Handling Time and Reactivity for Synchronization and Clock Drift Calculation in Wireless Sensor/Actuator Networks

Marcel Baunach
Graz University of Technology, Institute for Technical Informatics, Graz, Austria

Keywords: Event Timestamping, Reaction Scheduling, Time Synchronization, Clock Drift Calculation, Information Tagging.

Abstract: The precise temporal attribution of environmental events and measurements as well as the precise scheduling and execution of corresponding reactions is of utmost importance for networked sensor/actuator systems. Apart, achieving a well synchronized cooperation and interaction of these wirelessly communicating distributed systems is yet another challenge. This paper summarizes various related problems which mainly result from the discretization of time in digital systems. As an improvement, we'll present a novel technique for the automatic creation of highly precise event timestamps, as well as for the scheduling of related (re-)actions and processes. Integrated into an operating system kernel at the lowest possible software level, we achieve a symmetric error interval around an average temporal error close to 0 for both the timestamps and the scheduled reaction times. Based on this symmetry, we'll also introduce a dynamic self-calibration technique to achieve the temporally exact execution of the corresponding actions. An application example will show that our approach allows to determine the clock drift between two (or more) independently running embedded systems without exchanging any explicit information, except for the mutual triggering of periodic interrupts.

1 INTRODUCTION

Wireless sensor/actuator networks (WSAN) are commonly deployed to observe and interact with their environment. In this respect, temporal and spatial information are the two most fundamental measures for the "attribution" or "tagging" of states and events (i.e. state transitions) within any observed environment. In this context, the states describe a set of physical and logical conditions at a given position and at a certain time, and they are specified by one or more continuous or discrete state values.

Recorded over a certain period of time, variations in the state values allow the detection and analysis of events and event patterns within the environment (Wittenburg et al., 2010) (Römer, 2008). These variations do not only indicate the events' spatial extend, propagation speed, and influence on the environment, but most commonly they also allow the prediction of future states for both the observing system and its surrounding. In this regard, the interaction with the environment, which we already proclaimed to be the most central objective in sensor/actuator systems, typically requires the precise knowledge of time and space to be associated with a node's self-captured and externally obtained values in order to properly correlate the contained information, and to trigger adequate reactions. In fact, measured or otherwise obtained environmental information is often useless unless it is associated with temporal and spatial information.

This paper starts with a discussion of various problems regarding time in digital systems. Next, we present a novel approach for taking timestamps, for measuring and specifying temporal delays, and for scheduling and ensuring reaction times with a symmetric temporal error around 0. Finally, a real-world test bed shows how periodically communicating embedded systems can determine their relative clock drifts without any additional information exchange.

2 TIME IN DIGITAL SYSTEMS

In contrast to specific position vectors and state values of e.g. sensor nodes (which can change sporadically, arbitrarily and independently from each other), time is a common property. It is system independent, and advances continuously with a globally constant rate of change. If the sensor nodes manage to establish a network-wide and consistent notion of time, this in-

Baunach M..
Handling Time and Reactivity for Synchronization and Clock Drift Calculation in Wireless Sensor/Actuator Networks.
DOI: 10.5220/0004680400630072
In *Proceedings of the 3rd International Conference on Sensor Networks* (SENSORNETS-2014), pages 63-72
ISBN: 978-989-758-001-7

formation provides a natural base for their joint interaction and with the environment. Since processors in synchronous digital systems like sensor nodes are always driven by a clock generator C with frequency f_C and period $\lambda_C = \frac{1}{f_C}$, time and time intervals can easily and individually be measured – at least in theory: If it is possible to count the number of elapsed clock periods since a well defined point in time, e.g. the system start, each captured event e, e.g. indicated by an interrupt, can be attributed with its current counter value c_e. Consequently, the event's absolute local system time $\tilde{t}_e$ can easily be recovered by

$$\tilde{t}_e := c_e \cdot \lambda_C, \tag{1}$$

and the time difference (i.e. the delay) $\tilde{\Delta}_{e_1,e_2}$ between two events e_1, e_2 computes as

$$\tilde{\Delta}_{e_1,e_2} := \tilde{t}_{e_2} - \tilde{t}_{e_1} = (c_{e_2} - c_{e_1}) \cdot \lambda_C. \tag{2}$$

Obviously, both the time $\tilde{t}_e$ and the delay $\tilde{\Delta}_{e_1,e_2}$ already involve a concept-inherent imprecision caused by the discretized counter values $c_e \in \mathbb{N}$. In addition, we silently assumed for Eq. (1) and Eq. (2) that λ_C is perfectly known and constant. Neither is true under real-world conditions, and leads to well-known problems, we'll address and counteract in this paper.

Furthermore, as often requested for interactive systems, a reaction r can be scheduled for a captured event e. Its intended execution time $t'_r \in \mathbb{R}$ is commonly related to any event timestamp $t'_e \in \mathbb{R}$ by the specification of a corresponding delay $\Delta'_{e,r} \in \mathbb{R}$:

$$t'_r = t'_e + \Delta'_{e,r} \tag{3}$$

However, in the best case, i.e. if the (operating system) scheduler permits the timely switch to the responding task's context, the reaction will be triggered upon reaching the corresponding counter value $c_r \in \mathbb{N}$ and the corresponding system time $\tilde{t}_r$. In any case, the finally observable reaction delay depends on the resolution λ_C of the system timer:

$$c_r = c_e + \left\lfloor \frac{\Delta'_{e,r}}{\lambda_C} \right\rfloor \qquad \tilde{t}_r = \tilde{t}_e + \left\lfloor \frac{\Delta'_{e,r}}{\lambda_C} \right\rfloor \cdot \lambda_C \tag{4}$$

Although the inherent rounding imprecision is quite intuitive and introduces various hidden implementation problems in real systems, it is commonly simply ignored. Moreover, for concurrent task systems with dynamic execution flows, there is an additional error in $\tilde{t}_r$ which is neither constant nor predictable. Since most embedded operating systems silently accept even this problem, application developers are urged to compensate the imprecision with little control at the task level. SensorOS (Kuorilehto et al., 2007) at least tries to execute reactions in time by scheduling the responsible task earlier. Yet, the applied "delta time" is constant and won't adapt to changing system loads as well.

In the following we'll indicate and discuss the causes and effects of the mentioned problems in detail, and present an approach at the kernel level to reliably compensate the related imprecision in the average case.

Problem P1: Discretization of time. The difference between the true global time t' and the individual system time $\tilde{t}$ has already become visible in Eq. (1) and Eq. (4). While the first advances continuously, the use of a digital counter leads to a discretization of the latter, and imposes a resolution which directly depends on the counter's clock frequency f_C. This may lead to serious systematic errors for the time measurement and the subsequent scheduling of reactions:

The simple *capturing of a timestamp t* for an event – the so called *timestamping* – is immediately affected by some inevitable rounding, and suffers from a measurement error $E_t \in I_1$ with $|I_1| = \lambda_C$. For the naïve and adverse reading of the timer counter, rounding down results in $I_1 := [0, \lambda_C)$, and induces a symmetry around the average measurement error $E_{t,\text{av}} = \frac{1}{2}\lambda_C$. Depending on the use of such timestamps, the emerging errors might accumulate during the system runtime. Similarly, the explicit *specification of delays* Δ'_t in software is also subject to rounding errors E_Δ. However, we can round half up (e.g. according to DIN 1333) manually when selecting a delay, and thus the corresponding error is $E_\Delta \in I_3 := [-\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C)$. Though not avoidable entirely, I_3 is at least symmetric around 0, and the average error is 0.

Based on these two fundamental error intervals I_1 and I_3, other intervals can be derived, and consequently exhibit an imprecision, too: For the *measurement of delays* Δ_E, we see the implicit compensation of the asymmetry in I_1: $E_\Delta \in I_2 := I_1 - I_1 = (-\lambda_C, +\lambda_C)$. In contrast, the *scheduling of reaction times t* on external events inherits the asymmetry in I_1: $E_t \in I_4 := I_1 + I_3 = [-\frac{1}{2}\lambda_C, +\frac{3}{2}\lambda_C)$. System reactions will consequently suffer from an average systematic lateness of $\frac{1}{2}\lambda_C$.

Table 1 summarizes the error types and their corresponding intervals which must be expected for the naïve capturing of timestamps by simply reading the timer register after the corresponding event occurrence (e.g. within an IRQ handler). The resulting effects, and our proposed solution to compensate this asymmetry, will be discussed later.

Problem P2: Capturing of timestamps. The creation of reactive systems demands for the precise assignment of timestamps for internal and external

Table 1: Error intervals for different discretization techniques (system time resolution: λ_C).

		Naïve discretization		Our discretization approach	
problem / error type	derived from	error interval	symmetry	error interval	symmetry
P1: capturing of timestamps	fundamental	$I_1 = [0, \lambda_C)$	$\not\downarrow \frac{1}{2}\lambda_C \not\downarrow$	$I_3 = [-\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C)$	0
P2: measurement of delays	$I_1 - I_1$	$I_2 = (-\lambda_C, +\lambda_C)$	0	$I_2 = (-\lambda_C, +\lambda_C)$	0
P3: specification of delays	fundamental	$I_3 = [-\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C)$	0	$I_3 = [-\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C)$	0
P4: scheduling of reaction times	$I_1 + I_3$	$I_4 = [-\frac{1}{2}\lambda_C, +\frac{3}{2}\lambda_C)$	$\not\downarrow \frac{1}{2}\lambda_C \not\downarrow$	$I_4 = [-\lambda_C, +\lambda_C)$	0

events. Reaching a voltage threshold at an analog-digital-converter (ADC) or detecting a signal edge at an I/O pin are just two simple examples. However, most observable changes within the environment have one thing in common: They are indicated to the CPU at runtime by so called *interrupt requests* (IRQs), and should be handled as soon and as fast as possible by the corresponding *interrupt service routines* (ISRs). Since ISRs are commonly higher privileged than regular application code, they will preempt the latter for their own execution. Thus, they seem to be perfectly suitable for capturing the timestamp for any emerging event. However, even the first instruction within each ISR is not executed before some additional delay, which is also known as *interrupt latency* Δ_{IRQ}: If the timer value c_{TS} for the timestamp itself is copied after another implementation-specific delay Δ_{ISR} within the ISR, then we can compute the discrete timestamp $\tilde{t}_e$ for the captured event e as follows:

$$\tilde{t}_e = c_{\text{TS}} \cdot \lambda_C - (\Delta_{\text{IRQ}} + \Delta_{\text{ISR}}) = \tilde{t}_{\text{TS}} - \Delta_{\text{TS}} \qquad (5)$$

Hence, a prerequisite for reliable time tracking via Eq. (5) is, that the correction value Δ_{TS} is constant and free from rounding errors with respect to the discrete system time period. As we will see, both can be achieved through careful code preparation.

Problem P3: Simultaneity and scheduling reliability. Although the perfectly simultaneous transition of two states can never occur in real systems[1], the surjective discretization of time can easily lead to the assignment of exactly the same system time for multiple events or scheduled actions. Since resource conflicts often prevent the truly parallel processing of events as well as the simultaneous execution of (re)actions, they usually lead to an implicit serialization. The order depends on the task scheduler and the internal task priorities. Since there is most commonly just a single IRQ controller, this is already true for the generation of timestamps. In fact, the maximum degree of parallelism is always limited by the number of available functional units[2]. A reliable scheduling (e.g. for complying to hard real-time demands) must be achieved through either static techniques at development time or dynamic methods at runtime. A corresponding technique for dynamic resource management under real-time conditions is presented in (Baunach, 2012).

Problem P4: Imprecision in the timer frequency. Time measurement in digital systems is usually accomplished by using a pulse generator with a specified frequency f_0. Internally, this component uses an oscillator (most commonly a quartz crystal) to generate a periodic clock signal. The characteristics and stability of such oscillators depend significantly on their manufacturing parameters, age, and various environmental conditions like e.g. voltage variations (Hewlett Packard, 1997): A varying frequency drift Δf must always be expected. Its relative error $\frac{\Delta f}{f_0}$ is commonly expressed in units of parts per million (ppm). For simple low-cost quartzes, and within the typical temperature ranges of WSAN applications, the temperature sensitivity of a typical HC49 quartz can already result in deviations of $\pm$ 20 ppm.

Variations in the clock precision are especially critical in distributed applications. Since time measurement is initially individual for each involved system and therefore can drift apart, this may quickly generate inconsistent data, and must be compensated by adequate and repeated synchronization measures.

Problem P5: Global time base and synchronization with other systems. When does time measurement actually start, i.e. when is or was time $t_0 = 0$? If we consider a completely independent system which uses the notion of time only for its internal operation,

[1]The resolution of the time measurement must simply be chosen fine enough!

[2]Functional units refer to e.g. processors and their cores, or to autonomously operating peripheral components.

e.g. to capture events and to schedule actions by a partial order[3], the use of a pure local time with arbitrary begin is absolutely sufficient – e.g. time $t_0 = 0$ may simply indicate the system start. However, as soon as time is of global relevance, e.g. if actions have to take place synchronized on different systems, a common time base is often indispensable. This immediately raises the question about which time or which system is used as reference. In any case, its provider should be highly available and exhibit a high clock stability and precision. Several methods exist for the actual synchronization: Some are based on (regular) time checks or on the measurement of the pairwise drift between the involved systems. Others rely on dedicated reference systems and allow the synchronization based on centrally triggered events like e.g. radio broadcasts (cf. GPS and the DCF77 protocol). Finally, distributed methods are available for multihop systems to successively achieve a common time base, e.g. via Desynchronization (Mühlberger, 2013).

3 AN ADVANCED TIME DISCRETIZATION APPROACH

Considering the aforementioned problems, which originated from the integration of time-awareness into digital systems, P1-P3 directly affect the environmental interaction and can be addressed by each system individually. In contrast, P4 and P5 require some information exchange with other systems. These "peers", however, are not necessarily available during the entire system runtime. For this reason, P1-P3 are treated locally at the embedded systems level (e.g. in the operating system kernel), while P4 and P5 must be addressed more globally (e.g. in the network layer or at application level).

At the embedded systems level, our approach relies on a hardware timer component to provide a local system timeline with a fixed temporal resolution. The timeline management is integrated directly into the OS kernel and accessible for all software layers through the OS API. This unifies the usage by application tasks and avoids execution time imponderabilities through unpredictable code interleaving at runtime. Based on our approach, the kernel automatically captures a timestamp $\tilde{t}_e$ for each interrupt e, and compensates the error's asymmetry about $\frac{1}{2}\lambda_C$ which would result from using the naïve approach with $I_1 = [0, \lambda_C)$ as explained in Section 2. Therefore, the kernel as a hardware abstraction layer provides standardized and architecture dependent interrupt service routines for introducing a constant and carefully dimensioned delay $\Delta_{TS} = \Delta_{IRQ} + \Delta_{ISR}$ before actually capturing the timer's counter value after the IRQ occurrence. According to Eq. (5) we then have to reduce the captured counter value by an adequate correction value Δ_{corr}: Selected properly, this correction finally results in the symmetry about 0 for $I_1 := [-\frac{1}{2}\lambda, \frac{1}{2}\lambda)$. While the timestamp measurement error E_{t_e} will still be equally distributed over I_1, this interval is shifted, and the average timestamp error is reduced from initially $\frac{1}{2}\lambda$ down to 0. At the same time, the propagation and amplification of systematic errors for time-dependent reactions will also be kept low and symmetric about 0, i.e. $I_4 = I_1 + I_3 = [-\lambda_C, +\lambda_C)$. Table 1 compares the error intervals of our compensation approach with the naïve technique.

How can this symmetry be guaranteed? In order to deal with the related problems P1 and P2, we propose a concept based on two synchronized clocks with interdependent frequency. Thereby, we assume the CPU frequency to be higher than the timer frequency, while conversely, the system time is derived from the quartz-stabilized CPU clock by an even integer divider. Commonly, both requests do not impose an unreasonable restriction on the hardware/software design: In fact, they are already satisfied in many systems, since usually only a single central oscillator is used as base for all other system clocks. While the CPU is commonly directly driven by this main clock, other components apply power-of-two dividers to derive their individual frequencies. Finally, and for computationally constrained embedded systems in particular, driving a local time with the maximum resolution would cause unnecessary CPU load[4].

Besides the following formal description of our approach, we also refer to the example in Figure 1 for a comprehensive understanding. Initially, we denote the CPU clock frequency as f and its period as λ. The system time frequency is denoted as f_C and its period as λ_C. In addition, we demand for

$$f_C := \frac{f}{\alpha} \quad \text{and} \quad \lambda_C := \alpha \cdot \lambda \quad \text{with} \quad \alpha \in \mathbb{N}^{\geq 2}, \alpha \text{ even}. \tag{6}$$

If an interrupt e occurs at time t'_e, the corresponding timer counter c_e will not be copied before some system inherent delay Δ_{TS} has passed. For our approach, we request this delay to take exactly Δ_c CPU cycles as follows:

[3]"Partial", since the discretization of time may lead to simultaneity ($\rightarrow$ P3).

[4]The system time must be accumulated in software at every timer overflow. Especially for timers with small word widths, this can quickly lead to a huge performance penalty. For instance, a 16 Bit counter counting at f_C = 1 MHz will already overflow after every 65.536 ms.

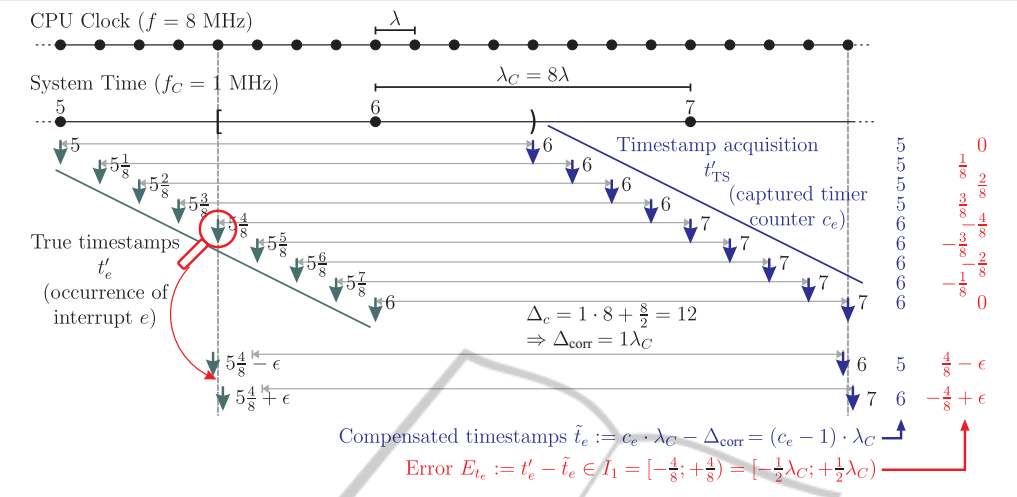

Figure 1: IRQ timestamp acquisition with our approach for various potential event occurrence times t'_e.

$$\Delta_c := n \cdot \alpha + \frac{\alpha}{2} \quad \text{with} \quad n \in \mathbb{N}_0 \tag{7}$$

Thus, the delayed acquisition of the timestamp takes place at time

$$t'_{TS} = t'_e + \Delta_{TS} = t'_e + \Delta_c \cdot \frac{1}{f} = t'_e + \left(n \cdot \alpha + \frac{\alpha}{2}\right) \cdot \frac{1}{f}. \tag{8}$$

To compensate for this delay, and to force the timestamp error interval I_1 to become symmetric around the true event occurrence time while also exhibiting an average error close to 0, we can select the correction value as an integer multiple of λ_C:

$$\Delta_{corr} := (n \cdot \alpha) \cdot \frac{1}{f} = n \cdot \lambda_C \tag{9}$$

Thus, we simply have to subtract n from the copied timer value c_e to compute the timestamp $\tilde{t}_e$ for the interrupt e:

$$\tilde{t}_e = \left\lfloor \frac{t'_{TS}}{\lambda_C} \right\rfloor \cdot \lambda_C - \Delta_{corr} = c_e \cdot \lambda_C - n \cdot \lambda_C = (c_e - n) \cdot \lambda_C \tag{10}$$

Since c_e (timer driven) and n (constant) are integers of architecture word width, their subtraction is easily accomplished. Besides, the result's resolution implicitly equals the resolution of the system time. However, we still have to prove the symmetry about 0 for the error intervals in Table 1:

Lemma 1. *For our discretization approach the error intervals I_1, I_2, I_3, and I_4 for taking timestamps, for measuring and specifying delays, as well as for computing reaction times are symmetric about 0.*

Proof. While I_3 is not affected by our novel approach, the demanded symmetry of I_1, I_2, and I_4 can easily be proofed by some interval arithmetic. With Eq. (8), (10) the expected error E_{t_e} of the timestamp $\tilde{t}_e$ computes as

$$\begin{aligned} E_{t_e} &= t'_e - \tilde{t}_e \\ &= \left[t'_{TS} - \left(n \cdot \alpha + \frac{1}{2} \cdot \alpha\right) \cdot \frac{1}{f}\right] - \left[\left\lfloor \frac{t'_{TS}}{\lambda_C} \right\rfloor \cdot \lambda_C - n \cdot \lambda_C\right] \\ &= \underbrace{t'_{TS} - \left\lfloor \frac{t'_{TS}}{\lambda_C} \right\rfloor \cdot \lambda_C}_{\in [0, \lambda_C)} - \frac{1}{2} \cdot \lambda_C \quad \in \quad \left[-\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C\right) \end{aligned}$$

$$\begin{aligned} \Rightarrow I_1 &= \left[-\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C\right) \\ \Rightarrow I_2 &= I_1 - I_1 = \left[-\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C\right) - \left[-\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C\right) \\ &= (-\lambda_C, +\lambda_C) \\ \Rightarrow I_4 &= I_1 + I_3 = \left[-\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C\right) + \left[-\frac{1}{2}\lambda_C, +\frac{1}{2}\lambda_C\right) \\ &= [-\lambda_C, +\lambda_C) \end{aligned}$$

□

3.1 An Implementation Example

As an example, we'll take a look at the integration of our novel approach into the reference implementation of *Smart*OS (Baunach et al., 2007) for the MSP430F1611 (Texas Instruments Inc., 2006) MCU and the SNoW^5 sensor nodes (→ Figure 1) . While the main clock drives the CPU at f = 8 MHz, the divider $\alpha = 8$ derives the frequency $f_C = 1$ MHz for the system time with a resolution of 1 µs. According to Eq. (7), an adequate delay Δ_c between each interrupt occurrence and the acquisition of its timestamp can be adjusted through n:

$$\Delta_c := n \cdot \alpha + \frac{\alpha}{2} = n \cdot 8 + 4 \quad \text{with} \quad n \in \mathbb{N}_0 \tag{11}$$

```
1 __irq_e:
2   ; hardware IRQ latency    ; +6 CPU cycles from t'_e              Δ_IRQ = 6λ
3   nop                       ; +1 CPU cycle                      \
4   nop                       ; +1 CPU cycle                       > Δ_ISR = 6λ
5   mov &TIMER_COUNTER, &TS   ; +4 CPU cycles for latch in        /
6                             ; Total delay of captured timestamp: Δ_TS = Δ_IRQ + Δ_ISR = 12λ = 1.5μs
7
8   mov #e, &__irq_number     ; save IRQ number for subsequent processing in the associated event handler
9   jmp __kernel_entry        ; jump to kernel code
```

Listing 1: Timestamping within a specially prepared kernel ISR for IRQ number e.

Listing 1 shows the standardized kernel ISR for any interrupt with number e: Since the CPU inherently delays the acceptance of an interrupt by $\Delta_{\text{IRQ}} = 6$ CPU cycles, we already have to select $n \geq 1$. In fact, we did select $n = 1$ and thus have to wait for an additional number of $\Delta_{\text{ISR}} := \Delta_c - \Delta_{\text{IRQ}} = 6$ CPU cycles within the ISR ($1 \cdot 8 + 4 = 6 + 6$). According to the specification of the `mov` instruction, which is used for saving the timer value `TS` in Line 5, it takes 4 CPU cycles until the value is read from the special function register `TIMER_COUNTER`. The remaining two cycles are filled up by `nop` instructions. After the acquisition of the counter value, the specific IRQ number is saved and the kernel mode is entered for the actual event handling.

To save CPU time the ISR will only save the current 16 Bit timer value which indicates the delay since the last timeline update. The computation of the final absolute timestamp $\tilde{t}_e$ is initially avoided, and delayed until the application's event handler requests this information from the OS. Then, according to Eq. (9), n is simply subtracted from the event's absolute counter value c_e, which in turn is the sum of the timeline and the just captured timer value `TS`. With Eq. (10) the result can directly be interpreted as absolute system time given in the timeline resolution of 1 μs:

$$\tilde{t}_e = (c_e - n) \cdot \lambda_C = (\texttt{timeline} + \texttt{TS} - n) \cdot \lambda_C \quad (12)$$

Note that the applied computation is correct, as long as the IRQ handlers are always executed in kernel mode where further interrupts are disabled and neither the `timeline` nor `TS` will change concurrently.

4 EVALUATION AND APPLICATION EXAMPLE

The test bed for demonstrating the benefit of our timestamping approach consists of pairs of nodes A, B playing some sort of Ping Pong game as depicted in Figure 2: By a wired or wireless remote connection, one node, WLOG B, triggers an IRQ signal e_0 which is received and timestamped ($\tilde{t}_0$) by the other node A through the just presented timestamping approach. After some fixed delay Δ_{delay} the signal will be returned by node A, and in turn the other node B catches, timestamps, and returns the signal after the same delay Δ_{delay}. Having received the last trigger e_n with local timestamp $\tilde{t}_n$ in a *perfect system*, the observed delay $\tilde{\Delta}_{\text{total},n}$ between each node's captured first and last signal timestamp should obviously equal the mathematically expected delay $\Delta'_{\text{total},n}$:

$$\tilde{\Delta}_{\text{total},n} := \tilde{t}_n - \tilde{t}_0 \stackrel{!}{=} 2n \cdot \Delta_{\text{delay}} =: \Delta'_{\text{total},n} \quad (13)$$

However, this equality will commonly not be observable in *real systems*. In fact each involved device will suffer from its own and the other device's imprecision:

First, the nodes apply independent clocks, drift apart, and, though globally fixed, they will finally *not* defer their responses by exactly the same delay Δ_{delay}. Though our nodes' CPUs are driven by quartzes from the same lot, the clock drifts vary depending on the selected node pair and the environmental influences described before. In fact, Figure 3 shows significantly different drifts $d_{A,B}(t)$ for three node pairs measured over some time t.[5]

Second, the responses must be scheduled and initiated by the responsible task on each node. Therefore, these tasks compute their next intended local response time t_r^* from each previously captured signal timestamp, and then sleep to release the CPU for other tasks. However, waking up sufficiently early to emit the signal in time is not that easy since some load-dependent and variable system overhead must always be taken into account.

Third, the base for each delay computation is never perfect since each captured timestamp $\tilde{t}_c$ exhibits an inherent error $E_{\tilde{t}_c} \in I_1$. While this cannot be

[5]The nodes with IDs 10, 11, and 72 were arbitrarily selected from our pool. The drift was measured via an oscilloscope tracking the delay between two periodically triggered I/O pins at both nodes of each pair. As a plausibility test, the measured drift of each pair corresponds perfectly to the other pairs' drifts: $918\,\frac{\mu s}{100s} + 1900\,\frac{\mu s}{100s} = 2818\,\frac{\mu s}{100s}$.

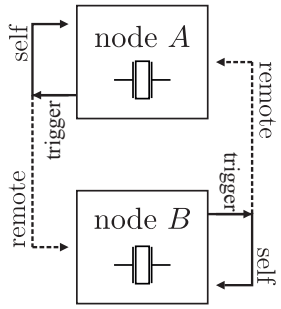

Figure 2: The Ping Pong test bed setup (remote connections can be wired or wireless).

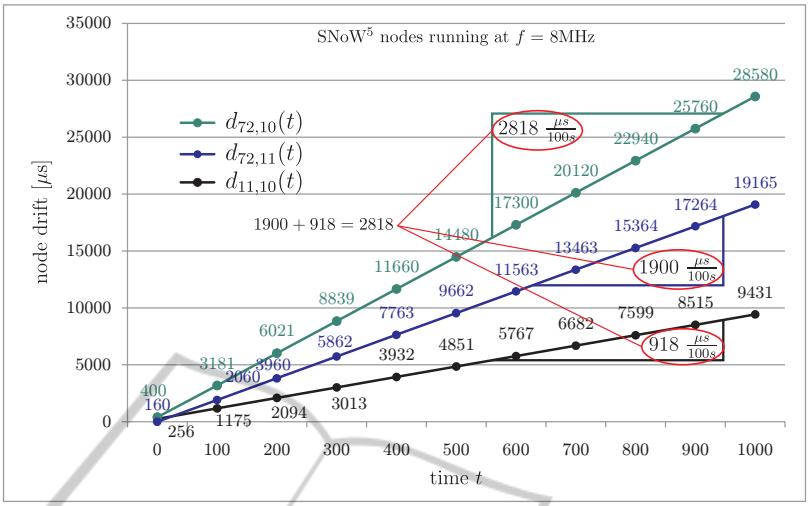

Figure 3: Clock drift for three node pairs.

avoided entirely as discussed before, the error should at least be about 0 µs in the average case according to Lemma 1.

4.1 Signal TX and Self-Calibration

For the precisely timed signal emission, we propose a dynamic self-calibration scheme based on self-observation. Therefore, the trigger signal will not only be captured by the other node where it is tagged with the timestamp $\tilde{t}_c$, but also by the emitting node itself. We denote the corresponding local timestamp as $\tilde{t}_r$. If the intended local response time for the current iteration has been computed as t_r^*, the lateness can be computed afterwards and used as compensation value Δ_{comp} to adjust the delay for the next iteration at its emission time t_r^*:

$$\Delta_{\text{comp}} := \tilde{t}_r - t_r^* \quad (14)$$

$$t_r^* := \tilde{t}_c + \Delta_{\text{delay}} - \Delta_{\text{comp}} \quad (15)$$

In fact, the response time precision error ($E_{t_r^*} \in I_4$) depends not only on the two timestamps and their particular precision error ($E_{\tilde{t}_r}, E_{\tilde{t}_c} \in I_1$), but also on the error in the measured delay ($E_{\Delta_{\text{comp}}} \in I_2$) and the hard coded delay for the reply ($E_{\Delta_{\text{delay}}} \in I_3$) itself. Since we intentionally selected $\Delta_{\text{delay}} := m \cdot \lambda_C$ with $m \in \mathbb{N}$, at least this value is free from rounding errors and $I_3 := [0;0)$ for this special application.

4.2 Pairwise Drift Calculation

For our tests we set up various node pairs A and B as depicted in Figure 2, and we were interested in each nodes' $x \in \{A,B\}$ local timing error e_x which was autonomously calculated by each node after n iterations:

$$e_x \overset{\text{Eq. (13)}}{:=} \tilde{\Delta}_{\text{total},n} - \Delta'_{\text{total},n} = (\tilde{t}_n - \tilde{t}_0) - 2n \cdot \Delta_{\text{delay}} \quad (16)$$

Obviously, both timing errors e_A, e_B have different sign unless the clocks are perfectly synchronous (then $e_A = e_B = 0$ µs). Additionally, we define the symmetry error e_{symm} as seen by an external observer as the average value over e_A, e_B. Since the average timestamp error $E_{t,\text{av}} \in I_1$ will accumulate over the two acquired trigger timestamps within each iteration, we expect

$$e_{\text{symm}} := \frac{e_A + e_B}{2} = 2n \cdot E_{t,\text{av}}. \quad (17)$$

If we *indeed* achieved the timestamp error interval I_1 to be symmetric about 0, i.e. by selecting $\Delta_c = n \cdot \alpha + \frac{1}{2} \cdot \alpha$ properly according to Eq. (7), we can expect two observations for any pair of nodes A, B:

1. If both values e_A and e_B are made available to an external observer, their measured clock drift $d_{A,B}$, as depicted in Figure 3, can be verified through

$$d'_{A,B} := e_A - e_B \overset{!}{=} d_{A,B} \quad \text{with} \quad d_{A,B} = -d_{B,A}. \quad (18)$$

2. According to Eq. (17), $e_{\text{symm}} \overset{!}{=} 2n \cdot 0\mu s = 0\mu s$, and thus both values e_A and e_B will show the same absolute values. In direct consequence, each node can autonomously estimate its own drift towards the other node autonomously:

$$\tilde{d}_{A,B} = 2 \cdot e_A \quad \text{(for node } A\text{)} \quad (19)$$

$$\tilde{d}_{B,A} = 2 \cdot e_B \quad \text{(for node } B\text{)} \quad (20)$$

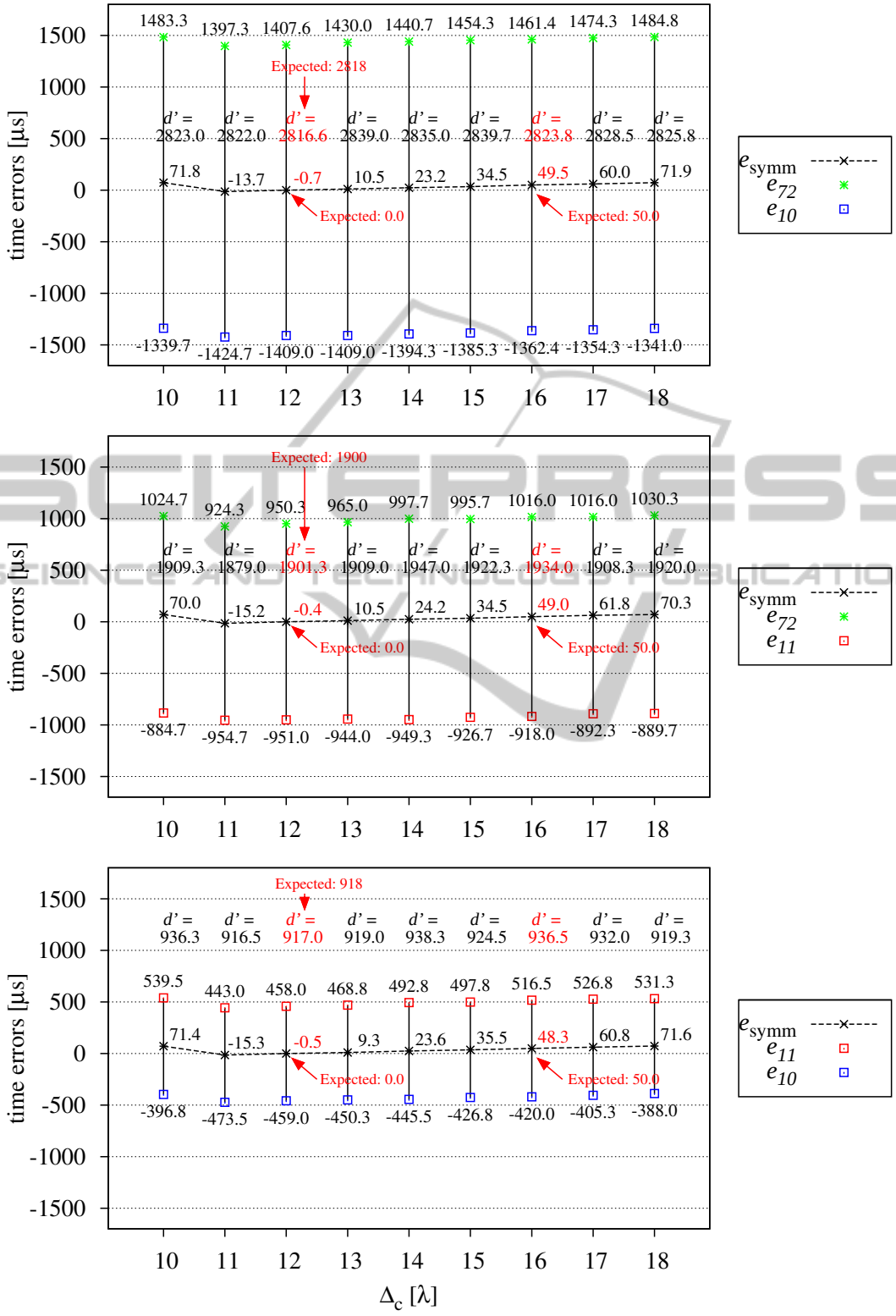

Figure 4: The node timing error for different values of Δ_c after 100 s as autonomously measured by each node (see Figure 3 for the expected values).

Table 2: Drift calculation for $\Delta_c = 12$.

	local information[1,2]			observer[3]
$\tilde{d}_{A,B}$	node 10	node 11	node 72	$d'_{A,B}$
$\tilde{d}_{72,11}$	***1900.0***	1902.0	1900.0	1900
$\tilde{d}_{72,10}$	2818.0	***2818.0***	2816.0	2818
$\tilde{d}_{11,10}$	918.0	916.0	***916.0***	918

[1] regular font: measured according to Eq. (19)
[2] ***bold/italic***: derived according to Eq. (21)
[3] true drift as expected from Figure 3

Table 3: Drift calculation for $\Delta_c = 16$.

	local information[1,2]			observer[3]
$\tilde{d}_{A,B}$	node 10	node 11	node 72	$d'_{A,B}$
$\tilde{d}_{72,11}$	***1884.0***	1836.0	2032.0	1900
$\tilde{d}_{72,10}$	2724.0	***2870.0***	2922.0	2818
$\tilde{d}_{11,10}$	840.0	1034.0	***890.0***	918

[1] regular font: measured according to Eq. (19)
[2] ***bold/italic***: derived according to Eq. (21)
[3] true drift as expected from Figure 3

In particular, the exchange of any additional data, such as timestamps, between the nodes is not necessary to obtain this information (since Δ_{delay} is constant).

The reason becomes clear when considering the involved error intervals over n iterations in Eq. (24): Obviously, all error intervals remain symmetric about 0 µs throughout the entire test. In particular, the average error for each variable is 0 µs, and consequently $e_{\text{symm}} = 0$ µs, too.

In contrast, if we intentionally violate Eq. (7) by using e.g. $\Delta_c := n \cdot \alpha$ instead, the average timestamp error interval would be symmetric around $E_t = \frac{1}{2}\lambda_C$. Consequently, $e_{\text{symm}} = 2n \cdot \frac{1}{2}\lambda_C$, and neither the autonomous drift computation through Eq. (19) nor the external drift verification through Eq. (18) would work any more.

4.3 Real-World Test Bed Analysis

Figure 4 shows the test bed results for the three already mentioned node pairs from Figure 3, and for various values of Δ_c after $n = 50$ iterations with $\Delta_{\text{delay}} = 1$ s ($\Delta'_{\text{total},n} = 100$ s). Note that the results repeat in a cyclic manner with period $\alpha = 8$, and thus the values for $\Delta_c = 10$ are similar to those for $\Delta_c = 18$.

When using $\Delta_c = 1 \cdot 8 + \frac{8}{2} = 12$, we did indeed achieve the expected symmetry error $e_{\text{symm}} \approx 0$ µs for all pairs. At least we received $|e_{\text{symm}}| < \lambda_C = 1$ µs, which is the timeline resolution and thus the best timestamp precision a node can reach. Furthermore, for $\Delta_c = 12$, $d'_{A,B} \approx \tilde{d}_{A,B}$ verifies the measured values from Figure 3. Most important, as shown in Table 2, the autonomously measured drifts between two nodes are almost perfect. Indeed, the maximum visible deviation is in range ±2 µs. Another fact which we can verify from this table is, that since WLOG node A knows its drifts $\tilde{d}_{A,B}$ and $\tilde{d}_{A,C}$ towards the two other nodes B and C respectively, it can also reliably derive the drift $\tilde{d}_{B,C}$ via

$$\tilde{d}_{B,C} := \tilde{d}_{A,C} - \tilde{d}_{A,B}. \tag{21}$$

For any other values of Δ_c violating Eq. (7), the nodes can not gain reliable information about their relative drift autonomously. When using e.g. $\Delta_c = 2 \cdot 8 = 16$, Figure 4 shows values close to the expected symmetry error $e_{\text{symm}} = 2n \cdot \frac{1}{2}\lambda_c = 50$ µs (cf. Eq. (17)). As a result, Table 3 summarizes the autonomously measured and computed drifts between the node pairs, and reveals quite large and asymmetric deviations towards the true drifts.

Besides the precision of the autonomous drift estimation, another interesting metric is the resulting trigger frequency. The theoretical value

$$f_{\text{trig}} := \left(2 \cdot \Delta_{\text{delay}}\right)^{-1} \tag{22}$$

will not be visible in reality since neither node uses a perfect clock. However, we would at least like to achieve

$$f_{\text{trig, av.}} = \left(\Delta'_{\text{delay},A} + \Delta'_{\text{delay},B}\right)^{-1}, \tag{23}$$

which is definitely the best compromise two nodes A, B can find if their true drift compared to the perfect global clock is unknown. Again, this is only possible if $e_{\text{symm}} = 0$. When looking at e.g. the graph for the nodes with IDs 11 and 72 in Figure 4, the extrapolation of e_{symm} leads to a symmetry error of -345.6 µs for $\Delta_c = 12$ and 42336 µs for $\Delta_c = 16$ within one complete day. Thus, the larger $|e_{\text{symm}}|$ the larger the deviation from the intended frequency $f_{\text{trig, av.}}$. The effects are once more visible in Tables 2 and 3: For $\Delta_c = 12$ the values in each row are almost equal (i.e. consistent), while they exhibit significant variations for $\Delta_c = 16$.

5 CONCLUSIONS AND OUTLOOK

In this paper we proposed an approach for obtaining precise timestamps $\tilde{t}_e$ for external events e, and for ensuring the precisely timed execution of reactions r at scheduled times $\tilde{t}_r$. The error intervals for both

$$
\begin{array}{lcccccc}
 & t_r^* & \overset{\text{Eq. (15)}}{:=} & \tilde{t}_c & + & \Delta_{\text{delay}} & - & \Delta_{\text{comp}} \\
\text{iteration} & I_4 & & I_1 & & I_3 & & I_2 \\
1: & [-\lambda_C;\lambda_C) & & [-\frac{1}{2}\lambda_C;\frac{1}{2}\lambda_C) & & [0;0) & & (-\frac{1}{2}\lambda_C;\frac{1}{2}\lambda_C) \\
 & \vdots & & \vdots & & \vdots & & \vdots \\
n: & [-n\lambda_C;n\lambda_C) & & [-\frac{1}{2}\lambda_C;\frac{1}{2}\lambda_C) & & [0;0) & & (-(n-\frac{1}{2})\lambda_C;(n-\frac{1}{2})\lambda_C)
\end{array} \tag{24}
$$

$$
\begin{array}{lcccc}
 & \Delta_{\text{comp}} & \overset{\text{Eq. (14)}}{:=} & \tilde{t}_r & - & t_r^* \\
\text{iteration} & I_2 & & I_1 & & I_1 \\
1: & (-\frac{3}{2}\lambda_C;\frac{3}{2}\lambda_C) & & [-\frac{1}{2}\lambda_C;\frac{1}{2}\lambda_C) & & [-\lambda_C;\lambda_C) \\
 & \vdots & & \vdots & & \vdots \\
n: & (-(n+\frac{1}{2})\lambda_C;(n+\frac{1}{2})\lambda_C) & & [-\frac{1}{2}\lambda_C;\frac{1}{2}\lambda_C) & & [-n\lambda_C;n\lambda_C)
\end{array}
$$

$\tilde{t}_e$ and $\tilde{t}_r$ are symmetric about 0. While the first is achieved through the unified and carefully prepared preprocessing of interrupts by the kernel, the latter becomes possible through a simple self-calibration scheme at application layer. Both techniques proved to be a great benefit for an inherent problem within distributed but interacting (embedded) systems: As long as time is not properly manageable locally by the individual nodes, network-wide synchronization and event or state tagging will hardly achieve the potentially feasible precision.

Using our approach, a corresponding test bed verified, that it is possible to determine the drift between two nodes without the explicit exchange of any quantitative information (like e.g. timestamps or previously measured delays). Instead, it is sufficient to periodically pass events (i.e. interrupts) between the nodes. Since suitable periodic behavior can also be found in several (wireless) communication protocols like (Støa and Balasingham, 2011), (Ito et al., 2009), (Mutazono et al., 2009), the proposed techniques can also be applied to support time synchronization and self-organization among the involved systems.

In fact, we already observed good time synchronization results when integrating our approach into the Desync protocol from (Mühlberger, 2013). Another objective for us is to support our timestamping concept in hardware: Within a hardware/software codesign project, a specifically prepared interrupt controller of an experimental CPU architecture is already able to pre-process and store timer values even for simultaneously occurring events.

REFERENCES

Baunach, M. (2012). Towards Collaborative Resource Sharing under Real-Time Conditions in Multitasking and Multicore Environments. In *17th IEEE International Conference on Emerging Technology & Factory Automation (ETFA 2012)*. IEEE Computer Society.

Baunach, M., Kolla, R., and Mühlberger, C. (2007). Introduction to a Small Modular Adept Real-Time Operating System. In *6. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze*. RWTH Aachen University.

Hewlett Packard (1997). *Fundamentals Of Quartz Oscillators*. Electronic Counters Series.

Ito, K., Suzuki, N., Makido, S., and Hayashi, H. (2009). Periodic Broadcast Type Timing Reservation MAC Protocol for Inter-Vehicle Communications. In *28th IEEE conference on Global telecommunications (GLOBECOM 2009)*. IEEE Press.

Kuorilehto, M., Alho, T., Hännikäinen, M., and Hämäläinen, T. D. (2007). SensorOS: A New Operating System for Time Critical WSN Applications. In *7th International Workshop on Embedded Computer Systems (SAMOS 2007)*, LNCS. Springer.

Mühlberger, C. (2013). On the Pitfalls of Desynchronization in Multi-hop Topologies. In *2nd International Conference on Sensor Networks (SENSORNETS 2013)*. SciTePress.

Mutazono, A., Sugano, M., and Murata, M. (2009). Frog Call-Inspired Self-Organizing Anti-Phase Synchronization for Wireless Sensor Networks. In *2nd International Workshop on Nonlinear Dynamics and Synchronization (INDS 2009)*.

Römer, K. (2008). Discovery of Frequent Distributed Event Patterns in Sensor Networks. In *5th European Conference on Wireless Sensor Networks (EWSN 2008)*.

Støa, S. and Balasingham, I. (2011). Periodic-MAC: Improving MAC Protocols for Wireless Biomedical Sensor Networks through Implicit Synchronization. In *Biomedical Engineering, Trends in Electronics, Communications and Software*. InTech.

Texas Instruments Inc. (2006). *MSP430x161x Mixed Signal Microcontroller*. Dallas (USA).

Wittenburg, G., Dziengel, N., Wartenburger, C., and Schiller, J. (2010). A System for Distributed Event Detection in Wireless Sensor Networks. In *9th ACM/IEEE International Conference on Information Processing in Sensor Networks (ISPN2010)*.