timestamp. If flag mismatch is observed, then the difference between the received timestamp and the previous local timestamp is used instead. After calculation of the clock difference given in lines 5-12 of Source Code 11, timestamps and validity flag are reset. Finally, clock correction is performed according to the calculated clock difference.

Used for generating and sending a reference frame, $time\_master\_nodeclock\_ind$ given in Source Code 12 is executed by the TM after each nominal period of the CAN clock. First, it checks if ISCS is enabled and determines if the reference message period is reached by reading a boolean flag set by the hardware timer. If a reference

Source Code 12 Time Master Node Clock Indication & refTx Function For ISCS Method

```
1  void time_master_nodeclock_ind(struct CAN_XR_PCS *pcs, int bus_level)
2  {
3    if (iscs_enabled && ref_msg_ready)
4    {
5      refTx(&mac);
6
7      ref_msg_ready = false;
8    }
9
10   /* Processing can be done here for non-reference frames */
11   /* ... */
12 }
13
14 void refTx(struct CAN_XR_MAC *mac)
15 {
16   /* Create the reference message */
17   tx_ref_msg Tx_Ref_M;
18
19   /* Discard last bit of the timestamp and place validity flag */
20   Tx_Ref_M.msg = curTS;
21   Tx_Ref_M.vFlag = vFlag;
22
23   /* Send the reference frame */
24   CAN_XR_MAC_Data_Req(mac,
25                       TIMESYNC_CAN_ID, CAN_XR_FORMAT_CBFF,
26                       sizeof(tx_ref_msg), (uint8_t *)&Tx_Ref_M);
27 }
```

frame is ready to be sent, it calls the lower level function $refTx$. Given in line 14 of Source Code 12, function $refTx$ generates the reference frame by discarding the last bit of the latest timestamp and placing the validity flag there. After the reference message is formed, the function transmits the reference frame by issuing a 32-bit data transmission request.

4.3 Results

To evaluate the performance of the software-based clock synchronization algorithms implemented on SDCC, the experimental setup described in Section 3.3.1 is used. As can be seen from Figure 3.7, one additional microcontroller is connected to the CAN nodes for performance measurement. Since the performance of a clock synchronization algorithm can only be measured by obtaining the actual difference between local clocks of the nodes, it should be ensured that each node collects data at the same time

instants. Hence, in order to perform such data collection, this additional microcontroller is used for triggering external interrupts on the CAN nodes. This is achieved by programming this microcontroller to provide periodic positive/negative edges to the CAN nodes via a wire. Additionally, a simple interrupt subroutine callback given in Source Code 13 is written to collect data when a rising edge is observed on the pins H2 and C1 of the STM32F4 and STM32F429I Evaluation Kits, respectively.

Source Code 13 Data Collection Interrupt Subroutine Callback

```
void gpio_exti_callback((void)
  if (global_sync_lock == 1)
  {
    /* Reset virtual clock */
    set_virtual_clock(0);
    /* Reset Gergeleit globals */
    prev_ts = 0;
    cur_ts = 0;
    /* Reset ISCS globals */
    curTS = 0;
    prevTS = 0;
  }

  else
  {
    if (data_coll_index != DATA_COLL_ARRAY_SIZE)
      data_coll_array[data_coll_index++] = read_virtual_clock();
    else
      while(1);
  }
```

The variable $global\_sync\_lock$ is reset by the nodes when the first reference frame is received. Hence, until a reference frame is received, the virtual clock of each node is reset and the global variables used by synchronization algorithms are set to 0. This operation is performed to provide an initial synchronization between the nodes for evaluating the performance of the synchronization algorithms properly. After the first reference message is received ($global\_sync\_lock$ is reset), each node starts recording its virtual time to $data\_coll\_array$ after each external interrupt. When this array becomes full, CAN nodes are stopped for evaluation of the collected data. As is summarized in Figure 4.4, by performing the described data collection method, it is ensured that each node collects data at the same time instants, denoted as td_1, td_2 and td_3.

In order to evaluate the performance of the clock synchronization algorithms under

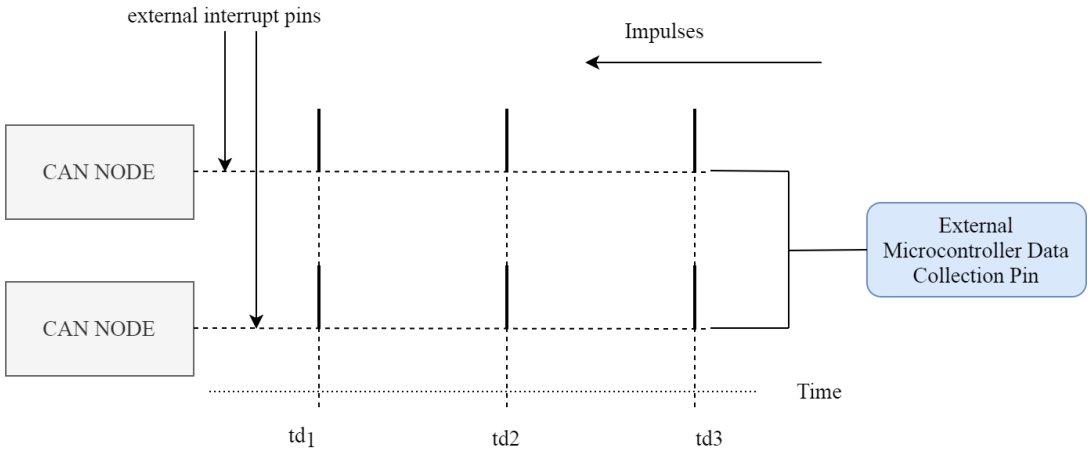

Figure 4.4: Data Collection for Performance Measurement

different bus loads, additional bus traffic should be generated. Therefore, the CAN nodes in the experimental setup shall be able to transmit dummy frames addition to time reference frames. To add this capability, the code block given in Source Code 14 is appended to the application node clock indication of each CAN node. By executing this block, additional frames are generated with period of $dummy\_frame\_period$. Hence, different bus loads can simply be achieved by changing this variable. Note that this variable is calculated according to the relation given in Equation 4.1.

Source Code 14 Creating Additional Traffic on the Bus

```
if (elapsed_time > dummy_frame_period)
{
  CAN_XR_MAC_Data_Req(&mac,
    DUMMY_FRAME_ID, CAN_XR_FORMAT_CBFF, 4, data);
}
```

$$dummy\_frame\_period = (dummy\_frame\_transmission\_time \cdot 2)/load \quad (4.1)$$

After the described interrupt subroutine and traffic generation block are integrated into the CAN nodes on the experimental setup, three different test cases are prepared for performance evaluation of the software-based clock synchronization algorithms on SDCC. In the first case, all clock synchronization algorithms are disabled and the clock drift between the CAN nodes is measured. In the second and third cases,

the reference frame period is selected as 200 milliseconds and the performance of the algorithms in Section 4.1 (Gergeleit) and 4.2 (ISCS) are measured under three different bus loads of 100%, 80% and 40%. In all three cases, the virtual clock of each node is sampled with a period of 2.5 milliseconds for 65 seconds. Afterwards, in order to calculate average clock drift between nodes, the differences of collected samples are obtained first. Then, the sum of the absolute clock differences is divided by the number of samples. Finally, this value is converted into microseconds by simply dividing it by clock frequency, which is 180MHz in our case. For the maximum clock drift, the maximum absolute clock difference is converted into microseconds.

After performing the first test case over 10 times, the results given in Table 4.1 are obtained. As is visualized in Figure 4.5, when no clock synchronization method is applied, the virtual clocks of CAN nodes linearly drifted apart 5269.36 microseconds after 65 seconds. By using this value, clock drift between CAN nodes is calculated as 81 ppm, which is realistic according to the discussion in Section 2.4 since we are using different boards with different oscillators for the nodes.

Table 4.1: Clock Drift Measurement Without Clock Synchronization Algorithm

Parameters	**No Clock Synchronization**
Average Clock Drift	2635.38 us
Maximum Clock Drift	5269.36 us

After performing the second and third test case over 15 times, the average and maximum clock drift values for different bus loads in Table 4.2 and Figure 4.6 are obtained. It is observed that ISCS decreased the maximum clock drift by 54% compared to Gergeleit's method. For lower bus loads of 80% and 40%, the improvement in the maximum clock drift is gradually reduced to 46%, since the performance of ISCS directly depends on the bus load. As it can be seen from Figure 4.7 on which the clock drift performances of both algorithms are compared, Gergeleit's method frequently performed overcorrections. On the other hand, ISCS manages to keep the clock drift in a smaller interval. As a result, the maximum clock drift is significantly decreased when ISCS is applied. In addition, as it can be inferred from Figure 4.8, the variance

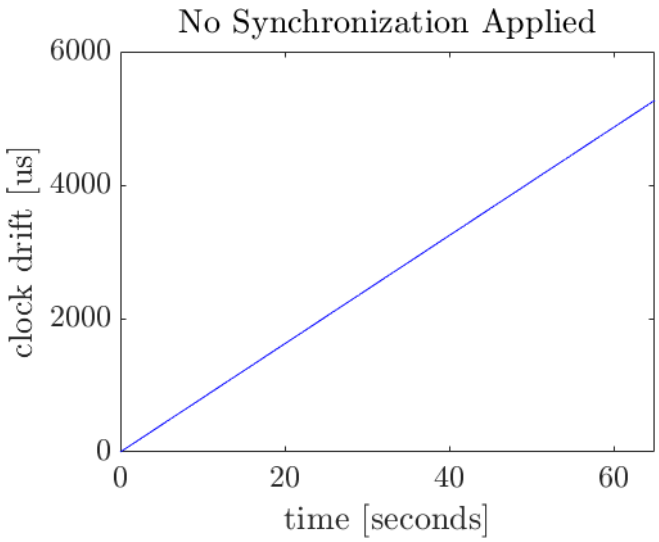

Figure 4.5: Evaluation of the Clock Drift between the CAN nodes

in the clock drift is also decreased drastically using the ISCS algorithm.

As in the case with the maximum clock drift, ISCS outperformed Gergeleit's method by 34% under full bus load when the average clock drift is considered. This improvement is decreased to 21% for lower bus loads, because of ISCS's performance dependency on the bus load. This dependency can be explained with ISCS's timestamp mechanism. Differing from Gergeleit's method, ISCS makes use of each frame on the bus to provide timestamps closer to the reference frames. Therefore, a decrease in the overall CAN frame frequency eventually increases the time duration between timestamps and reference frames and this degrades the performance of ISCS.

Table 4.2: Performance of ISCS and Gergeleit's Method on SDCC Under Different Bus Loads (with 81 ppm Clock Drift)

Parameters		**Gergeleit's Method [18]**	**ISCS [19]**
Average Clock Drift	100% Load	11.37 us	7.51 us
	80% Load	11.60 us	7.78 us
	40% Load	11.62 us	9.22 us
Maximum Clock Drift	100% Load	37.26 us	17.49 us
	80% Load	35.65 us	19.51 us
	40% Load	36.9 us	19.65 us

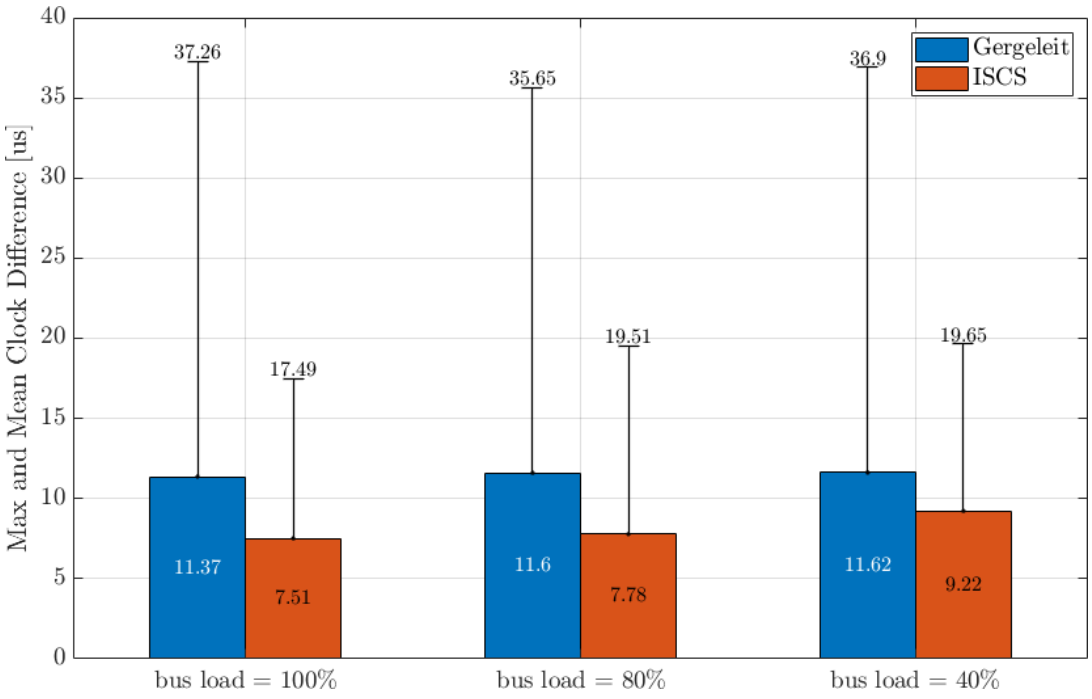

Figure 4.6: Performance of ISCS and Gergeleit's Method on SDCC Under Different Bus Loads (with 81 ppm Clock Drift)

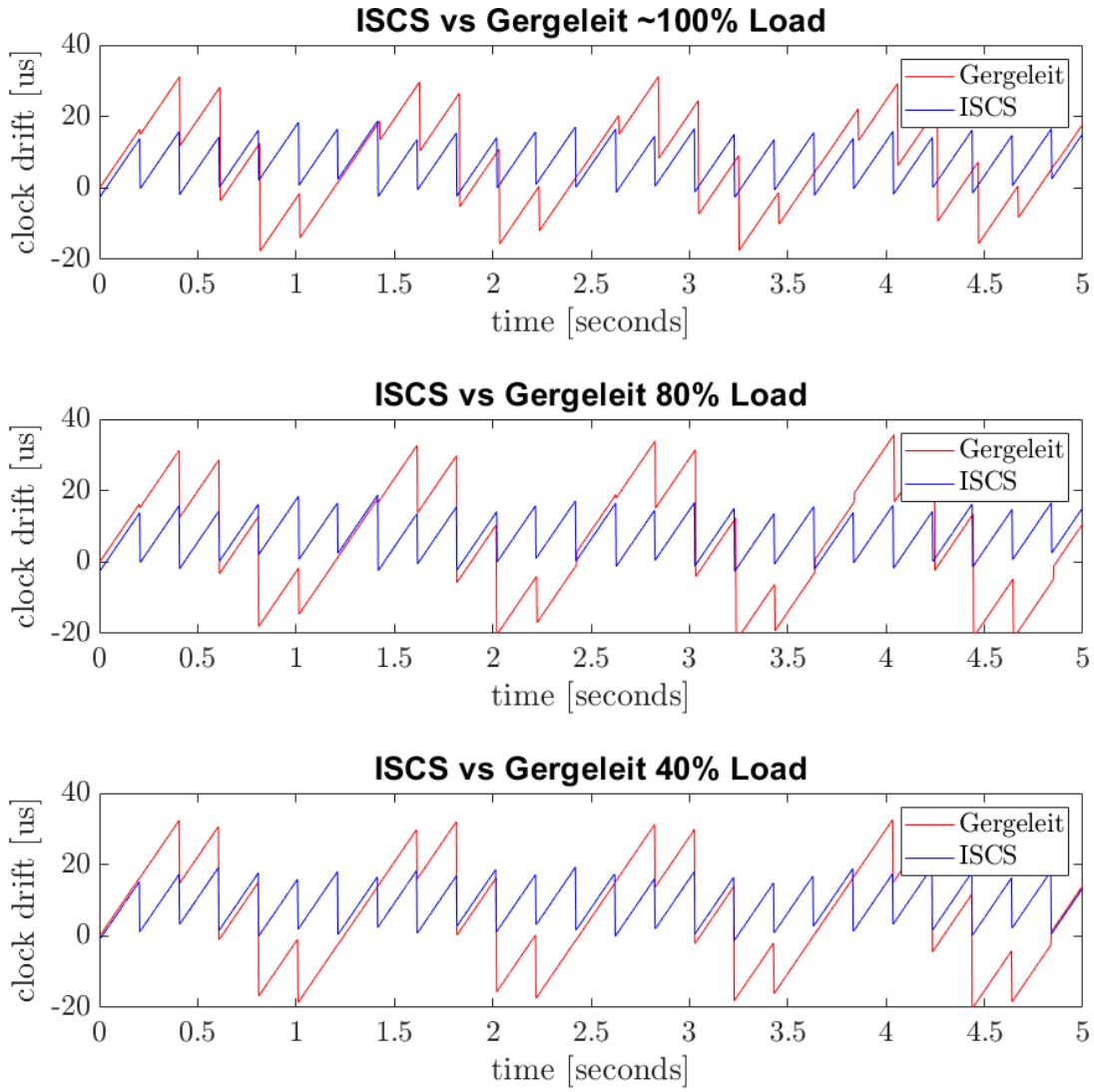

Figure 4.7: Clock Drift Comparison of ISCS and Gergeleit's Method on SDCC Under Different Bus Loads (with 81 ppm Clock Drift)

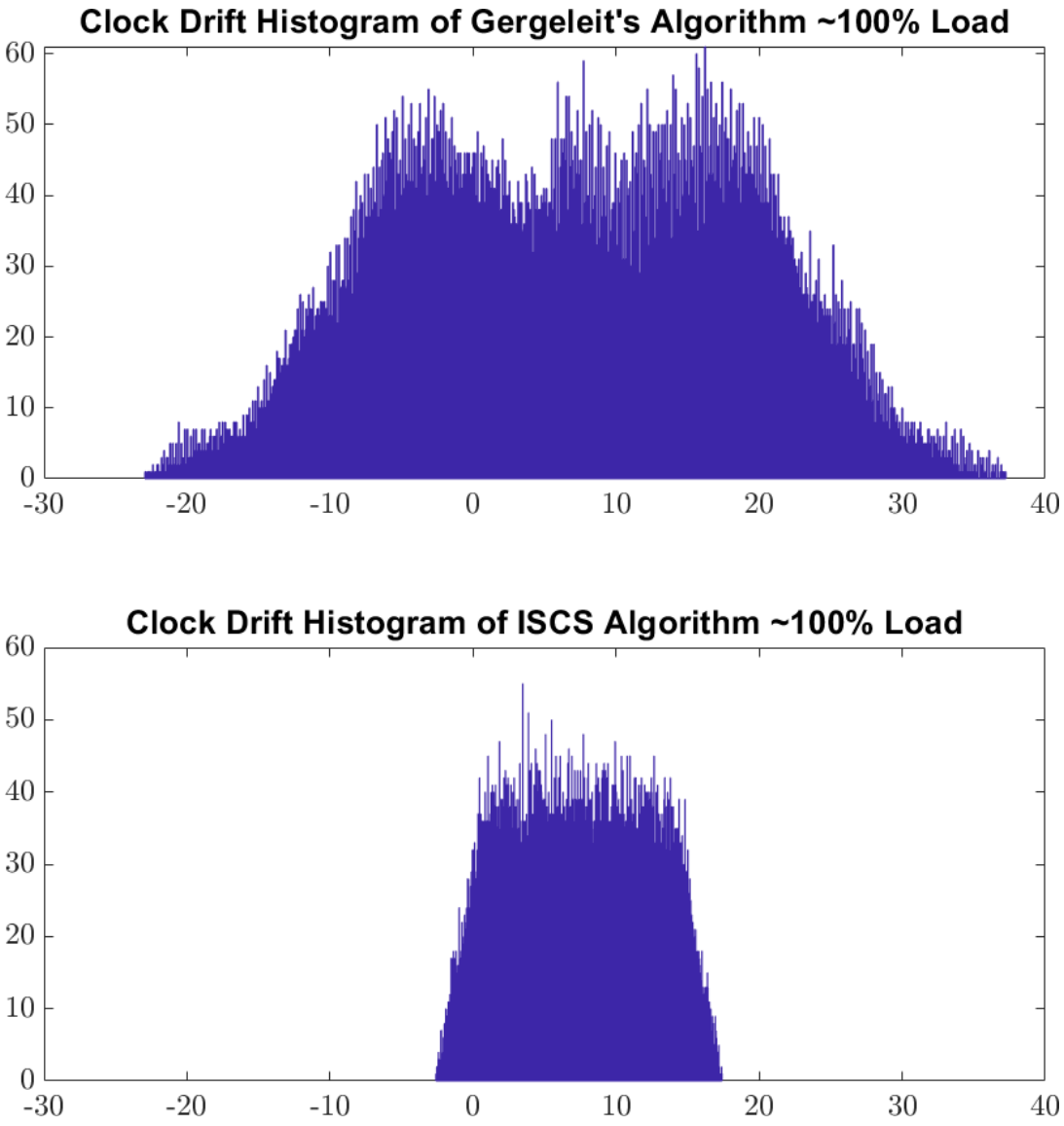

Figure 4.8: Clock Drift Histograms of ISCS and Gergeleit's Method on SDCC (with 81 ppm Clock Drift)

CHAPTER 5

HARDWARE APPROACH TO CLOCK SYNCHRONIZATION ON CAN

This chapter is concerned with the modification of the SDCC in order to implement the hardware-based clock synchronization algorithm in [19]. Section 5.1 briefly discusses the difficulties when modifying CAN controllers and Section 5.2 describes the hardware-based clock synchronization method in [19]. Implementation details for modifying the SDCC are given in Section 5.3 and experimental results are presented in Section 5.3.1.

5.1 Hardware Perspective on CAN Clock Synchronization

In addition to the software-based approaches given in Chapter 4, a hardware-based approach to clock synchronization on CAN will be investigated in this chapter. Differing from software-based approaches, hardware-based approaches focus on modifying or extending the CAN controller. Therefore, it requires thorough knowledge in hardware description languages such as VHDL or Verilog [42, 43]. Moreover, considering that the CAN controller itself is being modified, it is possible to end up violating the CAN protocol. On the other hand, hardware-based approaches are able to fully access the information in the underlying hardware. As a result, the algorithm for clock synchronization will be able to use all resources provided by the controller. Therefore, this may help to increase the overall performance of clock synchronization.

Unlike regular CAN controllers, the SDCC is a software-based controller that implements all CAN controller layers in the C programming language [37]. Hence, hardware approaches for clock synchronization on CAN can be implemented without diving into the hardware design, because all information embedded in regular CAN

controllers can directly be accessed from the application layer of the SDCC. Therefore, by using the SDCC, required modifications on the controller for clock synchronization can be implemented in software. In the following sections, a hardware-based solution called Phase Error-Based Clock Synchronization algorithm given in [19] will be described and its implementation on SDCC will be explained in detail. After that, its performance on the experimental setup will be evaluated.

5.2 Description of the PECS Algorithm

To provide hardware assistance to current software solutions for clock synchronization on CAN, the Phase-Error Based Clock Synchronization (PECS) algorithm is introduced in [19]. Differing from existing clock synchronization methods, PECS performs continuous clock correction between reference frames. Since the local clock of each node experiences the same clock drift as the CAN system clock, PECS suggests that the built-in phase error correction mechanism of CAN controllers can be used for clock correction.

Instead of directly using the phase error information provided by the CAN controller, the PECS algorithm keeps track of hard synchronization and soft synchronization events to avoid erroneous clock corrections. It performs clock correction when a soft synchronization takes place after the arbitration period. This is the case because different nodes may be transmitting at the same time until the winner of the contention period is decided during the arbitration period. Therefore, if a clock correction is performed using the phase error information during the arbitration period, it may create additional clock differences between nodes. For this reason, PECS dictates all nodes to synchronize with the transmitter, once the arbitration period is completed. Also, since different nodes access the bus during the acknowledgement period for sending an ACK or NACK, PECS does not perform clock synchronization during the acknowledgement period of a CAN frame.

The valid and invalid edges for the clock correction performed by the PECS algorithm are illustrated in Figure 5.1. At the time instants t_1 and t_4, node B and C observe that arbitration and acknowledgment are taking place, hence they do not perform clock

correction. However, once they perform soft synchronization at the instants t_2 and t_3, they correct their clock according to PECS, since it is certain that only one node is transmitting. According to PECS, at the time instants t_2 and t_3, node B and C read the calculated phase error from the CAN controller and update their clocks according to (5.1), where TQ represents the nominal period of the CAN system clock and e_p represents the calculated phase error.

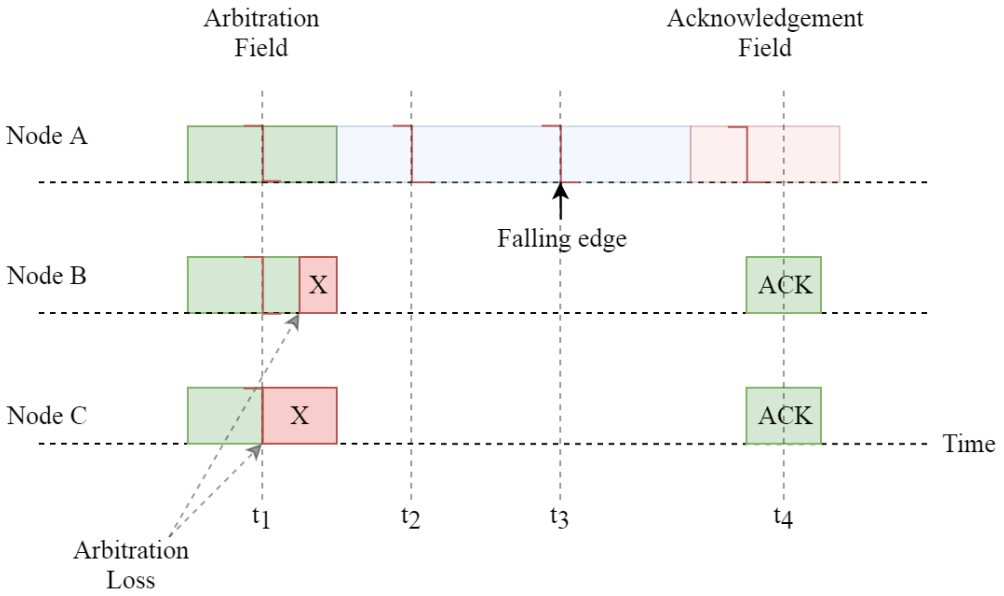

Figure 5.1: Valid Clock Correction Edges for PECS

$$virtual\_clock = virtual\_clock - (e_p \cdot TQ) \tag{5.1}$$

In this equation, the value of TQ shall be calculated according to the relation between the virtual clock and CAN system clock to perform clock correction. In order to calculate it, (5.2) can be used.

$$TQ_{virtualclockticks} = virtual\_clock\_frequency/(quanta\_per\_bit \cdot CAN\_bit\_rate) \tag{5.2}$$

Unlike other algorithms, PECS is designed to increase the accuracy of the existing clock synchronization algorithms by providing some hardware support. It does not focus on synchronizing different nodes with reference frames. Instead, it continuously performs clock correction according to phase error for reducing the clock drift between reference frames. Hence, without a method that focuses on synchronizing

different nodes, PECS will not solve the clock drift problem. Instead, it should be integrated to existing algorithms like Gergeleit and ISCS for better performance.

5.3 Implementation of the PECS Algorithm on SDCC

Differing from Gergeleit's method and ISCS, PECS requires modifications on the CAN controller hardware. This is because PECS algorithm needs access to the phase error value calculated by the controller hardware and most of the time it is not accessible. However by using SDCC, the user can directly access all information that the CAN controller calculates or uses. Hence, accessing the phase error with software becomes possible. Unlike other clock synchronization methods, PECS does not require transmission of a CAN frame, because it uses the services of other algorithms such as Gergeleit's method or ISCS. Therefore, different from the implementations in Section 4.1 and Section 4.2, the application layer primitives such as data request and data indication will not be used. Instead, a lower layer primitive that calculates the phase error will be modified for implementing PECS.

The PCS layer primitive that is responsible for phase error calculation/correction is called "$quantumclock\_m\_ind$". It is invoked by the PMA layer on the edges of the nominal bit time. In addition to its phase error related operations, it detects the edges on the bus and performs soft and hard synchronization accordingly. Since PECS requires the phase error information and needs to detect the instants where hard and soft synchronization take place, it can be implemented by modifying the function $quantumclock\_m\_ind$. This modification is illustrated in Source Code 15, by omitting the parts that are not changed.

As is given in Source Code 15, the function $quantumclock\_m\_ind$ performs edge detection at the beginning of its execution. Then it calculates the phase error and performs a hard or soft synchronization if necessary. Since PECS should be performed after a soft synchronization, the code block given in lines 10-18 of Source Code 15 is added. In this code block, it is ensured that a soft synchronization is performed and the phase error is checked to be nonzero to perform virtual clock correction. Additionally, since PECS dictates that clock correction shall only be

performed between arbitration and acknowledgement fields of a CAN frame, this is checked by using the data structures provided by the SDCC. The state of the RX_FSM (`pcs->mac->state.rx_fsm_state`) explained in Section 3.1, is checked to be between identifier and acknowledgment states for filtering the valid conditions for virtual clock correction. Once it is ensured that conditions are met for PECS clock correction, the virtual clock is updated according to the Equation 5.1.

Source Code 15 Required modification on SDCC for PECS Algorithm

```
static void quantumclock_m_ind(
    struct CAN_XR_PCS *pcs, unsigned long ts, int bus_level)
{
  /* Edge detection and Phase Error Calculation */
  /* ... */

  /* Checking Soft or Hard Sync is necessary */
  /* ... */

  /* Added for PECS Algorithm */
  /* Check if edge is valid for PECS */
  if (pecs_enabled && soft_sync && (phase_error != 0) &&
     (pcs->mac->state.rx_fsm_state > CAN_XR_MAC_RX_FSM_RX_IDENTIFIER) &&
     (pcs->mac->state.rx_fsm_state < CAN_XR_MAC_RX_FSM_RX_ACK))
  {
    set_virtual_clock(read_virtual_clock() -
                      (phase_error * TQ_VCLOCK_CYCLES));
  }

  /* Sampling and transmission request handling */
  /* ... */
}
```

Please note that enabling PECS algorithm does not require modifications on the upper layer clock synchronization methods, since PECS operates independently. Just providing virtual clock access to PECS is enough for PECS and upper layer algorithm to work together.

5.3.1 Results

As described before, the PECS algorithm requires an additional reference message-based method to perform clock synchronization between CAN nodes on the network. As a result, evaluating its performance alone is not meaningful. Therefore, in this section, the performance of PECS will be evaluated by pairing it with two software-

based clock synchronization algorithms, ISCS and Gergeleit's Method. In order to perform data collection and to calculate average and maximum clock drift values, the exact methods described in Section 4.3 will be used.

For performance evaluation, PECS is paired with ISCS and Gergeleit's method under three different bus loads of 100%, 80% and 40%. As in Section 4.3, the reference frame period is selected as 200 ms and the virtual clock of each node is sampled at a period of 2.5 ms for 65 seconds. After repeating each case over 15 times, the average and maximum clock drift values for different bus loads in Table 5.1 and 5.2 are obtained. Note that the results in Table 4.1 are also added to this table for observing the performance difference after PECS is enabled in both of the methods.

As can also be seen from Figure 5.2, on which the clock drift performance of PECS on ISCS and Gergeleit's method are compared, the PECS algorithm decreases the average and maximum clock drifts in both algorithms significantly. It improves the performance of Gergeleit's method by 40% under full bus load, when the average clock drift is considered. Similarly, the average clock drift for ISCS is decreased by 60% under full load when PECS is enabled. This decrease in the average and maximum clock drift is a direct result of performing clock corrections between reference frames according to the phase error, because the virtual clock of each node experiences the same amount of drift with its local CAN clock. Also note that, when bus load is decreased, it is observed that the improvement in the clock drifts also drops for both algorithms. Under 40% bus load, PECS increased the performance of Gergeleit's method and ISCS by 23% and 28%, respectively. This decrease in the improvement can be explained with the dependency of the number of clock corrections performed by PECS on the number of frames (signal edges) on the bus. Since PECS performs clock corrections after soft synchronizations, decreasing the bus load also decreases the number of clock corrections performed, as it can clearly be seen from Figure 5.3 and 5.4. As a result, it can be stated that the performance improvement of PECS increases with the bus load.

Table 5.1: Performance of Gergeleit's Method with PECS on SDCC Under Different Bus Loads (with 81 ppm Clock Drift)

Parameters		**Gergeleit's Method [18]**	**Gergeleit's Method [18] + PECS [19]**
Average Clock Drift	100% Load	11.37 us	7.09 us
	80% Load	11.60 us	8.69 us
	40% Load	11.62 us	8.92 us
Maximum Clock Drift	100% Load	37.26 us	21.92 us
	80% Load	35.65 us	23.89 us
	40% Load	36.9 us	26.81 us

Table 5.2: Performance of ISCS with PECS on SDCC Under Different Bus Loads (with 81 ppm Clock Drift)

Parameters		**ISCS [19]**	**ISCS + PECS [19]**
Average Clock Drift	100% Load	7.51 us	2.72 us
	80% Load	7.78 us	3.67 us
	40% Load	9.22 us	6.67 us
Maximum Clock Drift	100% Load	17.49 us	15.94 us
	80% Load	19.51 us	18.9 us
	40% Load	19.65 us	19.01 us

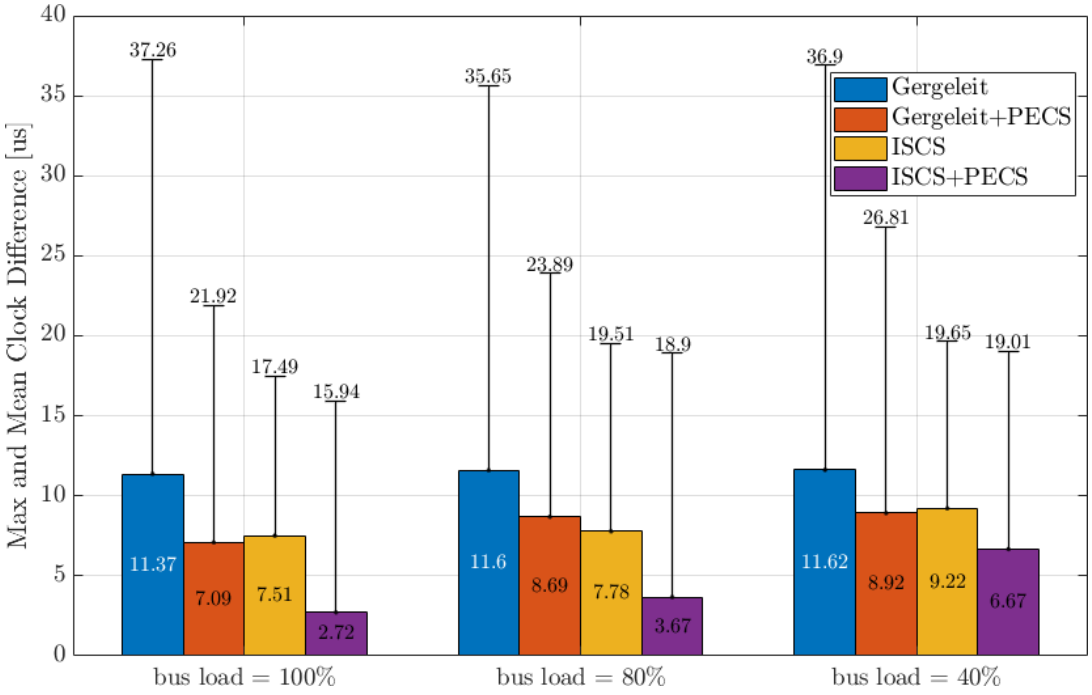

Figure 5.2: Performance of PECS on SDCC Under Different Bus Loads (with 81 ppm Clock Drift)

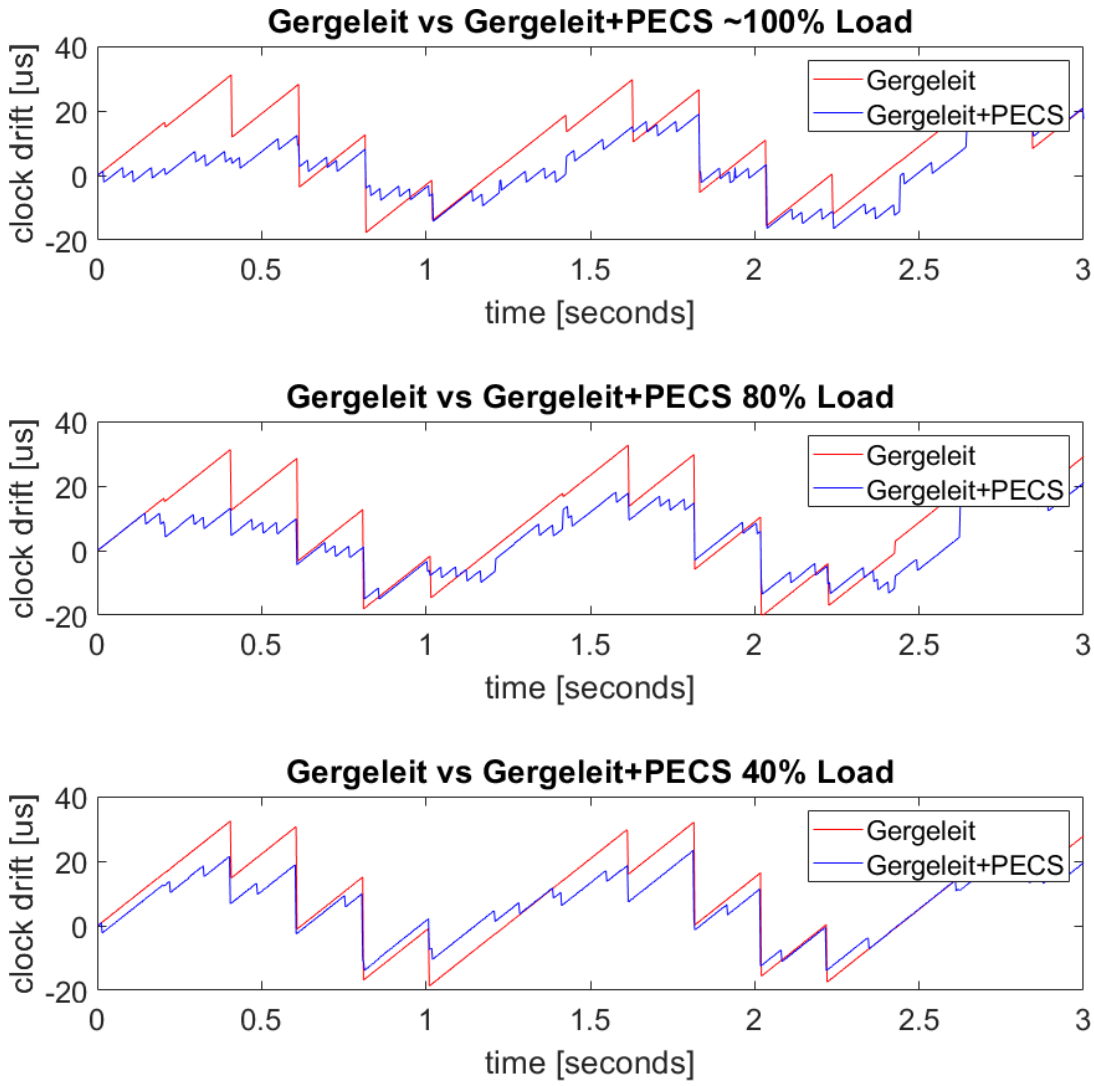

Figure 5.3: Clock Drift Comparison of Gergeleit's Method with PECS on SDCC Under Different Bus Loads (with 81 ppm Clock Drift)

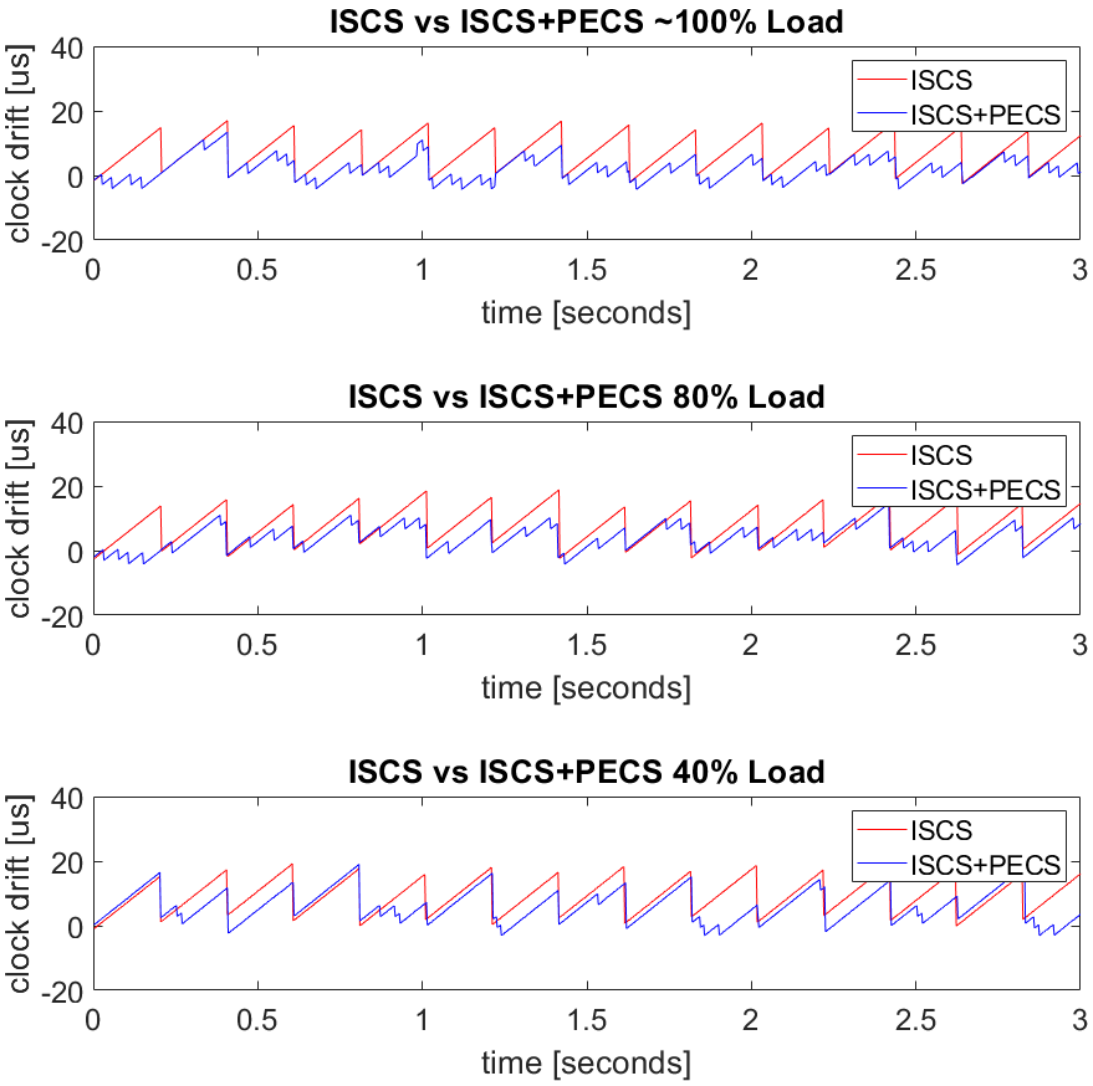

Figure 5.4: Clock Drift Comparison of ISCS with PECS on SDCC Under Different Bus Loads (with 81 ppm Clock Drift)

CHAPTER 6

CONCLUSION

The Controller Area Network (CAN) is the most widely used in-vehicle communication bus. Although CAN has been used without modifications since its official release in 1986, the recent advances in automotive technology require additional features of CAN such as clock synchronization. Since the CAN functionality is generally implemented in custom hardware, the development of such modifications generally requires a large effort.

Accordingly, this thesis is based on the realization of a Software-Defined CAN Controller (SDCC), which implements the CAN functionality in software and hence allows modifications to the original CAN protocol. The thesis first describes the layered architecture of the SDCC including the working principles of the underlying state machines. Then, an examination of the hardware dependency of the SDCC enables the deployment on different platforms and the steps followed for porting the hardware dependent modules to the experimental setup are given. In order to establish the functionality of the SDCC, different software-based clock synchronization algorithms for CAN are realized on the application layer of the SDCC. These algorithms include Gergeleit's method and the novel Improved Software-based Clock Synchronization (ISCS) algorithm. Following the implementation of the software-based methods, the possibility of modifying the CAN controller is exploited by implementing a hardware-based clock synchronization approach which uses the bit-level timing of CAN. The Phase Error-Based Clock Synchronization (PECS) is first explained in detail and then implemented by modifying the Physical Coding Sub-layer of the SDCC. Finally, the performance of all the clock synchronization algorithms is evaluated and compared based on experiments.

The implementation of the SDCC on the experimental setup proved that communication between CAN nodes can be established without a typical CAN controller implemented in hardware. Furthermore, it is shown that modifications or extensions on CAN can be performed without requiring a hardware design of the CAN controller. Additionally, the results of clock synchronization algorithms implemented on SDCC indicated that promising clock precision values can be achieved without any hardware controller support and clock synchronization algorithms that require modifications of the CAN controller can be validated in software.

The future work includes porting the SDCC to different microcontrollers and increasing the maximum CAN bit rate that can be achieved. Additionally, the bit timing and bus synchronization primitives of SDCC can be converted into interrupt-driven functions to enable the user to perform other tasks while the SDCC is waiting for an interrupt to advance its state machine.

REFERENCES

[1] M. Lévesque and D. Tipper, "A survey of clock synchronization over packet-switched networks," *IEEE Communications Surveys Tutorials*, vol. 18, pp. 2926–2947, Fourthquarter 2016.

[2] ISO, ISO11898, *Road Vehicles—Interchange of Digital Information— Controller Area Network (CAN) for High-Speed Communication*, 1993.

[3] K. W. Schmidt, "Robust priority assignments for extending existing controller area network applications," *IEEE Transactions on Industrial Informatics*, vol. 10, pp. 578–585, Feb 2014.

[4] A. Valenzano and G. Cena, "Controller area networks for embedded systems," in *Networked Embedded Systems* (R. Zurawski, ed.), pp. 15–1–15–38, Boca Raton, FL, USA: CRC Press, 2009.

[5] R. Davis, A. Burns, R. Bril, and J. Lukkien, "Controller area network (CAN) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, pp. 239–272, 02 2007.

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

[7] ISO, ISO17458, *Road Vehicles — FlexRay Communications System*, 2013.

[8] M. Grenier, L. Havet, and N. Navet, "Pushing the limits of CAN - scheduling frames with offsets provides a major performance boost," in *Proc. of the 4th European Congress Embedded Real Time Software*, January 2008.

[9] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, M. Walther, and R. Bosch, "Time triggered communication on CAN," *Proceedings of the 7th Int. CAN Conference, Amsterdam, The Netherlands*, 2000.

[10] G. Cena, I. C. Bertolotti, T. Hu, and A. Valenzano, "CAN with extensible in-frame reply: Protocol definition and prototype implementation," *IEEE Transactions on Industrial Informatics*, vol. 13, pp. 2436–2446, Oct 2017.

[11] G. Bloom, G. Cena, I. C. Bertolotti, T. Hu, and A. Valenzano, "Supporting security protocols on CAN-based networks," in *2017 IEEE International Conference on Industrial Technology (ICIT)*, pp. 1334–1339, March 2017.

[12] T. Ziermann, S. Wildermann, and J. Teich, "CAN+: A new backward-compatible controller area network (CAN) protocol with up to 16× higher data rates.," in *2009 Design, Automation Test in Europe Conference Exhibition*, pp. 1088–1093, April 2009.

[13] J. Ferreira, P. Pedreiras, L. Almeida, and J. A. Fonseca, "The FTT-CAN protocol for flexibility in safety-critical systems," *IEEE Micro*, vol. 22, pp. 46–55, July 2002.

[14] G. Cena, I. Bertolotti, T. Hu, and A. Valenzano, "On a software-defined CAN controller for embedded systems," *Computer Standards & Interfaces*, vol. 63, pp. 43–51, March 2019.

[15] A. Diarra, T. Hogenmueller, A. Zimmermann, A. Grzemba, and U. A. Khan, "Improved clock synchronization start-up time for ethernet avb-based in-vehicle networks," in *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, pp. 1–8, Sep. 2015.

[16] S. Tuohy, M. Glavin, C. Hughes, E. Jones, M. Trivedi, and L. Kilmartin, "Intra-vehicle networks: A review," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, pp. 534–545, April 2015.

[17] L. L. Bello, R. Mariani, S. Mubeen, and S. Saponara, "Recent advances and trends in on-board embedded and networked automotive systems," *IEEE Transactions on Industrial Informatics*, vol. 15, pp. 1038–1051, Feb 2019.

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

[19] M. Akpinar, K. W. Schmidt, and E. G. Schmidt, “Improved clock synchronization algorithms for the controller area network (CAN),” in *2019 28th International Conference on Computer Communication and Networks (ICCCN)*, pp. 1–8, July 2019.

[20] W. Stallings, *Data and Computer Communications*. New Jersey: Pearson Education, Inc, 10 ed., 2014.

[21] L. E. Frenzel, *Handbook of Serial Communications Interfaces: A Comprehensive Compendium of Serial Digital Input/Output (I/O) Standards*. Oxford: Elsevier Science & Technology, 1 ed., 2016.

[22] R. Z. Uwe Koppe, Christian Schlegel, “The future of CANopen FD.” [Online]. Available: http://www.microcontrol.net/download/presse/interview_canopen_fd_koppe_cia_en.pdf, [Accessed 20 Jan. 2020].

[23] C. Cristian, Flaviu & Fetzer, “The timed asynchronous distributed system model. parallel and distributed systems,” *IEEE Transactions on. 10. 642 - 657*, 1999.

[24] C. A. Latha and H. L. Shashidhara, “Clock synchronization in distributed systems,” in *2010 5th International Conference on Industrial and Information Systems*, pp. 475–480, July 2010.

[25] A. Monot, N. Navet, and B. Bavoux, “Impact of clock drifts on CAN frame response time distributions,” in *ETFA2011*, pp. 1–4, Sep. 2011.

[26] H. Kopetz and W. Ochsenreiter, “Clock synchronization in distributed real-time systems,” *IEEE Transactions on Computers*, vol. C-36, pp. 933–940, Aug 1987.

[27] P. Veríssimo, L. Rodrigues, and A. Casimiro, “Cesiumspray : A precise and accurate global time service for large-scale systems,” *Real-Time Systems*, vol. 12, pp. 243–294, 05 1997.

[28] P. Verissimo and L. Rodrigues, “A posteriori agreement for fault-tolerant clock synchronization on broadcast networks,” in *[1992] Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing*, pp. 527–536, July 1992.

[29] F. Cristian, "Probabilistic clock synchronization," *Distributed Computing*, vol. 3, pp. 146–158, Sep 1989.

[30] K. P. Birman, "Clock synchronization and synchronous systems," in *Reliable Distributed Systems: Technologies, Web Services, and Applications*, pp. 493–508, New York, NY: Springer New York, 2005.

[31] J. B. G. Rodriguez-Navas and J. Proenza, "Hardware design of a high-precision and fault-tolerant clock subsystem for CAN networks," in *Proceedings of the 5th IFAC International Conference on Fieldbus Systems and their Applications, Aveiro, Portugal*, 2003.

[32] B. Donnelly and J. Cosgrove, "Achieving microsecond accuracy with 32 bit microcontrollers using the controller area network (CAN)," in *Proc. Irish Signals and Systems Confs*, pp. 508–513, 2004.

[33] C. Eriksson, H. Thane, and M. Gustafsson, "A communication protocol for hard and soft real-time systems," in *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*, pp. 187–192, June 1996.

[34] J. Allan and D. Lee, "Fault-tolerant clock synchronization with microsecond-precision for CAN networked systems," in *Proceedings of the 9th International CAN Conference*, 2003.

[35] Autosar, "Specification of Time Synchronization over CAN." [Online]. Available: https://www.autosar.org/fileadmin/user_upload/standards/classic/19-11/AUTOSAR_SWS_TimeSyncOverCAN.pdf, [Accessed 25 Dec. 2020].

[36] G. Rodriguez-Navas, S. Roca, and J. Proenza, "Orthogonal, fault-tolerant, and high-precision clock synchronization for the controller area network," *IEEE Transactions on Industrial Informatics*, vol. 4, pp. 92–101, May 2008.

[37] ISO, *ISO/IEC 9899:2018: Information technology — Programming languages — C*, 2018.

[38] STMicroelectronics, *Discovery kit with STM32F407VG MCU User Manual, UM1472*, 2017. Rev 6.

[39] STMicroelectronics, *STM32429I-EVAL evaluation board for the STM32F429 line User Manual, UM1667*, 2015. Rev 1.

[40] Microchip Technology, *MCP2551 High-Speed CAN Transceiver Datasheet, DS21667D*, 2003.

[41] Arduino, "ARDUINO MEGA 2560." [Online]. Available: https://store.arduino.cc/usa/mega-2560-r3, [Accessed 17 Jan. 2020].

[42] "IEEE Standard for VHDL Language Reference Manual," *IEEE Std 1076-2019*, pp. 1–673, Dec 2019.

[43] "IEEE Standard for Verilog Hardware Description Language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1–590, April 2006.