Software-Based AUTOSAR-Compliant Precision Clock Synchronization Over CAN

Florian Luckinger and Thilo Sauter, *Fellow, IEEE*

***Abstract*—Modern cars are characterized by a growing number of sensors, actuators, and advanced driver assistance systems that require a synchronized view of the time. These devices are typically interconnected by CAN, which is still the most important in-vehicle network. Precise clock synchronization over CAN is difficult due to the properties of the bus, and current approaches often use dedicated hardware or proprietary software solutions. A few years ago, however, the AUTomotive Open System ARchitecture (AUTOSAR) development alliance published a standardized synchronization method. In this article, we investigate what synchronization precision can be realistically achieved in a real-world automotive hardware and software environment. This involves pure software timestamping, standard CAN controllers without hardware modifications, and a typical automotive real-time operating system. We evaluate several approaches to reduce the synchronization jitter using filtering and optimizing the timestamping procedure. Experiments show that, ultimately, a precision better than 50 μs can be achieved with a fully AUTOSAR-compliant software implementation.**

***Index Terms*—Automotive embedded system, AUTomotive Open System ARchitecture (AUTOSAR), CAN, clock synchronization, real-time system, timestamping.**

I. Introduction

OVER the years, many in-vehicle networks have been developed to meet the needs of the increasing number of devices and data volume to be processed. In recent years, even Ethernet is gaining importance in the automotive sector [1]. Autonomous driving and, in general, driver assistance systems require sensors like cameras, RADAR, or light detection and ranging (LIDAR) and accentuate the need for high communication bandwidth [2], [3]. On the other hand, there are classical safety-relevant applications that do not require large amounts of data, such as anti-lock braking system (ABS), or traction control. In this domain, the controller area network (CAN) bus is still more than 30 years after its introduction, the predominant networking solution to connect electronic control units (ECUs), sensors, and actuators.

Automotive applications, such as all distributed systems, require a proper alignment of processes and, therefore, the synchronization of local clocks at the network nodes to achieve a common view on time. Proper synchronization is a basic requirement for various functions: from capture-and-replay methods during the development phase of ECUs to the coordination of sensor data acquisition and control actions during normal operation to the recording of error data for diagnostic purposes [4]. In the AUTomotive Open System ARchitecture (AUTOSAR), clock synchronization, therefore, plays a central role across the different network domains [5], with different mechanisms adapted to the capabilities of the individual networks and time gateways connecting the time domains [6]. For CAN, AUTOSAR defines a simple one-way reference broadcast scheme with a time master regularly distributing its local time to the connected slaves [7].

In packet-oriented networks, clock synchronization always relies on timestamps being drawn upon sending and receiving of dedicated synchronization messages. These timestamps are used to calculate the deviation between reference and local clocks and to steer the adjustment. Consequently, the accuracy of the synchronization is essentially determined by the accuracy of the timestamps. In this context, the timestamping jitter, i.e., the variation of the latency between the reception of the message and the actual drawing of the timestamp, plays a decisive role. Hardware timestamps are known to generally yield better performance than software timestamps [8] but require dedicated hardware support [9]. Especially in the cost-sensitive automotive domain, CAN controllers with hardware timestamping capabilities are not yet common. The relevant CiA 603 document specifying a hardware timestamping unit was released in 2017, and to date, implementations are only available in CAN IP (intellectual property) cores from big embedded system [9], not in classical single-chip controllers. If we consider that the entire market for automotive CAN nodes in 2018 was about 3.4 billion[1] and that IP cores are used typically in large ECUs, we may safely assume that the vast majority of CAN controllers will still have to rely on software timestamping for the near future.

In software-based solutions, timestamps are ideally drawn immediately after a frame is received by the controller, which

Manuscript received August 15, 2021; revised December 21, 2021 and January 22, 2022; accepted February 3, 2022. Date of publication February 9, 2022; date of current version July 11, 2022. This work was supported by TU Wien Bibliothek through its Open Access Funding Programme. Paper no. TII-21-3566. *(Corresponding author: Thilo Sauter.)*

Florian Luckinger is with Elektrobit, 1050 Vienna, Austria (e-mail: florian.luckinger@elektrobit.com).

Thilo Sauter is with the Institute of Computer Technology, TU Wien, 1040 Vienna, Austria, and also with the Department for Integrated Sensor Systems, University for Continuing Education Krems, 2700 Wiener Neustadt, Austria (e-mail: sauter@ict.tuwien.ac.at).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TII.2022.3149923.

Digital Object Identifier 10.1109/TII.2022.3149923

[1][Online]. Avilable: https://www.renesas.com/us/en/application/automotive/in-vehicle-networking-application

can be noticed either by actively monitoring the receive buffer or by using interrupts. With such approaches, synchronization accuracies in the range of several microseconds have been achieved [10]–[12]. However, these laboratory setups mostly used dedicated embedded software *only* for clock synchronization and ignored that, in practice, embedded platforms are used also for other tasks, which require some sort of operating system (OS). Therefore, in a realistic scenario, the influence of the OS on the synchronization performance should be taken into account, too.

The performance of software timestamping, and particularly the timestamping jitter, depends strongly on how network messages are handled by the OS. Evidently, the most accurate solution is to process incoming messages immediately via hardware interrupts, as nonpreemtible tasks lead to higher synchronization jitter [13]. On the other hand, in safety-critical real-time systems, interrupts are a source of nondeterminism and interfere with the normally preferred static scheduling approach. How to reconcile the two has been debated for a long time, with no real conclusion [14], [15]. The ISO 26262-6 standard strongly recommends a "restricted use of interrupts" in design of software architectures for safety-critical automotive systems [16]. Based on this, a pragmatic approach adopted by many automotive software developers in the industry is to use interrupts mainly for tasks that can be scheduled relatively well, such as the reading of sensors triggered by an ECU. For events that are genuinely asynchronous, such as network messages between subsystems or ECUs that are not globally synchronized, a more common approach is to poll the buffers of the communication controller regularly or to include dedicated preemption points in the scheduler [17], [18], which is roughly equivalent. According to our knowledge, straightforward preemption of running tasks by network communication is rather depreciated in industrial practice to avoid problems with schedule execution, even at the expense of higher communication jitter.

The purpose of this article is to analyze the possibilities to achieve precision clock synchronization over CAN in full compliance with the AUTOSAR standard in a *practical* automotive software environment, i.e., considering that clock synchronization is not the only task and that the embedded platform must also run an operating system. To this end, the study is based on two assumptions. First, we consider implementations using only software timestamping, meaning that they can rely on standard CAN controllers not needing hardware extensions or dedicated hardware solutions. Second, we focus our investigation on a real-world automotive real-time operating system using cooperative (nonpreemptive) round-robin task scheduling without interrupts.

The rest of this article is organized as follows. Section II reviews related work in the field of software-based clock synchronization of CAN. Section III introduces the AUTOSAR clock synchronization basics. Sections IV and V present the timestamping approach and the synchronization strategy. Section VI describes the experimental testbed and the baseline results, whereas Sections VII and VIII evaluate performance improvements. Finally, Section IX concludes this article.

II. Related Work

Clock synchronization over CAN has been a topic of interest for a long time in the CAN community, and many different solutions have been proposed in the course of time. Two main groups can be distinguished: The first targets time-division multiple access (TDMA) schemes for time-triggered communication, where clock synchronization is a low-level network service to align time slots. This usually is done by exploiting the bit timing properties of CAN, and often requires dedicated hardware to implement timestamping. A classical example is time-triggered CAN (TTCAN) level 2, which introduces a dedicated Network Time Unit to synchronize the local time with the network time [19], reaching a synchronization precision around 10 μs. Also, Carvalho and Pereira [20] implemented a TDMA scheme with dedicated hardware. The theoretically achieved precision was less than 1 μs, however, at the expense of an extremely high resynchronization rate leading to a high bus load, which made the approach impractical.

A different approach was investigated in [21] for a CAN-like on-chip network in a multiprocessor system on chip (SoC). The synchronization protocol is based on a simple synchronization message and partly implemented in hardware. Therefore, a precision of 2 μs can be achieved. Contrary to many other approaches, the authors also considered a real-time microkernel which accommodated multiple applications and was tightly connected to the TDMA slots.

A less tight TDMA approach was investigated in [22]. In their theoretical study, the authors investigated clock synchronization to reduce bus contentions between multiple nodes and to improve scheduling. Based on a pure software solution and taking into account also pre- and postprocessing times of data in the applications, they postulated that a synchronization precision of 1 ms was sufficient.

The approach in [23] was inspired by TTCAN, although not limited to time-triggered systems. It proposesd an additional hardware module for timestamping and clock synchronization, to be implemented in parallel and independent of the actual CAN controller. The evaluation showed an achievable precision of 20 μs.

The second group is more relevant for this article and comprises event-triggered communication schemes. The approach in [10] is one of the earliest and was the basis and inspiration for many subsequent solutions. It used simple state correction and a short, from today's viewpoint, impractical synchonization interval to achieve a precision in the range of 20 μs. The approach was taken up by other authors, such as [11], who obtained a precision of $\sim$10 μs.

The study in [24] examined the use of low cost hardware in network control systems. It used precision time protocol (PTP) with modifications necessitated by the restricted payload in CAN, and a message format similar to the one defined by AUTOSAR. The timestamping was completely software based, and the authors achieved a precision better than 100 μs, which was limited by the clock resolution.

In [25], a simulation-based analysis of two synchronization approaches was presented. The analysis of an algorithm based

TABLE I
COMPARISON OF RELATED WORK

Solution	Precision	Comm. scheme	Timestamping mechanism	HW support required	OS considered	Synchronization mechanism	Synchronization strategy
TTCAN level 2 [19]	∼10 µs	TDMA	Hardware	yes	no	ref. broadcast	rate adjust
Carvalho et al. [20]	∼1 µs	TDMA	Hardware	yes	no	implicit slot timing	phase adjust
Breaban et al. [21]	∼2 µs	TDMA	Hardware	yes	CoMik µkernel	mod. AUTOSAR	state adjust
Daigmorte et al. [22]	∼1 ms	TDMA	Software	no	implicit	ref. broadcast	state adjust (?)
Rodriguez-Navas et al. [23]	∼20 µs	any	Hardware	yes	no	ref. broadcast	rate adjust
Gergeleit et al. [10]	∼20 µs	event-triggered	Interrupt	no	no	ref. broadcast	rate adjust
Lee et al. [11]	∼10 µs	event-triggered	Interrupt	no	no	ref. broadcast	state adjust
Marti et al. [24]	∼100 µs	event-triggered	Interrupt	no	no	PTP + ref. broadc.	state adjust
Akpinar et al. [25] (simul.)	∼9 µs	event-triggered	immediate	no	no	ref. broadcast [10]	state adjust
Akpinar et al. [25] (simul.)	∼4 µs	event-triggered	immediate	no	no	ref. broadcast	phase adjust
Akpinar et al. [12]	∼5 µs	event-triggered	Interrupt	no	no	ref. broadcast [10]	rate adjust
Akpinar et al. [12]	∼100 µs	event-triggered	Interrupt	no	no	AUTOSAR	state adjust

on [10] offering simple clock state correction yields a worst-case precision of about 9 μs, depending on the synchronization interval. The more sophisticated second approach uses information about the bit timing of CAN, provides clock drift compensation, and reaches a precision of $\sim$4 μs. In [12], the authors presented experimental results for their clock drift correction approach, achieving about 5 μs precision. Interestingly, they also evaluated the native AUTOSAR synchronization solution with simple state correction. This solution achieved $\sim$100 μs for a synchronization interval of 1 s.

Table I presents an overview of the related work. A typical characteristic of most of the approaches is that they focus only on the clock synchronization. Performance data are mostly based on embedded evaluation platforms running only CAN communication and synchronization tasks. Interdependencies with operating systems (even lean ones) that are inevitable once the system has to fulfill other tasks are rarely considered. Accordingly, timestamping is mostly assumed to be immediately available after message reception, and it is typically implemented using interrupts that are quickly serviced. These solutions can reach a synchronization precision in the μs range; however, they do not go together well with the requirements discussed in the previous section, in particular the one that network communication should not preempt other tasks. Therefore, we investigate in this article what performance can be attained if such requirements are taken into account.

III. AUTOSAR Clock Synchronization Over CAN

In this article, we consider only local synchronization, i.e., the internal synchronization of a group of nodes, irrespective of the global time (which could be given, e.g., by GPS). We consider a distributed system with N participants, each with its own clock C_i. In the ideal synchronized case, all N clocks show the same value at any time. In real systems, however, deviations occur. A measure for the synchronization quality of distributed systems is the precision Π. It indicates the maximum deviation between the clocks in the system and can be defined as

$$\Pi = \max_{i,j \in 1,2,\ldots,N} |C_i(t) - C_j(t)| . \quad (1)$$

The clocks used in this article are adder based. A signal is generated by an oscillator with a fixed frequency. This signal is used as a basis for the local clock, and with each oscillator

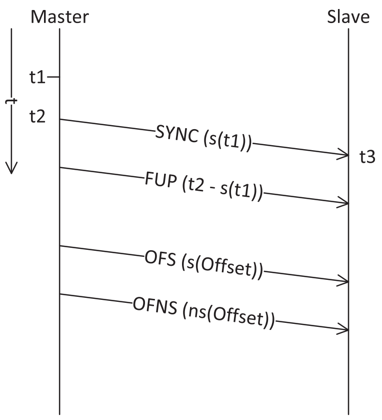

Fig. 1. Clock synchronization message exchange over CAN in AUTOSAR.

tick, a preset value is added to the current clock value. By changing the clock increment, the speed of the clock can be adjusted. This allows for compensating *rate* differences between clocks, in addition to setting the clock *state* to a given value.

AUTOSAR clock synchronization over CAN [7] adopts a master–slave reference broadcast scheme tailored to the properties of the CAN bus. In particular, it has to cope with the 8-B limit for the user data. Fig. 1 shows the message exchange. The master first takes timestamp t_1 from the local clock. The seconds part of t_1 is sent to the slave with the SYNC message. The SYNC message is timestamped when sent by the master (t_2) and when received by the slave (t_3). The time difference between t_1 and the transmit timestamp t_2 is transmitted using the follow-up (FUP) message. After exchange of the SYNC and FUP messages, the slave knows the timestamps t_2 and t_3 and can use them for synchronizing its clock. The reason for splitting the information transfer and taking the additional timestamp t_1 lies in the limited payload of CAN. Time in AUTOSAR is encoded as a 64-b number with nanosecond resolution [9]. Together with additional information bits needed to distinguish the individual messages and possible counter overflows, 8 B are not enough to include everything in one CAN frame. The OFS and OFNS messages are optional and are used for the transmission of an independently measured clock offset.

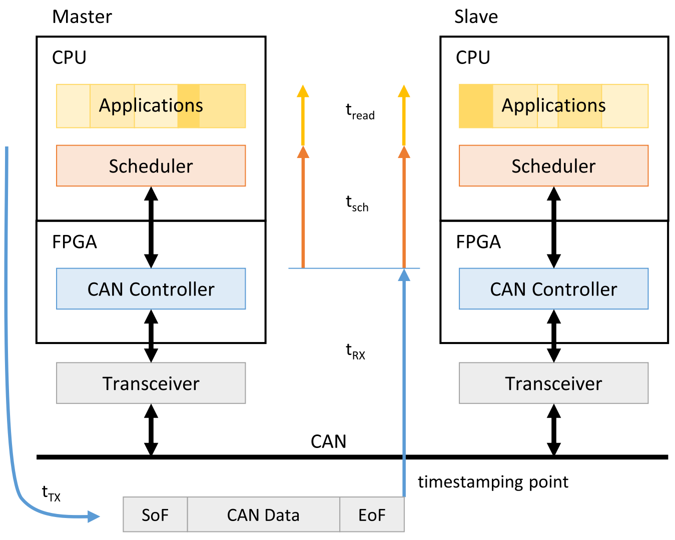

Fig. 2. System model indicating essential delay components. Only the receive path is relevant for clock synchronization.

IV. Software Timestamping Approach

A peculiarity of the AUTOSAR clock synchronization over CAN standard is that the reference point for timestamps is the end of the frame, not the start, like in other synchronization protocols. This actually helps to overcome the timing uncertainties of the bus access. The synchronization task runs on the CPU and is invoked cyclically by the scheduler. When the send function is called, the data to be sent are placed in the send buffer, and the function returns. It would be straightforward to use this instant to draw the send timestamp; however, the actual transmission depends on the bus load and the priority of the SYNC message. Even if it has the highest priority, sending must wait if another transmission is currently in progress and the bus is busy.

When a CAN frame is sent, it is read back from the bus by the sender, too, to check for possible error flags and the acknowledgement flag. If transmission was successful, the controller places the frame in the receive buffer like any other frame received over the bus. The CAN receive task periodically reads this buffer and notifies the application, which will then read the clock counter value. The send and receive timestamps can, therefore, be generated with the same mechanism. This situation is reflected in Fig. 2, which shows the system model with the most essential components and delays [26].

Owing to the timestamp being drawn at the end of the frame, the transmission latency t_{TX} plays no role. For the purpose of this article, we may safely assume that the SYNC frames are simultaneously received by master and slaves so that the receive latency t_{RX} plays no role either. In reality, there will be some jitter in the frame processing inside the CAN controller, as well as the propagation delay along the bus line. All of these delays are, however, hardware related, and as we are analyzing software timestamping, these delay components can be considered constant and negligible.

The latency until the timestamp is finally available in the application process is ultimately made up of the following two components.

1) t_{sch} is the delay caused by the real-time scheduler. It describes the time span between receipt of the message in the message memory of the CAN controller and the callback in the software. Under ideal conditions, the maximum value of this delay is equal to the period of the receive task plus the software delays that occur when the callback is called. However, under real conditions, this delay is larger because the receive task can be delayed by other tasks. In addition, the value increases when several messages are received at a time.
2) t_{read} is the delay in reading the clock value. It is the time span between calling the read function and the actual reading of the timestamp. It depends on the execution time of the software and is, therefore, not constant.

The general procedure of synchronization differs depending on the role of the participant. The master first reads the local time and stores the auxiliary timestamp t_1. The seconds part of t_1 is sent with the SYNC message. The SYNC message is received again by the master as soon as it was sent successfully, and the application is informed about it by a callback. When the callback is triggered, the local time is read and stored as timestamp t_2. From t_2 and the sent seconds part of t_1, the difference is calculated and sent with the FUP message. This ensures that the timestamps are related to the end of the CAN frame.

The slave always waits for incoming SYNC messages. If such a message is received, the local time is read and stored as timestamp t_3. From the SYNC message, the seconds part of t_1 is read and stored. After that the system waits for an FUP message with the same sequence number. The seconds part of the timestamp is read from this message, and t_2 is reconstructed. The slave then has t_2 from the master time domain and t_3 from its own time domain. From these two timestamps, the deviation of the two clocks can now be calculated and used for synchronization.

With t as the reference point at which the SYNC messages are placed in the receive buffers of the CAN controllers of the nodes, the timestamps are given by

$$t_2 = t + t_{\text{Master}} = t + t_{\text{sch,Master}} + t_{\text{read,Master}} \quad (2)$$

for the master side and

$$t_3 = t + t_{\text{Slave}} = t + t_{\text{sch,Slave}} + t_{\text{read,Slave}} \quad (3)$$

for the slave side. Given the identical timestamping procedure on both sides, the expressions are naturally the same. It must be noted, though, that the delay components are stochastic variables whose distributions may vary for each device.

V. Clock Synchronization

Based on the timestamps collected, the slave can adjust its local time to that of the master. A simple state correction is depreciated because of possible discontinuities in the time scale. Rate correction is commonly preferred, where the slave clock rate is adjusted to gradually reduce the difference between the local clock value $C_{\text{Slave}}(t)$ and the reference $C_{\text{Master}}(t)$ received from the master. The rate calculation is based on the last synchronization round i, where

$$P_{\text{Master,i}} = t_{2,i} - t_{2,i-1} \quad (4)$$

and

$$P_{\text{Slave,i}} = t_{3,i} - t_{3,i-1} - c_{\text{offs}} \tag{5}$$

are the current clock periods for master and slave, respectively, and c_{offs} accounts for a possible clock offset. If the clocks are perfectly syntonized, $P_{\text{Slave,i}} = P_{\text{Master,i}}$, and the ratio

$$R_i = P_{\text{Master,i}}/P_{\text{Slave,i}} \tag{6}$$

is, therefore, used as a correction factor to determine the adjusted clock tick length TL

$$TL_i = TL_{i-1} \cdot R_i \tag{7}$$

which is the increment for the adder-based clock. If in addition, a significant clock offset $t_3 - t_2$ exists, the clock rate is additionally changed to a temporary rate that is faster or slower than the nominal rate, and applied for a short time to compensate the offset quickly. As will be shown in Section VIII, rate filtering can be applied to reduce variations in R_i induced by the scheduler granularity. Algorithm 1 shows the pseudocode of the clock adjustment. Not explicitly shown in the algorithm is the startup phase. After power-up, the clock states will be undefined with respect to one another. Simple state correction is then performed to synchronize them before the algorithm switches to regular rate correction.

Algorithm 1: Synchronization Algorithm.

```
1:  function CANCallbackmessage
2:    $R \leftarrow$ Actual rate in nanoseconds per clock tick
3:    $\Delta_{\text{corr}} \leftarrow$ Amount of time corrected by offset
      correction in last round
4:    $t_{\text{Slave}} =$ getTime()
5:    $t_{\text{Master}} =$ getTimestampFromMessage(message)
6:    $R =$ RateCorrection($t_{\text{Master}}, t_{\text{Slave}}, \Delta_{\text{corr}}$)
7:    $\Delta_{\text{corr}} =$ OffsetCorrection($t_{\text{Master}}, t_{\text{Slave}}, R$)
8:  function RateCorrection$t_{\text{Master}}, t_{\text{Slave}}, \Delta_{\text{corr}}$
9:    $t_{\text{Master},-1} \leftarrow$ Master timestamp of last sync round
10:    $t_{\text{Slave},-1} \leftarrow$ Slave timestamp of last sync round
11:    $R_{-1} \leftarrow$ Rate calculated in last sync round
12:    $R_{\text{Max}} \leftarrow$ Maximum rate to be applied
13:    $R_{\text{Min}} \leftarrow$ Minimum rate to be applied
14:    $x = R_{-1} * (t_{\text{Master}} - t_{\text{Master},-1})/(t_{\text{Slave}} -$
       $t_{\text{Slave},-1} - \Delta_{\text{corr}})$
15:    if $x > R_{\text{Max}}$ then
16:    $x = R_{\text{Max}}$
17:    if $x < R_{\text{Min}}$ then
18:    $x = R_{\text{Min}}$
19:    if FilterActivated
20:    $R =$ ApplyFilter($x$)
21:    else
22:    $R = x$
23:    $t_{\text{Master},-1} = t_{\text{Master}}$
24:    $t_{\text{Slave},-1} = t_{\text{Slave}}$
25:    $R_{-1} = R$
26:    return $R$
27:  function OffsetCorrection$t_{\text{Master}}, t_{\text{Slave}}, R$
28:    $C \leftarrow$ Factor by how much the given rate can be
       changed in ppm
29:    $\Delta \leftarrow t_{\text{Master}} - t_{\text{Slave}}$
30:    $I \leftarrow$ Synchronization interval in nanoseconds
31:    $R_{\text{temp}} \leftarrow$ Temporary rate used for offset correction
32:    $N \leftarrow$ Number of ticks on which the temporary rate
       is applied
33:    $N_{\text{Max}} \leftarrow$ Maximum ticks that can be used for
       offset correction in this interval
34:    $N_{\text{Max}} = I/R$
35:    $N = |\Delta| * 1000000000/C$
36:    if $N > N_{\text{Max}}$
37:    $N = N_{\text{Max}}$
38:    CorrectionPerTick $=$
       sign($\Delta$) $* R * C/1000000000$
39:    $R_{temp} = R+$CorrectionPerTick
40:    $\Delta_{corr} =$CorrectionPerTick$*N$
41:    return $\Delta_{\text{corr}}$
```

VI. Experimental Setup

The synchronization algorithms are implemented on an EBX200 hardware platform from Elektrobit, a modular multipurpose embedded system that is used for automotive development tasks, such as restbus simulations and capture-replay. Especially the latter needs also high-accuracy clock synchronization. It consists of a PowerPC processor and an Altera Cyclone V field programmable gate array (FPGA). The processor contains two cores, one of which runs Linux and is responsible for connecting and transferring measurement data to the host PC via TCP/IP. Real-time tasks are executed on the other core, including the clock synchronization and message exchange via CAN.

The communication controller for the CAN bus which is implemented in the FPGA uses the MCAN IP core from Bosch [27]. This controller would offer an integrated function for recording both transmission and reception timestamps in the data link layer. However, these timestamps are drawn at the *beginning* of a frame, which is incompatible with the AUTOSAR specification mandating timestamping at the *end* of the frame. Therefore, this function cannot be used.

For the given purpose of this work, i.e., the investigation of purely software-based clock synchronization with a standard CAN controller, one could argue that an FPGA platform is unnecessary. On the other hand, the FPGA allows simple access to the internal signals needed for performance evaluation. Moreover, it permits monitoring without influencing system operation, which makes it an ideal development platform.

The EBX200 platform runs a real-time OS handling the tasks executed on the real-time core. It is based on a cooperative scheduler with a nominal period of $T_s = 500$ μs. Tasks include system functions like OS housekeeping, debugging functions that provide data from the real-time core to the host, the send and receive tasks for communication via CAN, FlexRay, or Ethernet, the clock synchronization function, and other user tasks. Being nonpreemptive, the scheduler does not support interrupts, so that access to interfaces can only be done by polling. For communication over CAN, this means that the receive buffer is checked

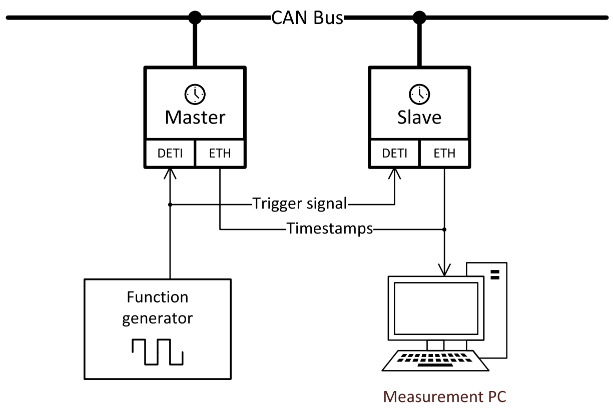

Fig. 3. Experimental setup (DETI: Digital event triggered input, ETH: Ethernet).

every scheduler period if a frame has been received. This gives a worst case latency of T_s until frames are registered by the software. As the scheduler is cooperative, the actual period may also vary slightly depending on the execution time of individual tasks.

For performance evaluation, two devices are synchronized via CAN using the standard resynchronization period $T_r = 3$ s specified by AUTOSAR. During this time, the worst case clock drift is given by the known expression

$$\Gamma = 2 \cdot \delta \cdot T_r \tag{8}$$

with δ as oscillator drift rate. The oscillators of the experimental platform have $\delta = 50$ ppm, which yields a drift of $\Gamma = 300$ μs during the synchronization interval. We notice that this is in the order of magnitude of the scheduler period, which means that the timestamping uncertainty will be the dominating factor influencing the synchronization precision. In the experiments, the master is left free running, and the synchronization is done on the slave side. One master and one slave are sufficient because the synchronization messages are one-way broadcasts from the master and additional slaves would not add complexity or extra bus load. For simplicity, and also with a view to the dominating impact of the OS, usual tests of corner cases such as rapid heating or cooling of individual oscillators are omitted. Instead, contrary to many related works, the experiments are conducted over several hours to collect a representative amount of data.

To determine the precision, the clocks on the master and the slave are read out periodically at the same time. This measurement must be minimally invasive and must not interfere with the other processes in the system. To this end, and to minimize measurement uncertainty, the reading of the clocks is triggered by digital signals. Fig. 3 shows the structure of the measurement setup. A function generator is used as the trigger source, generating a 10-Hz square wave signal. At each positive edge, a timestamp is generated in the FPGAs on both devices simultaneously. The generated timestamps are then read out by the software on the nonreal-time core and sent to the monitoring PC for data collection and processing. The difference $C_{\text{Slave,j}} - C_{\text{Master,j}}$ between the two associated clock samples on both devices describes the mutual deviation of the clocks and can then be used to determine the precision of the synchronization.

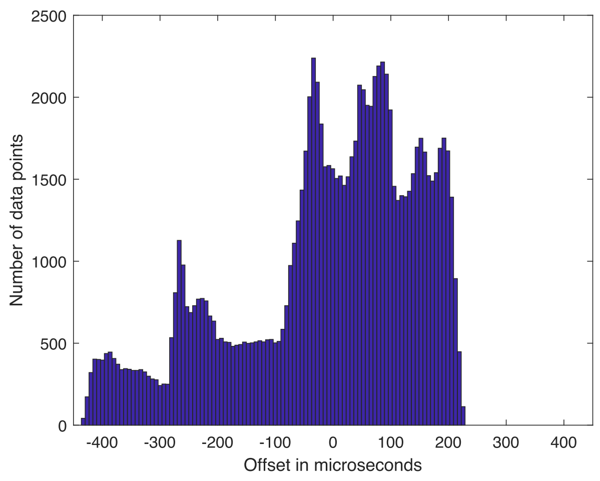

Fig. 4. Distribution of the master–slave clock offset using standard settings.

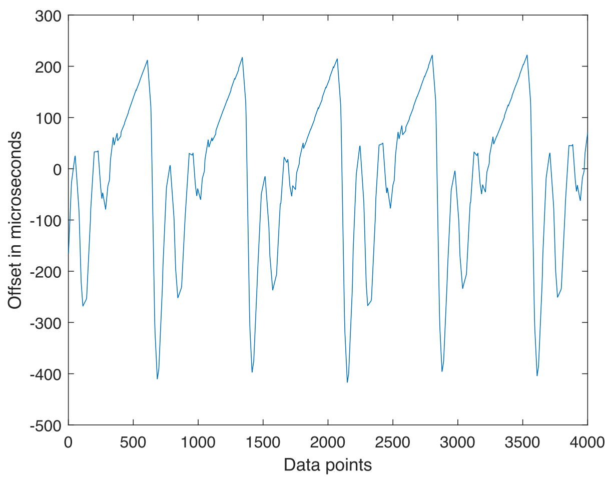

Fig. 5. Deviation between master and slave clock. Sampling frequency is 10 Hz, synchronization interval is 3 s.

For the baseline measurements, the platforms were operated with default software settings. Around 100 000 clock readings were recorded from both devices. With a trigger period of 10 Hz, this corresponds to a measurement duration of 2.78h. Fig. 4 shows the resulting histogram of the clock deviation. It reveals a sort of quantization effect that becomes more obvious if we look at a window of the time series data (see Fig. 5). The pattern that can be observed is a result of the polling-based frame reception strategy combined with the scheduler period. Depending on when the synchronization message is received within the cycle, there is a delay of up to T_s until the timestamp is actually drawn. As the schedulers of the two devices are not synchronized and the actual scheduler period may jitter depending on the load, the scheduler cycles of the two devices drift with respect to each other. This leads to “cycle slip” situations where the synchronization message slips into the next (or previous, depending on the relative clock drift) cycle and the clock deviation calculated from the timestamps suddenly sees a jump of $\pm T_s$ from one synchronization period to the next. The synchronization algorithm tries to correct this apparent offset quickly, which leads to an oscillatory behavior as seen in Fig. 5 that takes a few rounds to settle until a steady state is reached again.

VII. Improving Software Timestamps

As seen before, in a pure software solution, the timestamps are expected to exhibit large jitter due to the scheduling delay. For operating systems like the one used in this study, this delay is directly related to the polling period of the CAN receive task. But since the scheduler works cooperatively, the exact delay cannot be determined. One proven measure to reduce jitter is to bring the timestamp closer to the hardware [8]. Most solutions based on software timestamping use interrupts for this purpose. An interrupt is triggered as soon as a message is received from the CAN controller. However, as outlined in the introduction, this is not possible with the given OS and generally seen as critical in automotive systems. To improve synchronization performance, we subsequently investigate how to optimize the polling routine. This can be achieved by several measures, including the following.

Reducing the scheduler period allows to read CAN messages more often from the CAN controller, reducing the latency between receiving the message in the controller and calling the software callback. Under ideal conditions, when the task is always executed at the scheduled time, this measure promises a linear improvement of the timestamp accuracy. It should be noticed, though, that there is a tradeoff between the latency reduction due to more frequent task execution and the overhead introduced by task switching. This is a general issue of debate around fully preemptive versus nonpreemptive real-time OS [17].

Increasing the priority of the receive task will reduce the jitter of the timestamp, especially in the case of a contingency if the scheduler has many tasks in its queue. However, it does not protect against delays caused by tasks already running.

Timestamping in the driver brings the timestamping closer to the hardware. The CAN driver consists of several access functions to the CAN controller. These functions receive messages from the controller memory and then trigger the corresponding callbacks. Instead of waiting for the callbacks, the timestamp can be generated directly in the access functions. This leads to a reduction of the software dependency and, thus, to a reduction of the variable delay.

The measures mentioned make only limited sense in themselves because they mostly influence each other. Only a combination of these measures can provide substantial improvement, the most promising measure certainly being the reduction of the task period.

For the experiments, the scheduler period was reduced to $T_s = 100$ μs, which is the minimum possible to avoid excessive task switching. As supporting measure, the CAN receive task was given highest priority. Moreover, default tasks for other automotive communication systems like FlexRay were removed. The synchronization period of 3 s was kept unchanged. These optimizations are still in line with the device configuration for realistic automotive use cases. Fig. 6 shows the resulting histogram of the slave clock offset, and Fig. 7, a section from the time series data. As expected, the histogram is narrower thanks to the smaller quantization error resulting from the shorter scheduler period. The improvement is, however, not fully linear because there are still the cycle slip effects leading to marked over- and undershoots of the adjustment algorithm.

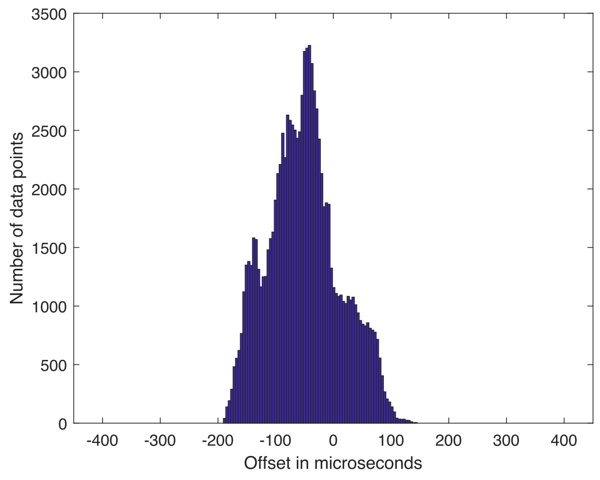

Fig. 6. Histogram of clock offset with improved scheduler settings.

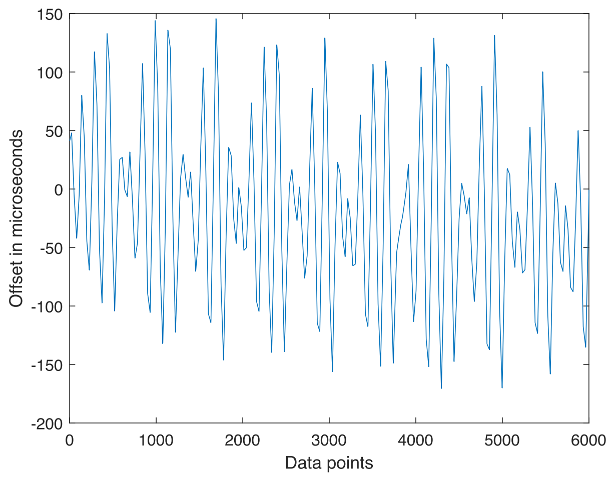

Fig. 7. Clock offset behavior with improved scheduler settings (window equals 10 min).

VIII. Rate Filtering

So far, the nominal clock rate has been calculated only from the last synchronization interval. The apparent challenge when using software timestamps is the enormous jitter induced by the scheduler, as seen in the previous sections. To reduce this influence, a filter can be applied to the calculated rate as is commonly done in clock synchronization. This idea is based on the assumption that the real rate of the reference clock changes only very slowly with respect to the synchronization period, given that the setup of the system is static.

The filter should suppress fast changes stemming from timestamping jitter, but follow rather slow frequency variations of the

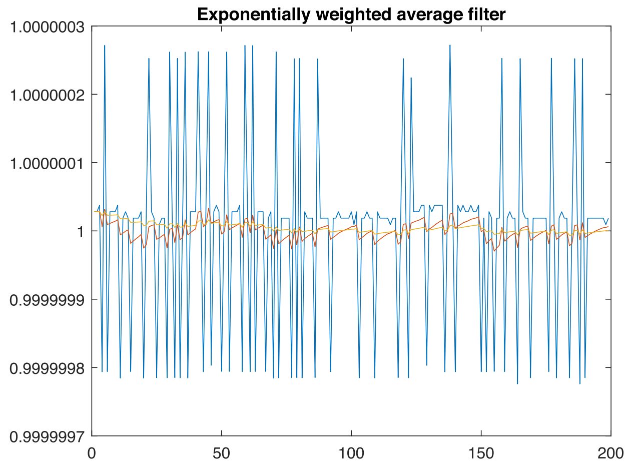

Fig. 8. Typical time series input data (blue) to the rate filter together with the output of an exponential weighted average filter for different widths (yellow: $M = 40$, orange: $M = 10$).

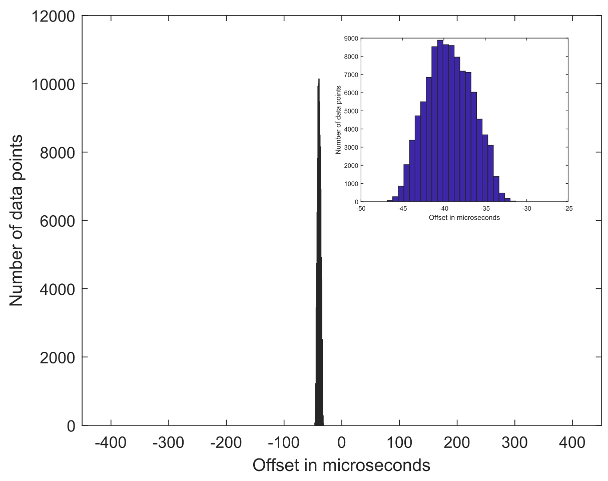

Fig. 9. Histogram of clock offset with rate filtering. The inset shows the close-up of the distribution, the *x*-axis range of the large diagram was not adjusted to allow comparison with the previous results.

local oscillator due to temperature changes and aging. For embedded systems like the ones investigated in this article, resource usage is an additional criterion. Furthermore, the filter selection must pay attention to the particular distribution of the data which will differ from usual random noise. To this end, timestamps were recorded simultaneously by the two devices in the setup like during normal operation. From the timestamps, the relation of the synchronization periods $(t_{\text{Slave,i}} - t_{\text{Slave,i−1}})/(t_{\text{Master,i}} - t_{\text{Master,i−1}})$ is computed. The value 1 corresponds to the case that the two devices run completely synchronously. In this case, the intervals of both the devices have the same duration. Fig. 8 shows typical data, revealing the influence of the scheduler granularity. The peaks in the data come from cycle slip situations as mentioned before.

Three approaches were evaluated for suitability as rate filter. The most intuitive choice would be a classical moving average filter

$$x_i^* = \frac{1}{N}\sum_{n=0}^{N} x_{i-N} \tag{9}$$

with moving window size N. Obviously, a larger window results in better filtering, but has also the drawback that more data need to be stored and that the filter reacts slower.

From the nonlinear filter category, a moving median filter would be an option [28] because it is efficient for outlier removal. However, the quantization effects in Fig. 8 are not really outliers. On the other hand, the filter does not provide averaging for the noisy part of the data. Rather, experiments showed that the median filter tends to produce an offset. Furthermore, it is computationally expensive as it involves storing and sorting the data within the window before calculating the median, and no recursive or iterative implementation is possible.

An exponentially weighted average filter offers a resource-saving alternative. It requires only the last filter value and the new data point for the calculation. The new value is weighted with a constant factor

$$c_f = e^{-1/M} \tag{10}$$

that depends on the filter width M. The filtered data points x_i^* can be derived from the raw data x_i recursively as

$$x_i^* = \begin{cases} \frac{1}{i}(x_i + x_{i-1}^* \cdot (i-1)), & i \le M \\ x_i \cdot (c_f - 1) + x_{i-1}^* \cdot c_f, & i > M. \end{cases} \tag{11}$$

This filter combines simple implementation and the property that recent data points are weighted more. This allows the filter to react to changes in the rate and at the same time to smooth out the influence of the jitter. Fig. 8 shows the filter output for two different widths. For the further experiments, the width of this filter was set to $M = 15$ as a compromise between response time and averaging performance. The filter was applied in addition to the improvements presented in the previous section.

Fig. 9 shows the histogram that now resembles more the normal distribution like behavior to be expected from a well working synchronization. Like already in Fig. 6, we can notice a distinct offset between master and slave clock that apparently cannot be compensated by the synchronization algorithm. This is typical for reference broadcast schemes like the one employed by AUTOSAR and due to the lack of round trip delay measurements. This offset is not only due to the network propagation delay but also due to the mean values of other stochastic delay parameters, in particular to the software delay on the master side between sending the synchronization message and generating the associated timestamp after reading the message from the buffer. As the master runs freely, this delay is more deterministic than on the slave side. The time series (see Fig. 10) demonstrates that the filter effectively reduces the oscillations of the servo algorithm, although the basic timestamping jitter is of course still present.

In addition, we also applied the rate filtering approach to the system with the original settings, i.e., $T_s = 500$ μs, to account for cases where shortening of the scheduler period would be infeasible. Time series and histogram of the clock offsets are similar to Figs. 6 and 10 and, therefore, not explicitly shown, but the aggregated results are depicted in Fig. 11 and included in

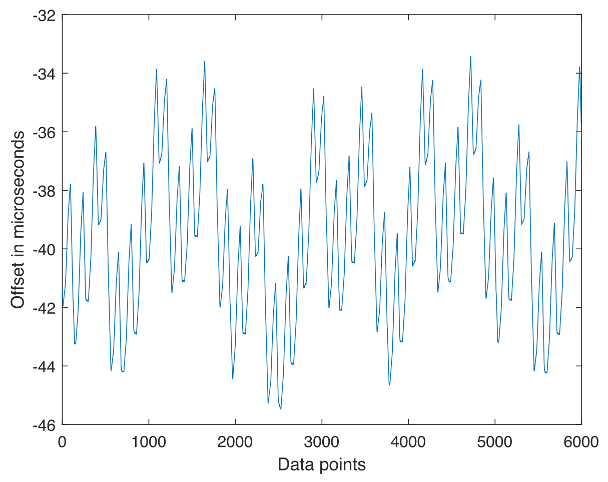

Fig. 10. Clock deviation between master and slave using the rate filtering approach.

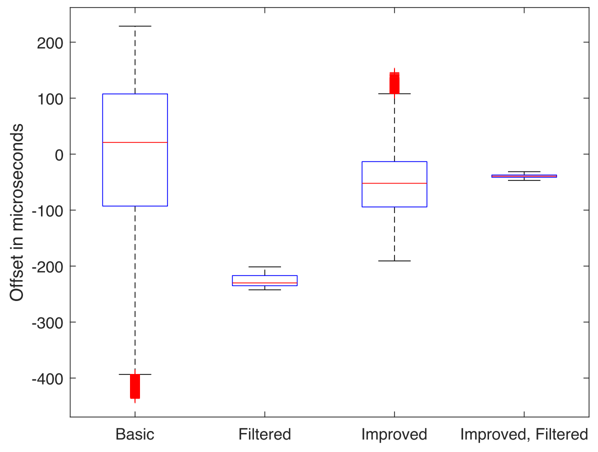

Fig. 11. Comparison of clock offsets between master and slave using different scheduler periods and rate filtering.

TABLE II
COMPARISON OF RESULTS

T_s	Rate filter	Mean	Std. dev.	Precision
500 µs	No	-14 µs	164 µs	∼437 µs
500 µs	Yes	-226 µs	11 µs	∼242 µs
100 µs	No	-69 µs	70 µs	∼191 µs
100 µs	Yes	-33 µs	3 µs	∼47 µs

Table II. It can be seen that ultimately, rate filtering is far more effective than reducing the scheduler period.

IX. Conclusion

The ambition of this article was to investigate clock synchronization over CAN following the AUTOSAR standard in a *realistic* automotive hardware and software environment that is not tweaked specifically for the purpose of this research. The most important findings from our experiments were that proper rate filtering has by far the largest impact on synchronization performance. Shortening the scheduler period primarily reduces a possible residual offset between master and slave clock, and optimizing the task list and task priorities may provide additional improvement, but to a lesser extent.

Table II summarized the results of the previous sections. Competing approaches seemed to be still better, however they either used dedicated hardware or interrupt-based timestamping—solutions that have been explicitly excluded in our work as not fully compatible with the limitations of automotive embedded system development. Most of these works do not consider operating systems, either, which may have an impact on interrupt handling and, thus, on the timestamping latency. Furthermore, performance figures from the literature must be taken with care because it is not always clear how the authors actually define clock precision. If we disregard the residual clock offsets in Fig. 11 and only evaluate the syntonization performance (which is sometimes also used as an indicator for the synchronization precision), we find a standard deviation in the range of ~ 10 μs and below, depending on the scheduler settings. This is comparable with the best performing solutions from the literature.

We conclude that under the given boundary conditions, our results marked probably the optimum that can be realistically obtained by truly software-based clock synchronization according to the AUTOSAR standard. Major improvements could only be achieved by further reducing the timestamping uncertainties. One option could be to improve the rate filtering, e.g., by detecting cycle slips and eliminating them (i.e., the spikes in Fig. 8) from the filter input. The most promising strategy, however, is certainly to use hardware-assisted timestamping. It would solve the problem with nondeterminism caused by interrupts because the timestamps and the associated messages could be retrieved from the buffer at any time through a regular scheduled task. This would also reduce, albeit not completely, the offset between master and slave clocks that cannot vanish in the one-way broadcast scheme used by AUTOSAR. Removing also this residual error would require true round-trip delay measurements that are beyond the scope of the standard.

The future of CAN might also offer interesting potentials. The next generation of CAN will be CAN XL. This version will offer data rates up to 10 Mb/s and a significantly enlarged payload of up to 2048 B, while retaining the media access scheme of CAN and backward compatibility with the previous CAN FD version. This will also permit integration of TCP/IP. As this standard is still under development, it is not yet considered by AUTOSAR, and, therefore, we can only speculate about ways to perform clock synchronization in CAN XL. As the basic media access scheme remains unchanged, the standard AUTOSAR synchronization mechanism [7] should work also with CAN XL. However, integrating TCP/IP also opens the possibility to include PTP as synchronization protocol. This could be particularly interesting with AUTOSAR's time domain concept [5] as it would allow seamless time integration of CAN XL segments into upper-level automotive ethernet networks.

Acknowledgment

The authors would like to thank M. Kobelrausch, TU Wien, for the valuable discussions and complementary information about automotive embedded software development.

References

[1] J. Huang, M. Zhao, Y. Zhou, and C.-C. Xing, “In-vehicle networking: Protocols, challenges, and solutions,” *IEEE Netw.*, vol. 33, no. 1, pp. 92–98, Jan./Feb. 2019.

[2] L. L. Bello, R. Mariani, S. Mubeen, and S. Saponara, “Recent advances and trends in on-board embedded and networked automotive systems,” *IEEE Trans. Ind. Informat.*, vol. 15, no. 2, pp. 1038–1051, Feb. 2019.

[3] R. Bogdan *et al.*, “Optimization of AUTOSAR communication stack in the context of advanced driver assistance systems,” *Sensors*, vol. 21, no. 13, 2021, Art. no. 4561.

[4] S. Raju, G. Jeyakumar, A. Mukherji, and J. K. Thanki, “Time synchronized diagnostic event data recording based on AUTOSAR,” in *Proc. IEEE Int. Conf. Adv. Netw. Telecommun. Syst.*, 2017, pp. 1–6.

[5] *Specification of Synchronized Time-Base Manager*, AUTOSAR Standard 421 CP Release 4.3.1, Dec. 2017.

[6] H. J. Kim, U. Lee, M. Kim, and S. Lee, “Time-synchronization method for CAN-Ethernet networks with gateways,” *Appl. Sci.*, vol. 10, no. 24, 2020, Art. no. 8873.

[7] *Specification of Time Synchronization Over CAN*, AUTOSAR Standard 674 CP Release 4.3.1, Dec. 2017.

[8] A. Mahmood, R. Exel, and T. Sauter, “Impact of hard-and software timestamping on clock synchronization performance over IEEE 802.11,” in *Proc. 10th IEEE Workshop Factory Commun. Syst.*, 2014, pp. 1–8.

[9] F. Hartwich, “CAN frame time-stamping—Supporting AUTOSAR time base synchronization,” in *Proc. 16th Int. CAN Conf.*, 2017, pp. 04-1–04-5.

[10] M. Gergeleit and H. Streich, “Implementing a distributed high-resolution real-time clock using the CAN-bus,” in *Proc. 1st Int. CAN Conf.*, 1994, pp. 1–6.

[11] J. Allan and D. Lee, “Fault-tolerant clock synchronization with microsecond-precision for CAN networked systems,” in *Proc. 9th Int. CAN Conf.*, 2003, pp. 07-1–07-9.

[12] M. Akpınar, E. G. Schmidt, and K. Werner Schmidt, “Drift correction for the software-based clock synchronization on controller area network,” in *Proc. IEEE Symp. Comput. Commun.*, 2020, pp. 1–6.

[13] J. Mitaroff-Szécsényi, P. Priller, and T. Sauter, “Compensating software timestamping interference from periodic non-interruptable tasks,” in *Proc. 22nd IEEE Int. Conf. Emerg. Technol. Factory Automat.*, Sep. 2017, pp. 1–4.

[14] K. Sandstrom, C. Eriksson, and G. Fohler, “Handling interrupts with static scheduling in an automotive vehicle control system,” in *Proc. 5th Int. Conf. Real-Time Comput. Syst. Appl.*, 1998, pp. 158–165.

[15] C. Menard, A. Goens, M. Lohstroh, and J. Castrillon, “Achieving determinism in adaptive AUTOSAR,” in *Proc. 23rd Conf. Des., Automat. Test Eur.*, 2020, pp. 822–827.

[16] *Road Vehicles - Functional Safety - Part 6: Product Development At the Software Level*, ISO Standard 26262-6:2018, International Organization for Standardization, Geneva, Switzerland, 2018.

[17] B. Blackham, V. Tang, and G. Heiser, “To preempt or not to preempt, that is the question,” in *Proc. Asia-Pacific Workshop Syst.*, 2012, pp. 1–7.

[18] B. Blackham, Y. Shi, and G. Heiser, “Improving interrupt response time in a verifiable protected microkernel,” in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 323–336.

[19] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther, “Time triggered communication on CAN (time triggered CAN - TTCAN),” in *Proc. 7th Int. CAN Conf.*, 2000, pp. 1–7.

[20] F. C. Carvalho and C. E. Pereira, “A runtime stability analysis of clock synchronization precision on a time-triggered bus prototype,” *SBA: Controle Automação Sociedade Brasileira de Automatica*, vol. 20, pp. 45–52, Mar. 2009.

[21] G. Breaban, S. Stuijk, and K. Goossens, “Time synchronization for an asynchronous embedded CAN network on a multi-processor system on chip,” in *Proc. IEEE Int. Symp. Precis. Clock Synchronization Meas., Control, Commun.*, 2017, pp. 1–6.

[22] H. Daigmorte, M. Boyer, and J. Migge, “Reducing CAN latencies by use of weak synchronization between stations,” in *Proc. 16th Int. CAN Conf.*, 2017, pp. 4-12–4-19.

[23] G. Rodriguez-Navas, S. Roca, and J. Proenza, “Orthogonal, fault-tolerant, and high-precision clock synchronization for the controller area network,” *IEEE Trans. Ind. Informat.*, vol. 4, no. 2, pp. 92–101, May 2008.

[24] P. Marti, M. Velasco, C. Lozoya, and J. Fuertes, “Clock synchronization for networked control systems using low-cost microcontrollers” Automat. Control Dep., Tech. Univ. Catalonia, Barcelona, Spain, Tech. Rep. ESAII-RR-08-02, 2008.

[25] M. Akpinar, K. W. Schmidt, and E. G. Schmidt, “Improved clock synchronization algorithms for the controller area network (CAN),” in *Proc. 28th Int. Conf. Comput. Commun. Netw.*, 2019, pp. 1–8.

[26] F. Luckinger and T. Sauter, “AUTOSAR-compliant clock synchronization over CAN using software timestamping,” in *Proc. 17th IEEE Int. Conf. Factory Commun. Syst.*, 2021, pp. 49–52.

[27] Bosch, “M CAN Controller Area Network Users Manual,” Gerlingen, Germany, Tech. Rep. Revision 3.2.1, 2015.

[28] L. Tan and J. Jiang, “Digital signals and systems,” in *Digital Signal Processing*, 3rd ed., L. Tan and J. Jiang, Eds. New York, NY, USA: Academic, 2019, ch. 3, pp. 59–89.

Florian Luckinger received the master’s degree in electrical engineering from TU Wien, Vienna, Austria, in 2021.

Since 2015, he has been with Elektrobit, Vienna, Austria, where his main interest is the development of automotive embedded systems. His research interest also includes the development of clock synchronization solutions for the automotive environment.

Thilo Sauter (Fellow, IEEE) received the Ph.D. degree in electrical engineering from TU Wien, Vienna, Austria, in 1999.

He was the Founding Director of the Department for Integrated Sensor Systems, University for Continuing Education Krems, Wiener Neustadt, Austria, and is currently a Professor of Automation Technology with TU Wien. He has authored or coauthored more than 350 scientific publications. His expertise and research interests include embedded systems and integrated circuit design, smart sensors, and automation and sensor networks with a focus on real-time, security, interconnection, and integration issues relevant to cyber-physical systems and the Internet of Things in various application domains such as industrial and building automation, smart manufacturing, or smart grids.

Dr. Sauter is a Senior Administrative Committee Member of the IEEE Industrial Electronics Society. He has held leading positions in renowned IEEE conferences. He has also been involved in the standardization of industrial communications for more than 25 years.