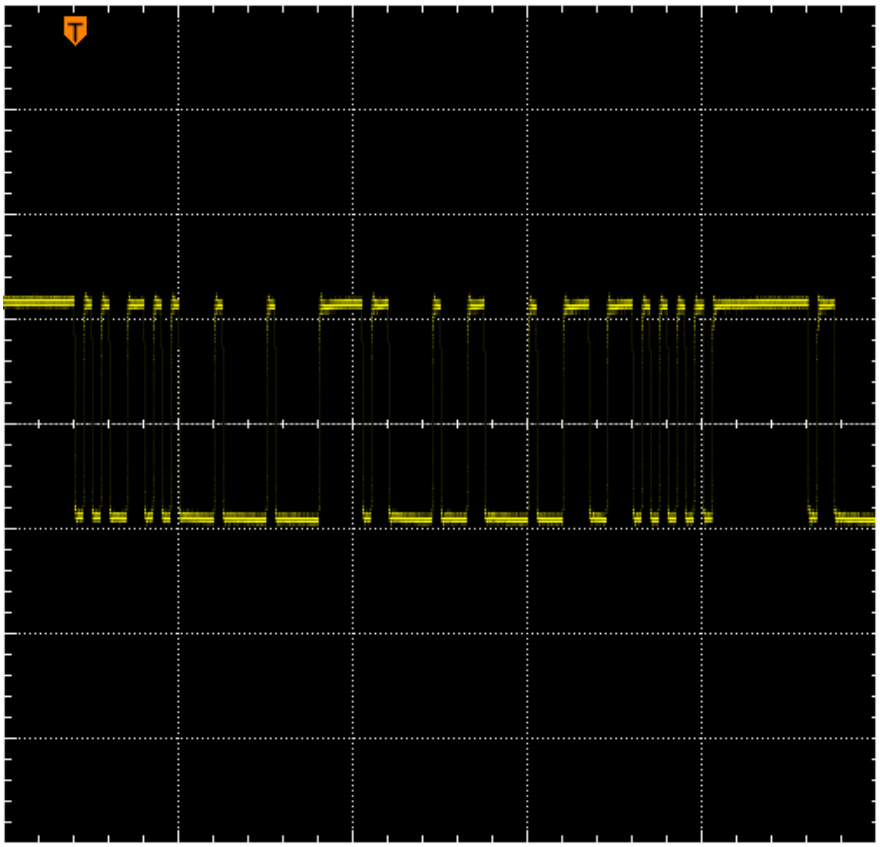


Clock Synchronisation Method Over Bandwidth-Limited CAN-Bus

Master's thesis in Embedded Electronic System Design

CHRISTOFFER MATHIESEN
DAVID KVIST

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2021

Master's thesis 2021

Clock Synchronisation Method Over Bandwidth-Limited CAN-Bus

CHRISTOFFER MATHIESEN
DAVID KVIST

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2021

Clock Synchronisation Method Over Bandwidth-Limited CAN-Bus
CHRISTOFFER MATHIESEN
DAVID KVIST

Supervisor: Kent Lennartsson, Kvaser AB
Supervisor: Lars Svensson, Computer Science and Engineering
Examiner: Per Larsson-Edefors, Computer Science and Engineering

Master's Thesis 2021
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Part of a differential CAN-bus transmission.

Typeset in LaTeX
Gothenburg, Sweden 2021

Clock Synchronisation Method Over Bandwidth-Limited CAN-Bus
CHRISTOFFER MATHIESEN
DAVID KVIST
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Clock synchronisation is an integral part of systems where the ordering of events is needed. While implementations exist on the CAN-bus to provide clock synchronisation, additional improvements are needed to provide a robust protocol while limiting the amount of traffic used in a system. Many solutions rely on the entire system architecture to be known or a timekeeper be preassigned before the start of the system, thus limiting the adaptability and/or scalability of the protocol. Here we aim to implement a solution that does not depend on knowledge of the number of nodes, and where each node can independently or collectively retain timekeeping regardless of what happens to the timekeeping master device. The timekeeping master node should not be necessary to be preassigned, instead, the system should be capable of arbitrating the master node at startup and achieve synchronicity. Furthermore, nodes should be able to arrive and leave an active system at any moment, also known as being hot-plug enabled. The nodes are envisioned to be of limited computational ability and not necessarily be constructed with the same hardware. As such a protocol capable of handling counter rollover and devices with different local oscillator frequencies is to be designed. While a lot of design insight was gained and a demonstrator system was partially implemented in hardware the task was underestimated and ultimately not completed. A complete HDL-implementation might be very possible, but it might not be better than a system comprised of both hardware and software.

Keywords: Timekeeping Synchronisation CAN FPGA Master-arbitration

Acknowledgements

This work is our final part in the Embedded Electronic System Design programme at the Department of Computer Science and Engineering at Chalmers University of Technology. The thesis was conducted at Kvaser AB and comprising of 30 higher education credits. We would like to express our gratitude to those helping us throughout the project.

First and foremost, we would like to thank Lars Svensson, our academic supervisor at Chalmers. Your academic guidance has been invaluable but most importantly your enthusiasm and inquisitive questions have spurred us on when the outcome seemed most bleak. We are truly grateful that you always have had the time for us.

Next, we would like to thank Kent Lennartsson for his mentorship, guiding us in the technical aspects of CAN and synchronisation protocols during our work at Kvaser AB. Without your guidance, this thesis would not have been possible.

We would also like to thank Professor Per Larsson-Edefors, our examiner, for his invaluable feedback and insistence on us wrapping things up, bringing this report to fruition.

A big thank you to all our colleagues at Kvaser AB for making us feel part of the company and always cheering us on. There was never a doubt that we had a spot at the office during the COVID-19 pandemic and that our work mattered.

Finally, we would be remiss if we did not mention all the friends and family supporting us during this work with whom we have discussed many issues.

Thank you all!

Christoffer Mathiesen & David Kvist
Gothenburg, September 2021

Contents

Abbrevations

BSP - Bitstream Signal Processor
CAN - Controller Area Network
CDCAN - Christoffer-David-CAN
ETP - Event-triggered protocol
FSM - Finite State Machine
GT - Global Time
KCAN - Kvaser CAN, *(Proprietary HDL-IP)*
LO - Local Oscillator
LT - Local Time
LUT - Lookup Table
MCU - Microcontroller unit
MT - Macrotick
PTP - Precision Time Protocol
µT - Microtick
TS - Timestamp
TTP - Time triggered protocol
WR - White Rabbit

1

Introduction

Systems with transient processes use time-keeping and the precision of such time-keeping to accurately process, record, and act upon events and sensor data. Systems of higher complexity require high precision and accuracy.

The accuracy of timestamps is affected by deviations of the local oscillator of the devices connected to the system. Synchronising the real-time clocks of each node is of interest to enable high precision data collection.

Existing solutions such as the WR *(White Rabbit)* [1] protocol, used by CERN, and other systems built upon IEEE 1588 PTP *(Precision Time Protocol)* [2] need some modification to fit into real-time systems with low-performance MCUs *(Microcontroller Unit)*. In addition, these protocols need to be streamlined to fit into bandwidth limited protocols, such as the CAN *(Controller Area Network)* protocol [3].

Many systems performing clock synchronisation, such as those using PTP, designate a master clock. If the master was to be removed from the system, be that from malfunction or otherwise, a fallback clock would have to take the previous master's place. The fail-over to the fallback clock is a relatively complex process taking significant time.

Systems such as PTP in many cases rely on a service technician being available for replacements and repairs if anything goes awry. This is not possible in most CAN-bus connected systems such as a network of CAN-nodes in a vehicle. If the system was to be deployed in the automotive sector, high reliability and fault tolerance would be required. It would incur prohibitive costs of servicing breakdowns of vehicles because of the failure of a time-synchronisation failure. Sending a technician for each such failure, even if rare, would not be feasible due to the number of vehicles in current service.

Some previous work in synchronising over CAN include TT-CAN [4], FTT-CAN [5] and DD-CAN [6]. These protocols use different methods in ensuring synchronicity, but they are not without some limitations. Of these protocols, some use predefined structures at design time, leading to inflexible systems, while others require additional overhead on the CAN-bus.

Synchronisation and time-keeping use data structures and frames which are many

times unsuitable to fit into the small frame-sizes of CAN. Due to the small packages and low bandwidth of CAN, the timestamps have to be short; in conjunction with the low power MCUs of CAN-nodes, the sizes of synchronisation messages used in PTP are infeasible. Indeed, the headers alone are 34 bytes in PTP [2, 7], meaning a complete PTP-message would take multiple CAN-frames to transmit.

Systems relying on PTP usually are high-performance computers with high accuracy clocks. In contrast, in a CAN-system, the variability of clocks of different nodes can be as much as 1.58 % [3].

With these things in mind, developing a system that combined works on low-performance controllers, performs clock synchronisation, and also minimise added traffic onto an already constrained communication bus is a worthwile endeavour.

1.1 Requirement Specification

In this section a summary of specifications for the system will be proposed. The specifications follow from discussions with our supervisor Kent Lennartsson at Kvaser AB.

1.1.1 Functional Requirements

The system should autonomously assign a master clock for synchronisation without the need for intervention of a technician. The following attributes, functions and features are expected to be implemented or supported:

- Synchronise clocks over CAN.
- Auto-arbitrate new master node when needed.
- Support for Hot-plugging.
- Detect and handle performance degradation.
- High MTBF (Mean-time between failure).
- High scalability.
- Retain temporal ordering of timestamps.

1.1.2 Hardware Requirements

The envisioned system will be a heterogeneous system of nodes. The system will depend on each node containing the same HDL-modules. Due to the dependency of HDL-modules the project will be limited to identical boards but with the capability of running at different clock speeds.

A system prototype shall be demonstrated on a network of development boards at the end of the project. Kvaser's KCAN-module [8] will be used to interact with a CAN 2.0B [3] network with a minimum of three nodes.

The implementation shall be conducted with Terasic DE0-Nano [9] development boards. These boards feature an Altera Cyclone IV FPGA and the capability of running embedded Linux on an ARM-based processor.

1.1.3 Software Requirements

Software running in the embedded Linux will be used for test, verification, and aid in our development. Our solution should as far as possible not depend on software tools.

1.1.4 Operational and Environmental Requirements

The prototype should work in a regular laboratory environment. Other environments are not to be tested.

1.1.5 Quality Test Requirements

It should be possible to practically demonstrate the synchronisation and hot-plugging of nodes in a network of three or more nodes. Hot-plugging here refers to the disconnecting and the connecting nodes of all types, including the timekeeping master node while the system is active.

1.2 Delimitations

No attempts to fit competing solutions into the same platform or adaption for the same physical layer will be done, even though this would be interesting from a benchmarking perspective. We are limiting ourselves to a CAN 2.0B solution [3]. We will only consider a single bus without any bridges.

1.3 Outline

Some key concepts of the project will be explained in Chapter 2 to give the reader a theoretical foundation in relevant topics. In turn, this will enable the reader to relate to the decisions and choices made, both in design but also delimitations chosen for the project. Chapter 2 will present the required metrics and performance requirements to meet the specifications of the product and fulfil the aim of the project.

The design process and the design choices will be explained in Chapter 3. This chapter also contains verification processes and how benchmarks have been designed to provide tangible results.

The implemented demonstrator system will be discussed in Chapter 4 and the results from tests performed on the physical prototype will be presented in Chapter 5.

Analysis of the tests and the implementation will be discussed broadly in relation to

the project goals in Chapter 6, along with alternative solutions, possible improvements and future work.

Lastly, the authors' inference of the project and results will be presented in Chapter 7.

2

Theory

This chapter contains theory for real-time systems, methods of ordering events, the fundamentals of the Controller Area Network (CAN) protocol and how clock synchronisation is conducted in some other protocols. Also, figure of merits for the performance of clock synchronisation will be discussed as will some concepts regarding time-perception.

2.1 Time-Perception and Ordering of Events

Distributed real-time systems or processes depend on a common time base to meet high standards of reliability and deliver accurate timestamps for data [10]. The use of a time base corresponding and concurrent to an *external* time base is not necessarily needed for the ordering of transient events in one or more of connected nodes in a system. The nodes in one system should however conform to a common time base internal to the system for ordering of events to be possible.

The common time base of a distributed real-time system is, within the field of real-time systems, colloquially know as *Global Time* (GT) [10]. Global Time is an abstract notion and might not necessarily correspond to any physical clock in a distributed real-time system [11], and is used to order events in the system and to verify and/or ensure consistency of the system environment, information in messages, and collected data. Local clocks in each node in a synchronised system create an approximation of a Global Time in the system from each nodes' local perception of the time, LT *(Local Time)*[12]. Preferably each LT have the same cadence and speed as the GT.

The ordering of events in the system might seem trivial at first glance. Temporal ordering of events comes as a natural way to order events for humans [13]. This does, however, not take into consideration the differing perceptions of different nodes in a distributed system. A number of spatially separated but distinct processes communicate over a physical layer, such as a bus, which at a very minimum induces spatial distance speed-of-light-delays. Queuing for access to the bus can further change the delivery time of information to some nodes.

Due to the spatial separation of nodes (or other delays) one node, Node A, might perceive the event, E_A, to precede a different event, E_B. If the event E_B occurs at a

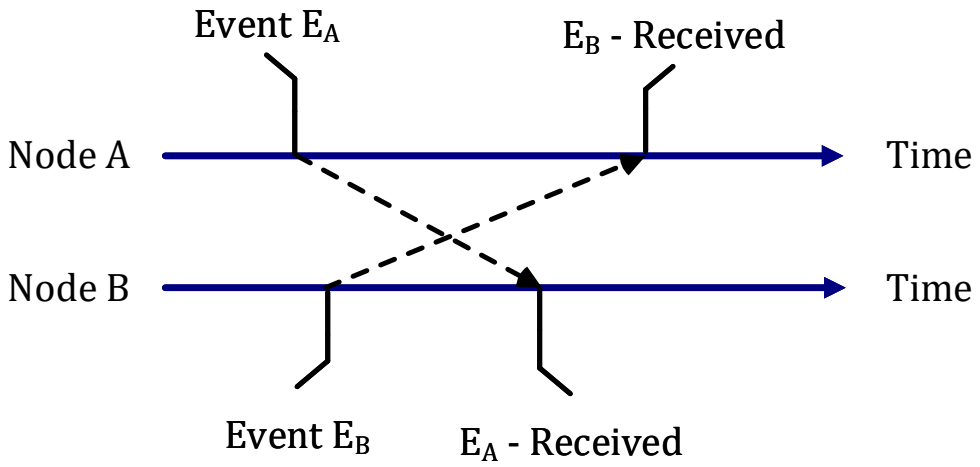


Figure 2.1: Ordering of events in nodes can give rise to different perceptions of the order in different nodes. Adapted from [13].

different node, Node B, the opposite ordering might be true for Node B's perception. This introduces an uncertainty of when an event has occurred, limiting the usefulness of such a distributed real-time system. Figure 2.1 demonstrates this phenomenon where the two nodes, Node A and Node B, have different notions of the order of the events E_A and E_B.

Causal ordering is temporal ordering but also includes cause and effect. For example, a connection of a data-transmission system must be established before a transmission can be sent or received [13]. Detection of this cause-and-effect phenomenon requires the system to not only being capable of temporal ordering but also ordering the causality: Which events cause, and therefore, precede another. The detection of cause-and-effect requires the nodes to have some notion or perception of time where each event locally can be timestamped and correlate to Global Time. This timestamp can together with an event be used at a different node to correctly order events in the system.

2.2 Time Concepts in Real-Time Systems

This section will introduce some time concepts and terms related to real-time systems.

2.2.1 Physical and Logical Clocks

The use of a universal time, such as the international reference time scale UTC *(Coordinated Universal Time)*, for tracking the order of internal events is seldom practical in an MCU *(Micro-controller Unit)*. MCUs usually interact on a different time scale than UTC, instead of having a shorter time span of interactions. Even though MCUs may not need to correlate to any other time-keeping standard, the limited purview does require some perception of time and order of events as discussed in Sec-

tion 2.1, this necessitates some kind of clock. The most rudimentary time-keeping is done by purely ordering events by arrival times, without a perception of the gap of time between any two events. This method of ordering events relies on a *logical clock* rather than a physical clock tracking discrete units of time. A logical clock is not sufficient for most control systems, nor is it very robust since nodes in a system might have different perceptions of the order of events. The concept of time in a system with a logical clock is tenuous at best.

A physical clock can be introduced to the system to enable timekeeping in a system by time-stamping events, hence improving the ability to order events. In addition, a physical clock enables the system to relate to distances of events in time rather than only the order of events. Clock pulses provide a known *cadence* to the system and are the primary carrier of time. All time-perception of the node is related to this clock. To create the necessary discrete time-units used in a real-time system, physical clocks are constructed consisting of counters and LO *(local oscillator)*, which provide discrete and reliable pulses [14, 15]. The oscillation frequency of the LO is not perfect but subject to the physical characteristic of the oscillator, which has some variance [11], as well as external factors such as temperature. In the context of this project, we assume all clocks to be digital and with discrete increments of time.

2.2.2 Units of Time

The LOs in two different nodes in a distributed real-time system can be synchronised to a common frequency; however, this requires hardware capable of acting on each clock pulse of the LOs. This also requires the nodes to be equipped with oscillators capable of reaching the same frequency as well as being equipped with hardware capable of tuning the resulting frequency. A less computational heavy method is to use intermediary *ticks*. These *ticks* which consist of a number of local oscillator pulses are the output of a physical clock as mentioned in previous subsection. These discrete time units have the potential to be the basis for the real-time system [11]. The smallest tick, or time-unit, in a node is not necessarily usable for synchronisation if its period is very short.

The ticks spanning multiple physical devices should be tuned to a common time base for nodes with different internal frequencies. Since the ticks are directly derived with the frequency of each nodes LO, the ticks are susceptible to the variance of the LO [16]. With the selection of the size of the counter, used for the generation of a tick, the *granularity* of the tick can be selected. A *granule* is the distance between two ticks, a small granule results in a fine granularity. Two synchronised nodes can be observed in a timing diagram in Figure 2.2.

In the same way, as the tick consists of several pulses from the local oscillator, multiple ticks can be used to generate a longer granule, a *macrotick* [17, 16]. Any of the levels of generated ticks, microtick or macrotick, can be what is shown outwards for timestamping between nodes. They are, in all levels, adjustable to compensate for drift or the tolerance of the LO. The period of the externally exposed tick can

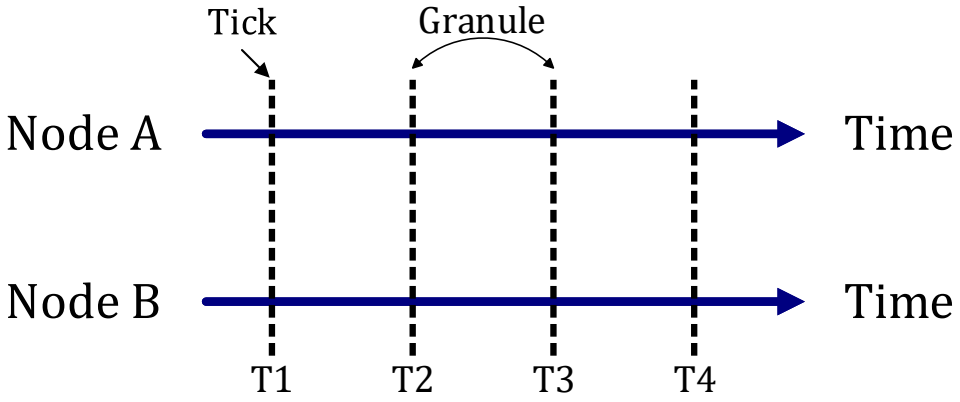
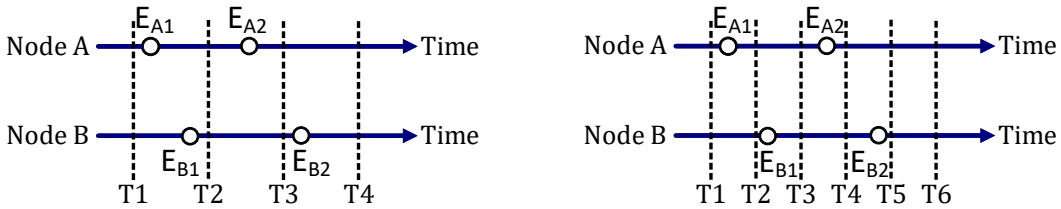


Figure 2.2: Two synchronised nodes and their granule and ticks. Ticks correspond to distinct moments in time, and granule is the distance between ticks.



(a) Ordering of E_{A1} and E_{B1} not possible.

(b) Ordering enabled through smaller granules.

Figure 2.3: Finer granularity enables ordering of events.

be considered to be the *cadence* of the Global Time in the system.

As mentioned in Section 2.1, events must be separated with timestamps of a minimum of two or more ticks for the temporal order be recoverable [17]. Two events occurring within the period between two ticks can be observed in Figure 2.3a which results in the order of the events not being recoverable. Nodes might have different perceptions of which event occurred first. Hench, a finer granularity is desirable to lower the risk of two events colliding.

There is a practical limit to how small a granule can be which is related to the system performance. A very small granule will give the system only a few clock cycles to execute instructions. In a system where time-perception needs to be common between all nodes, the granularity will be limited to the slowest node in the system [16] since the granule size is linked to the nodes' tick size. If a finer granularity is needed, the ticks can be generated from the LO and a Phased Locked Loop (PLL), generating a higher frequency than the frequency of the LO.

The absolute number of ticks or time-units which fits in the memory, or time vector, is a unit of time called an *epoch* [14]. The epoch can be a select number of seconds, ticks or some other discrete time-unit, which cover at least the longest transient event the system should be capable of tracking. The time period an epoch covers

directly depends on the number of bits available to store the data and the required resolution of the system.

If the ticks correspond to a beat of 1 Hz and the available vector is 16 bits long, then the total epoch length is 65 535 s, or 18.2 h.

2.2.3 Sparse and Dense Time

The synchronisation of the system can occur at different intervals. With regards to units of time, a synchronisation event can be triggered on each tick or with a period of several ticks apart [17]. Synchronisation on every tick, called *dense time*, gives a predictable system but has some drawbacks such as all local clocks or lengths of ticks must be able to relate to the same time base.

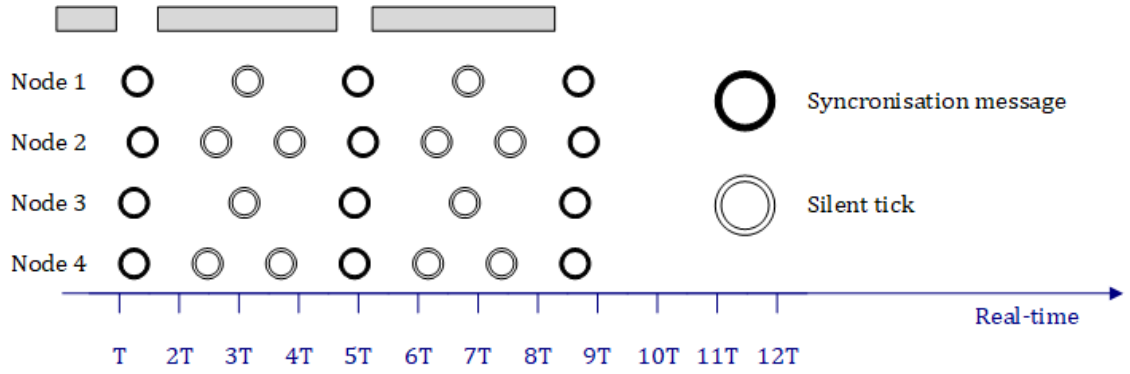


Figure 2.4: Timing-diagram of ticks in a system using sparse time. The black circles denotes an instance where the system synchronises. The gray bars denotes periods where the system nodes does not necessarily have a common time-perception.

Alternatively, *sparse time* can be used where a period between each synchronisation event elapses with no synchronisation communication and time is kept internally by the nodes' own Local Time. A sparse time system can be observed in Figure 2.4. Sparse time requires all nodes' local clocks to have the same cadence, or an integer factor thereof, to enable synchronisation with a common interval.

2.2.4 Linear vs Cyclic Time Representation

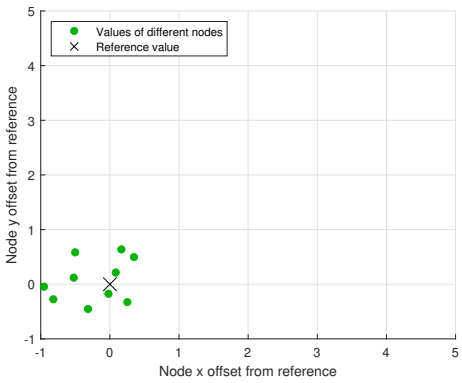
Events can be arranged linearly on an infinite or finite timeline. Shorter, repeating events, however, can be represented cyclically. If a task repeats indefinitely and each constituent task is in the same order in each event the event can be considered cyclic. One such system could be a data-sampling system collecting data from several sensors. If the system is to sample at a rate of 50 Hz each event will need a period of 20 ms for timestamps. Each sample event will be identical.

The total timeline of a system limited to processing cyclic and transient events will only need to be greater than the longest transient event. The representation of time in a digital system uses a finite vector, the epoch, which will inevitably roll over if allowed to progress for a sufficient amount of time.

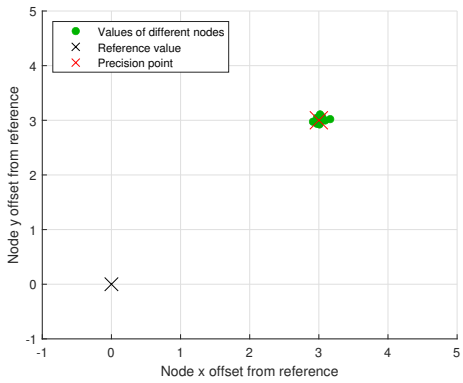
An algorithm needs to be devised to separate reoccurring cyclic events and as well handle if such an event is to happen during a rollover of the epoch.

2.3 Oscillators and Physical Clocks

The precision of a system corresponds to the reciprocity and cohesion of nodes in the system. The absolute value might not be strongly defined or deviate from a given reference value. Accuracy is how well the system's value matches a known reference or external source. Figure 2.5 demonstrates two systems. One has high accuracy, but is poor in precision, while the other has the opposite characteristic, being low in accuracy and high in precision. For this project, precision is more relevant than accuracy.



(a) System with poor precision but high accuracy.



(b) System with high precision but poor accuracy.

Figure 2.5: Precision and accuracy have different perspectives. Accuracy is how close to a set target values are. Precision is how close the grouping of values are.

A number of free-running clocks will deviate after some time, and eventually leave synchronicity even though they were initially synchronised [17]. This phenomenon of drifting clocks and oscillators forces distributed real-time systems to continuously adjust the local clocks in the nodes to keep the Local Time in line with the system's perception of Global Time. The frequency of synchronisation will bound the maximum offset of any two local clocks, thus also the worst-case precision.

The *drift rate* [17] ρ of a clock k, with respect to a reference clock m, can be calculated as,

$$\rho^k = \left| \frac{z^k(t+n) - z^k(t)}{z^m(t+n) - z^m(t)} - 1 \right|, \tag{2.1}$$

and is quantified by comparing the difference between a local clock's tick values at two discrete points in time t and $t+n$, and dividing the difference with the nominal amount of ticks in the reference m (typically master). z^k and z^m denotes the tick-values for clock k and m respectively.

The drift rate describes how many parts of a tick the local clock drifts compared to the reference. A perfect local clock will, therefore, have a drift rate of 0.00 tick/tick,

whereas a local clock that is one tick ahead (or behind) the reference after five ticks will have a drift rate of 0.20 tick/tick.

The longer a local clock with a non-zero drift rate runs, the worse the difference between it and the reference, called *offset*, will become. In a distributed system of n nodes, the *worst precision* [17],

$$\Pi = \max_{\forall t;\ \forall i,\ j \in \{1,2,\cdots,n\};\ i \neq j} \left| z^i(t) - z^j(t) \right|, \tag{2.2}$$

of a distributed system's common time-perception at time t is bound by the maximum offset between any two local clocks i and j (not necessarily between a master and a slave).

Equation 2.2 gives the worst possible offset in ticks between any two local clocks, and as such the finest time-resolution the distributed system can have. A value of two means that each local clock is at worst within two ticks of each other. A value of zero is equivalent to absolute precision, a system with all clocks perfectly synchronised at all times.

The required precision of a system has to be defined during the design process and will affect the required synchronisation rate R_{int} [17]. A local clock may also be offset somewhat from the reference even at the moment of synchronisation, and this is denoted as the *convergence function* Φ. The *drift offset* Γ indicates the worst divergence of any two clocks that are supposed to be synchronised and is defined by the function $\Gamma = 2\rho R_{int}$. With the precision equation, convergence function and the drift offset the required *synchronisation condition* for an ensemble of clocks is,

$$\Phi + \Gamma \leq \Pi, \tag{2.3}$$

which is visualised in Figure 2.6. If this synchronisation condition is violated, a node can not guarantee that it has the same perception of the global time as another node, and correct temporal ordering might not be possible.

2.4 Time Synchronisation

Synchronising discrete clocks comes with a few problems. If the built system is in a strict master-slave relationship, the system will stop being synchronised if the master is unable to communicate. If a backup master takes the role of master in the case of the original master being unable to communicate, the backup master needs to be aware of such a fall-back. If the original master returns after the backup master has assumed the time-keeping role, this situation must be detected and handled. If a slave becomes desynchronised, it must be able to reacquire synchronisation from the master.

Besides being able to handle the different nodes' states concerning each other, synchronising clocks is not as simple as the master just sending out a message telling the slaves what the time is. Heed must be taken to the communication protocol used and the physical layer itself. In the communication protocol, situations can arise

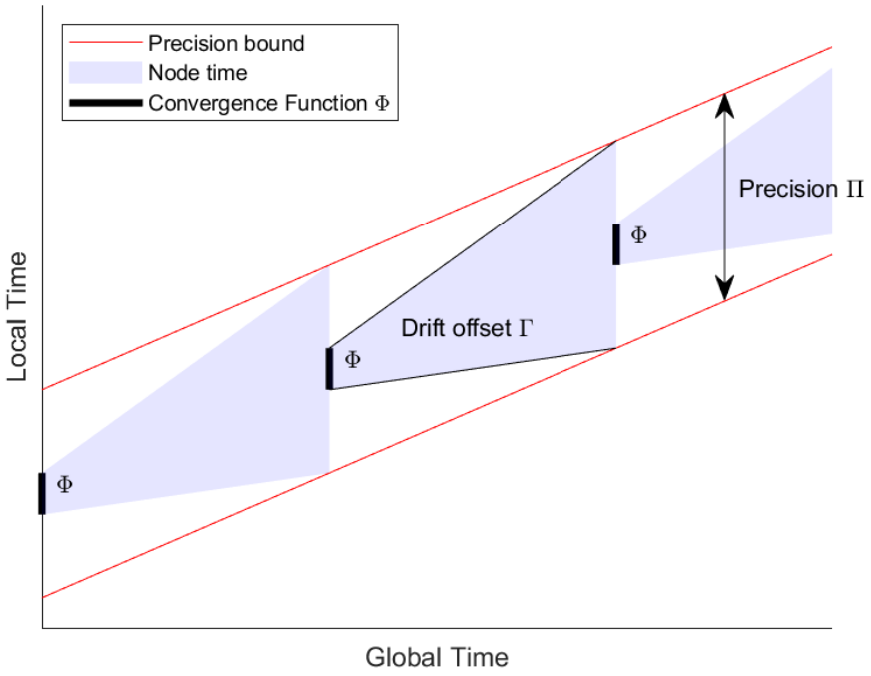


Figure 2.6: Visualisation of the synchronisation condition, adapted from [17]. The black bars denote a synchronisation of the LT to that of the GT, the local clock is set to a value within the convergence function Φ. Precision bound limits what values local clocks can assume and still be considered synchronised. The span between the upper and lower limit forms the worst precision Π. After each synchronisation, the local clocks continue to drift and is shown as increasing divergence (shaded area). The maximum divergence of two clocks are limited by the drift offset Γ. An increase in synchronisation frequency and lower LO-drift (Γ) can decrease the area between the upper and lower boundary which in turn improves the precision Π.

that affect the time it takes to transmit the synchronisation messages. For example, urgent messages with higher priority than that of the clock-synchronisation being sent first (queues and/or preemption). For any transmission, the physical propagation time will affect the time from when a master sends a message and a slave receives it.

Yet another complication for high accuracy in synchronisation is the frequencies and phase if the local oscillators in different nodes. A node with a high clock frequency can naturally have higher maximum accuracy in time-keeping, but this maximum accuracy can not be communicated to another node with a slower clock. Instead, the system will be limited to the accuracy of the node with the worst maximum accuracy. Further, if two nodes' oscillators are out of phase with each other they are going to sample the incoming synchronisations messages at different times, making the nodes have a slightly different perception of time even at the time of synchronisation.

Many different protocols to synchronise clocks in distributed systems have been developed. Some examples are *Network Time Protocol* [18] for distributing time over the internet, and *Precision Time Protocol* [2] and *White Rabbit Protocol* [1] for high precision clock distribution in an Ethernet-based LAN-network. The idea is for all nodes on the network to have a common time-perception, thus be able to communicate accurately with other nodes about, for example, when certain data was produced.

2.4.1 External Synchronisation

One of the easier ways of synchronising a system is to use a reference clock. One such system is a system which uses timestamps from an external GPS-data to receive time and synchronise a local clock. This method relies heavily on the external source to be accurate since there is no control over this external unit. The distance from the external source, both physically and in terms of data transmission, will affect the accuracy of the timestamps.

2.4.2 Precision Time Protocol

PTP *(Precision Time Protocol)* is an IEEE standard (IEEE 1588) published in 2002 [2, 14]. The goal of this protocol is to synchronise clocks in a packet-switched LAN with very high precision, even sub microsecond accuracy [2, 14]. This is done by a chosen master sending out synchronisation messages to slave nodes. In essence, a master sends the slaves two messages, where the first is a synchronisation message and the second is a follow up which contains the time when the synchronisation message was sent. The slave can then calculate the elapsed time between the received messages and adjust the local clock to match that of the master clock (which should correlate to the global time). Further, the slave also requests the propagation delay time from the master to the slave by sending the master a message in between the synchronisation-message and the follow up-message. The response is also part of the follow up. With these three messages, the slave can set its clock very accurately compared to that of the master.

Because PTP is made for packet-switched networks with potentially multiple ports per node, each node has to be able to act as both master and slave on different ports, depending on which port is communicating closest to the node with the *best clock*. The best clock is defined as the local clock in the system with the highest accuracy to the reference clock (typically given by GPS). In a system with multiple layers of nodes (for example, a tree-formed topology system) the node connected to the reference would become the best clock of the system (grand-master), the node connected to the grand-master node become a slave to the grand-master node while acting as master towards other nodes connected to it. There could be constellations made where multiple nodes are best clock nodes. Then each node would have to make a decision on which port should become the slave port, and which port should be passive.

PTP usually relies on hardware timestamps to achieve required accuracy but the use of software timestamps are possible in a clock synchronisation protocol [19]. However, software based timestamps decrease the accuracy of the protocol as each transmission has to get through the transmission stack before being sent, making each message inherently have delay and uncertainty of transmission time. The hardware solution is to insert the actual GT value *on the fly*, at the send-time of a synchronisation message.

2.4.3 White Rabbit

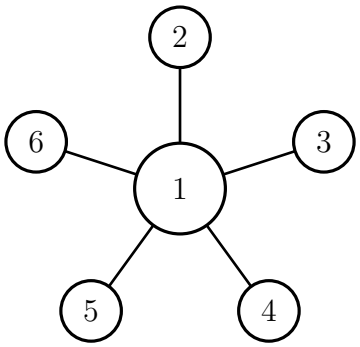
White Rabbit (WR) is a protocol that builds on PTP and *Synchronous Ethernet* to get sub-nanosecond accuracy [1]. It uses much of the functionalities found in PTP and adds upon it by synchronising the nodes' internal clocks to that of the master's by adjusting an internal Phase-Locked Loop (PLL) to exactly match the master's clock frequency. This is done by the master outputting a known set frequency which the slaves, in turn, recovers out of the transmission. This recovered frequency is then used to manipulate each slaves' PLL until they completely matches the frequency of the master. By tuning the frequency like this, drift is greatly reduced. In order to increase the timestamps' precision further the phase difference between the nodes and the master, PLLs are calibrated in order to compensate for differences in sampling time [1].

2.5 Network Topologies

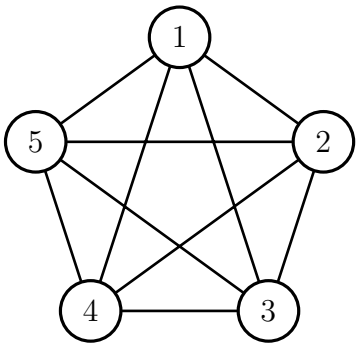
Networks may be ordered in different ways, and the chosen topology will alter what solutions for time-synchronisation are possible on the network. In Ethernet (the protocol used by both PTP and WR) the data-transfer topology can be set up in multiple ways, such as the mesh- or the star-topology which can be seen in Figure 2.7.

The star topology has low level of redundancy due to all traffic is traversing the centre node but requires few interconnects. The mesh structure in Figure 2.7b requires point-to-point connectivity which in turn enables each node to independently communicate with any other node in the network. PTP and WR, as packet-switched

time-synchronisation protocols, are designed for these topologies, but not necessarily limited to them.



(a) A star network of six connected nodes.



(b) Nodes in a mesh configuration.

Figure 2.7: Examples of star and mesh networks.

PTP uses the existing topology to evaluate what roles nodes will have. The node closest to the outside reference will become the master for nodes connected to it. Those slave nodes in turn will become masters towards any other nodes that they are connected to. For this to work nodes will have to have knowledge of which other nodes are participating in the time-keeping protocol to be able to synchronise. In the case of a node that does not support PTP or WR, any other nodes that have this non-supporting node as the only connection in the data-transfer topology will not be able to synchronise over PTP or WR.

In CAN the network topology of choice is the bus structure. With this structure all nodes share a common communication path, and as such all nodes are able to see all messages transmitted on this bus. The downside to the bus topology is that only one node can transmit on the bus at a time, which greatly limits the amount of data that can be transferred, and as such also the amount of bandwidth available to a time-synchronisation protocol. A benefit of the bus is the low cost of interconnects and that nodes do not have address specific nodes as in a mesh-network but rather broadcast messages onto the communication media. Messages meant for time-synchronisation would as such be available on the transmission medium for all connected nodes simultaneously.

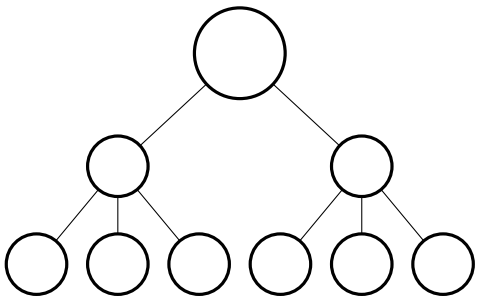
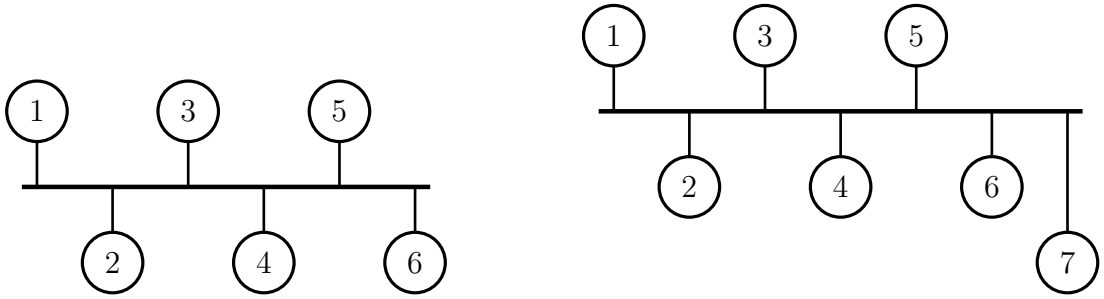


Figure 2.8: A communication tree with master-slaves hierarchy.



(a) Bus structure of six connected nodes.

(b) A communication bus with a node connected at a longer distance.

Figure 2.9: Two bus networks with different time-of-flight for transmissions.

Data transmissions on a network topology are affected by transmission delay, and different topologies change the node distribution such as the node receives the transmission with a delay. If the analogue wavefront on the bus of a bit is used for synchronisation the different nodes will have a skew corresponding to the time-of-flight and the phase uncertainty of the node's internal clock used for sampling the data bus. Additional communication can be used to calculate time-of-flight between transmitter and receiver in a system. Time-of-flight refers to the transmission delay of a package in a transmission medium.

If a node communicates on network using a bus-topology the positioning of the transmitting node and the receiving node affects the time-of-flight such as the network in Figure 2.9b having a longer worst case time-of-flight between nodes than the network which is depicted in Figure 2.9a. In a bus the time-of-flight is internode dependent.

A star topology with connection links between the centre root node 1 and outer nodes 2–6 of equal length as in Figure 2.7a would be easier to use for a time-synchronisation protocol than a bus since the time-of-flight calculation only goes from the root to the outer nodes.

PTP and WR would work in a bus topology, but are not designed for it. As PTP and WR are built for packet-switching networks, synchronising the propagation time needs individual messages from and to each specific node in PTP and WR, meaning that the more nodes that are connected to the bus, more of the maximum bus-bandwidth is used. Unique message-IDs would have to be bound to each node as well, and this, combined with the limited amount of possible IDs, means that the maximum number of connected devices is limited.

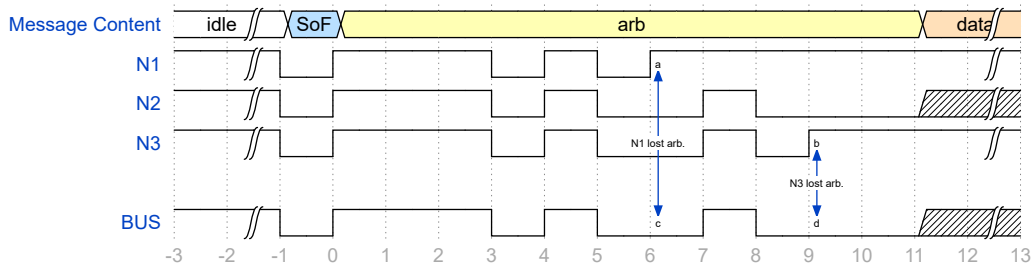


Figure 2.10: Three CAN nodes in arbitration, low level signifies a logic zero and is a dominant bit. At T=6 the N1-node tries sending a recessive bit and loses the arbitration. At T=9 the N3-node sends a recessive bit and loses the arbitration. N2 successfully wins the arbitration at T=11 and gets to transmit its data.

2.6 Controller Area Network (CAN)

The Controller Area Network (CAN) was originally developed by Bosch, [3] the original specification has since been incorporated in the ISO standard 11898 [20]. CAN is an asynchronous serial communications protocol primarily used in the automotive industry, but can be found in other real-time control systems. It has been expanded to support higher speeds but the CAN 2.0 standard supports bitrates only up to 1 Mbits / s.

CAN has a structure consisting of a data link layer and a physical layer [3]. The physical layer defines how the signals are transmitted and contain descriptions of synchronisation, bit timing and bit encoding. The data link layer processes error detection, frame decoding serialisation/deserialisation and more.

The physical layer of the CAN specification does not specifically describe which physical medium is used for the transmission of data but provides flexibility in the selection of physical medium. As each connected CAN device shares the same physical medium at all times (bus topology) all devices are jointly responsible for access arbitration, error-detection and fault-mitigation through the data link layer.

2.6.1 CAN Arbitration

Each CAN-device shares the physical medium and each device is capable of initiating a transmission at the same time. Which device gets access to the medium is decided through an arbitration process as shown in Figure 2.10 where three nodes arbitrate for access to the bus. At the start of each frame, there is an identifier and the message with the highest priority (lowest binary number) is the one that gets access to the medium. Because the physical layer's dominant level is a logic zero (meaning transmitted zeros are seen over that of simultaneously transmitted ones) this will provide the highest priority message access to the shared resource. Messages with lower priority (higher binary number) will be delayed and put to wait until the bus is free for retransmission.

The internal queue systems of a CAN-device may or may not order messages after

priority. Regardless of the structure of the queue, there is no way to guarantee the transmission of any specific package on the medium itself, both in regards to timing, or even access at all. If the bus is occupied fully with transmissions of higher priority, the lower priority packages will have to wait indefinitely. It is therefore not directly possible to correlate a software-generated timestamp to the actual timing of the package itself onto the bus.

2.6.2 Medium Delays

All transmissions are subject to the inherent delay from the medium and distance the signal will travel to reach the recipient, CAN is no different. However, because of the arbitration scheme described in Section 2.6.1, the maximum transmission delay possible within any specific CAN-system is depends on the bits/s-rate. As each transmitting node during the arbitration phase need to have the identifier bits they transmit compared by each other transmitting node's transmitted bits, the transmission of any specific bit need to reach each node before the next bit is transmitted. If the maximum transmission delay is overshot, the likelihood for erroneous transmissions is greatly increased to the point that a specific system might stop working. In the case of a CAN-system running at 1 Mbits/s, this maximum allowed propagation delay equals 1 µs. There is no lower bound.

2.6.3 Synchronisation Protocols in Bus Topologies

As the CAN protocol is commonly used in vehicles today, which consists of multiple nodes, synchronising these nodes is not a new problem. Multiple implementations in different layers have been designed to solve synchronisation, which all have inherent pros and cons. Protocols exist for synchronisation over Bus-networks; TT-CAN, FTT-CAN, DD-CAN and FlexRay to name a few [4, 5, 6, 16].

Flexray uses continuous periodic updates from a set master node to synchronise the slaves to a common global time [16]. DD-CAN instead sends the delay-times of a message included in the following message [6]. TT-CAN and FlexRay require that all data-transmissions are predefined and scheduled at the time of designing the system, and is therefore not able to cope with dynamic systems with changing or adapting structures. The implementations of FTT-CAN and DD-CAN create additional overhead in all communication on the bus. TT-CAN, DD-CAN and FlexRay all require specialised hardware solutions to be implemented in existing CAN-controllers [6].

2.7 Master Assignment and Consensus Forming

In systems that depend on the decisions made by one master node, such as keeping track of the time, the assignment of the master constitutes a problem itself. The system has to either have prior knowledge of which node is a suitable master, be able to discern a suitable master from the topology or have the participating nodes agree which node will be assigned the role. In PTP and WR the decision of which

node is a slave and which is a master depends on a node's position in the topology. A node that is connected to a reference (typically a GPS) will take on the role of master, and a node connected to this master node will take on the role of a slave. The problem with such hierarchies is that if the master node get disabled, its slaves will be left without a master (desynced), and can as such not be able to guarantee synchronisation in regards to the reference anymore.

Systems that have to make a common decision on which node gets assigned the master role have to use communication and a common algorithm to achieve the master-slave relation. One such algorithm for reaching consensus in real-time systems is the Raft algorithm [21]. In this algorithm, a system decides on a master by each node voting on which node it believes should become the master. Once a node reaches a majority vote, it takes on the role of master. In the case of the master being disabled, or one node reaching a time-out threshold, the system will get desynchronised. Once the absence of the master is detected, the remaining nodes will have another voting session, and another node will be assigned as master. In the case of using this method for synchronising time, A disadvantage of this method of time synchronisation is that it is system-specific and may not be able to be interpreted by outside observers as there is not necessarily any outside reference.

2.8 Evaluating a Synchronisation System

This project will mainly propose, implement and demonstrate a system capable of performing the tasks specified in Section 1.1. Some interesting metrics of a functioning system could be tied to, upstart time and initial synchronisation of a system, robustness when hot-swapping master nodes, achieved drift-rate of synchronised nodes etc.

Issues of importance for any product is that it works as intended and that any limitations are known. To prove the correctness of a distributed system, one has to first formalise the model, then analyse it through mathematics and logic. The results then have to be interpreted and applied to a real-world test [17]. The first tests verify that the system works in isolation. After this, it has to be shown that it still works even when coexisting with other devices or functions. For both in isolation and combined it has to be shown that the system can handle potential faults. Things of interest to test for a bus-based clock-synchronisation algorithm include the correct functionality of the different states, the possible precision achievable, the required load on the bus for correct behaviour, the startup and desynchronisation transient, system up-time, and robustness.

Functional verification is to be conducted before performance tests can be performed. For features that either works or do not this entails simply running the system and forcing it into a state in which the function under test should trigger. If the expected outcome corresponds to the observed, then the function can be considered verified.

To test quantifiable metrics data has to be collected and analysed. The startup and desynchronisation transient times are such figures of merits, as these impact the

robustness and recovery time of the system. Further, knowing how the number of nodes affects these transient times are of interest.

A figure of merits that could be used to quantify the correctness of the timekeeping is to look at the average error of the combined subservient nodes compared to the timekeeping node's interpretation of the time. This difference in time-perception could be both ahead of or behind the timekeeping node. It may also swing from ahead to behind multiple times compared to the timekeeping node's time-perception through the test duration. As such a median- or a mean-value might be misleading since a series of positive and negative offsets can average out to zero. An RMS-value *(Root-mean-square)* would handle both positive and negative numbers and could be used as a comparison metric of systems with different configurations, although should not necessarily be considered an absolute performance metric. The RMS-value could be plotted as the precision value as in Figure 2.5b giving a snapshot for interpretation.

While the system is to use multiple nodes the LOs of each node will differ as described in Section 2.3. How large difference in the LOs' cadence the system can handle and how fast the system can adapt to changing cadence is of interest.

The clock-synchronisation protocol will use a communication network where other traffic can affect the system performance. How traffic congestion affects the precision and robustness of the system will be tested.

3

System Design

In this chapter the envisioned design of our system and synchronisation algorithm will be presented. In addition, we will present how the design is to be realised in hardware. The function of the system will be described, such as how a system in steady-state behaves and how an uninitialised system achieves synchronicity between nodes.

The top-down system description is followed by a system-level protocol explanation. In addition, the node-level behaviour and node transitions will be described. This includes start-up behaviour to steady-state and fault-handling.

3.1 System Description

The system is envisioned to be a heterogeneous system consisting of multiple but different nodes connected to a communication bus. The system is to be implemented with CAN communications in mind. To facilitate the hot-swap capability of nodes in the system, each node capable of using the synchronisation-protocol will use the same HDL-blocks.

The system clock of each node should be regarded as free-running and not necessarily of the same frequency. While all nodes are using the same HDL-blocks and each node should be functionally indistinguishable, the accuracy and performance of each node could differ greatly. Be that from oscillator accuracy or lengths of registers limiting precision. All nodes should capable of leading the system as a master or follow as a slave albeit with different capability. Slave- or master-role will differ only by which control are used by an FSM *(finite state machine)* issuing these control signals. Thus a single flag (Master/slave) signalling node-role will change the state for each node. Nodes being unsuited to be master, perhaps due to a poor physical clock, should be rejected by the system.

When the system is in a steady-state a master is responsible for the transmissions of its LT *(Local Time)*, on the bus in regular intervals. Unlike a slave, whose LT is its perception of GT *(Global Time)*, the master's LT is the basis of the system's GT. The interval between each synchronisation transmission needs to be sufficiently short to bring drifting nodes' LT back in line with the system's GT. The interval of these transmissions depends on the synchronicity between the master and the node

with the oscillator that drift the most, as described in the Section 2.3.

If the system bus is disrupted or if the master is silent for a prolonged time the nodes should be capable of retaining GT, arbitrate a new master in the system and synchronise without big changes in GT.

3.2 The System-Level Protocol

To enable synchronisation each participating node has to use a common protocol. The CDCAN-protocol, *(Christoffer-David CAN)* for a lack of a better name, is a system-wide protocol for the clock synchronisation over CAN. This includes rules for communication and information formatting on the CAN-bus. Role assignment, fault handling and timekeeping is also part of the CDCAN-protocol.

Nodes not using the CDCAN-protocol should be capable of sharing a CAN-bus with devices conducting clock-synchronisation. Hence, enabling the CDCAN-protocol to be used in most if not any existing CAN systems. This is done by reserving a few numbers of CAN-addresses to the synchronisation protocol.

The system as a whole will have three distinct states: 1) startup-state; 2) running-state; and 3) fault-handling-state.

The difference between a master node and a slave node is what actions are conducted during the running-state at a node level. The running-state can be further divided into the two states *sync* and *assert*. Sync is when the node is waiting to transmit a sync-message due to a timer running out, or when waiting for a sync-transmission on the communication bus. Assert is when the system is awaiting a time-stamp of GT from the master node.

3.3 System Startup and Master Assignment

A node at start-up has no knowledge of other nodes and the system as a whole can be considered to consist of a collection of nodes with free-running clocks. An algorithm for assigning master needs to be designed. The methods WR *(White rabbit)* and PTP *(Precision Time Protocol)* relies on a different network topology than CAN and is a poor fit. The Raft algorithm as described in Section 2.7 is unsuited to use on a dynamic CAN-system where the number of nodes is unknown. Tallying a majority vote might not be reasonable in a CAN-system due to: 1) Voting is done sequentially and might introduce a long transient, 2) nodes can spoof themselves as multiple devices since CAN-IDs are not node specific or reserved, 3) the number of nodes in the system is unknown and can change throughout. Hence a CDCAN-protocol is derived with the following behaviour:

The primary task of the system at startup (T_0) is to determine if an already established network of synchronised nodes exists on the CAN-bus, and in this case, join the system with minimal disturbance.

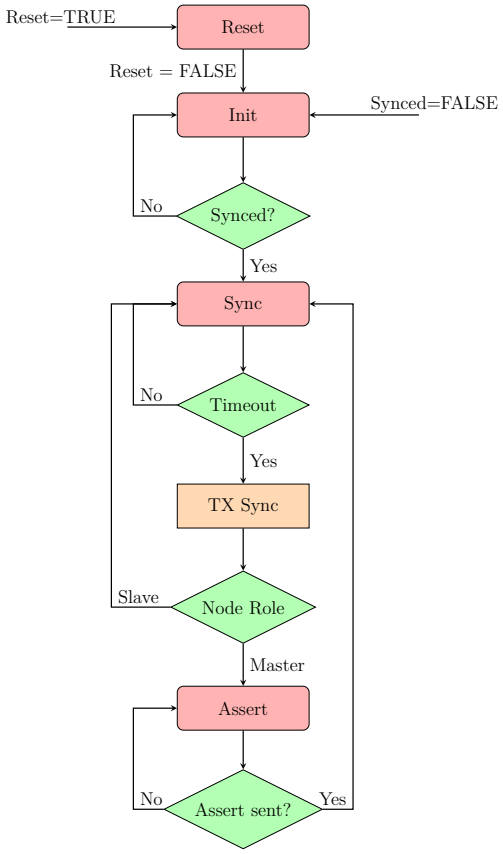


Figure 3.1: State transitions of the system, Sync and Assert-states are included in the running-state. The assert-state is only conducted by nodes with the Master-role, when a assert-message has successfully been transmitted the system transitions to Sync and awaiting the next sync-frame.

In the case that there is no obvious system on the communication bus the nodes will wait until a predetermined time has passed (T_1), determined to be the minimal, or optimal start-up transient, to ensure an established system's traffic would traverse on the bus and be detected.

If no established network has been detected within the allotted time frame T_0–T_1 the nodes will start a counter which depends on the node's perception of the cadence of ticks used for timekeeping in LT. The purpose of using local timekeeping makes it more likely that a node with fast running LO will assume the master role. This property is used in turn to facilitate slower nodes to turn their LT-value forward instead of backwards, as is required for sustaining casual ordering of events, as explained in Section 2.1.

When a sufficient amount of ticks of the LT-counter has passed (T_2) the node will transmit a synchronisation package. Unless another node challenges the synchronisation package the transmitting node will assume the role of master and a system has been established. The no-collision property CAN uses during the arbitration process mentioned in Section 2.6.1 is utilised in the protocol to avoid collisions between two potential masters with the same or very similar tick-rates.

The rejection by other nodes of the initial synchronisation package is based on when the synchronisation package is received and if it is received in a window when a new master is allowed. A message from a node outside the allowed window should be discarded since these nodes are deemed to be outside acceptable cadence, either being too fast or too slow.

The nodes can assume three different roles: 1) Master-role, the node assigned to timekeeping in the system. 2) Slave-role, the following node updating with the data from Master. 3) Unassigned, a node in startup which is yet to join or establish a CDCAN-network. In addition, a node can either be synchronised or unsynchronised where the latter always is the case for a node with the *unassigned* role.

3.4 Nominal Run of a Synchronised System

When the system is synchronised, the designated master node will periodically send out synchronisation messages for slave nodes to synchronise with. The state transitions and conditions depicted in Figure 3.1 are for nodes of both roles.

The CDCAN-protocol synchronises with two different messages. The first message, Sync_1, is transmitted as a notification that an updated time is to be sent. A following message, Assert_2, is transmitted a time later and contains a timestamp of what the master considered the GT was when Sync_1 was sent.

The diagrams in Figure 3.2 show how the two node-roles involved in synchronisation act. The slave nodes will update their LT-tempo in accordance with the value received in Assert_2 plus the difference in time of receiving Sync_1 and Assert_2. The nodes' perspective of this sequence of events is shown in Figure 3.3.

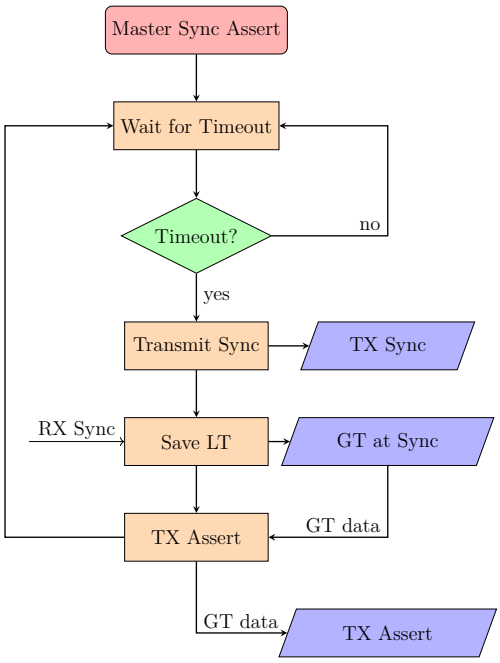
With the master transmitting the Message Sync_1 is at a time **T0** onto the bus. Nodes of all types save LT at the instance a Sync_1 message is received.

The Master then transmits its perception of GT at **T0**. The transmitted package Assert_2 is sent at time **T1**. **T1** is not at a constant time from **T0**. If the Assert_2 is delayed, **T1** will occur later.

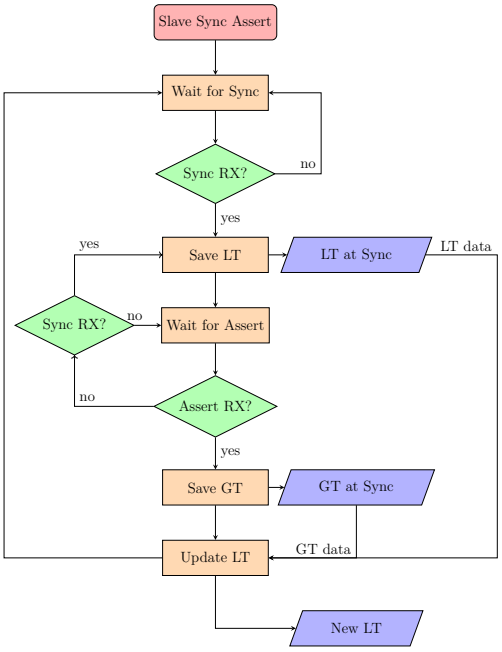
The slaves will save their LT at the time $\mathbf{T0} + \Delta$ upon receiving the Sync_1 message.

Δ is the delay from master to slave in the medium which could be considered a static error. In addition, the Δ is the reference to the receiving slaves tick-generator which might have a slightly different cadence to the master.

If a master does not send out a message in time, a slave will request the master to update the system with the current GT. Essentially, instead of the master sending Sync_1, the slave does. If the master node is still in the system, it will try to respond to the message by sending Assert_2, with the slave's Sync_1 as the reference point. If another Sync message, Sync_3, arrives before the master can send Assert_2, the saved LT in every node is updated to the time of the newest Sync message, Sync_3. The queued message Assert_2 containing the time at Sync_1 is scrapped by the master. A



(a) The master transmits its LT (GT) periodically or when requested externally.



(b) The slave-nodes updates their timekeeping with data from the master and its LT.

Figure 3.2: The Sync-assert protocol is depicted for nodes with Master or Slave role respectively.

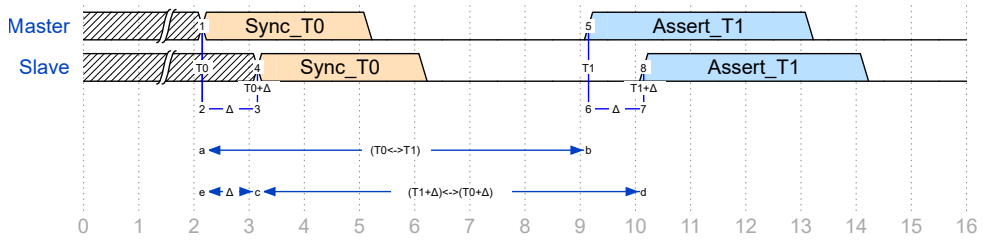


Figure 3.3: Sync-assert-process from instigated by Master at **T0** is perceived by Slave at $\mathbf{T0} + \Delta$ where Δ is the delay from Master-Slave transmission. The delay is greatly exaggerated in this figure from a real system.

new assert-message, Assert_4, containing the time at Sync_3 is put in queue.

3.5 System Fault-Handling

This section will outline how the system protocol should handle some potential faults. The LT-counter is not to be reset but kept running if the node moves from being synchronised to unsynchronised.

3.5.1 Master Node is Silent

If the master fails to respond to multiple $Sync_1$-requests by slaves in the system, the system will fall back to the arbitration process of starting up while retaining the same cadence and GT for each connected node. Since the GT is retained, a new master after the arbitration process will continue with the same GT and cadence.

This enables two separate networks, previously unconnected, to be connected with minimal impact on the temporal ordering of events.

3.5.2 Two Networks Reconnected After Separation

If two networks are reconnected from a previous common network, the master node transmitting first will be considered the new master of the combined network. The resigning master will compare the GT from its network and the new combined network. Unless the difference in the two GTs is significant the resigning master should resign so silently.

If the difference in GT is significant the nodes should note this to safeguard the casual ordering of events.

3.5.3 $Assert_2$ Message Received Without Preceding $Sync_1$

The node ignores the $Assert_2$-message and assumes it lost the preceding communication. Since a system has been detected (only a master can transmit $Assert_2$) the node assumes the role of slave and resets its timeout counters. If the node was the master of a system, but receive an $Assert_2$-message, it quietly goes to a slave-state instead.

3.6 Common Structures and Knowledge Required for Each Node

For there to be a synchronisation task, it is obvious that there have to be multiple numbers of nodes on a bus (the system) that are to agree on a common time perception. To enable the CDCAN-protocol to work, each node will need preexisting knowledge of several things, which are: 1) the LO's *(Local Oscillator's)* intended frequency; 2) the system's intended granule; 3) The size of the circular time-base; 4) the CDCAN identifier numbers and 5) the system's intended precision.

The LO frequency and system's granule is needed to adjust the counter values needed to generate the tick. A node with a faster LO will require the tick-generator a higher value to count to than a node with a slow LO. Inversely, a finer system granule will require the tick-generator to count to a lesser number than a coarser system granule. With this knowledge in each node, the system can handle having nodes with multiple different LO frequencies.

The size of the circular time-base is similarly of need to know basis, as this is the

number that is to be agreed upon within the system. If two nodes have different perceptions of the size of the time-base, they will not be able to agree on the time. A node with a smaller time-base size than intended will see numbers greater than what is achievable with its perception as nonsense, and a node with greater size than intended will see the system as periodically skipping in time.

The CDCAN identifier numbers are defining what IDs the CAN-messages that are used for the CDCAN algorithm is using. Any other identifier will automatically be disregarded for CDCAN communication. With this knowledge, a node can create ticks with intended periodicity and listen to CDCAN messages on the bus.

3.6.1 Time Counter and Time-Unit Considerations

The smallest time-unit available to the system is the system clock of 80 MHz and this is the basis for all following time-units. The smallest granule in the system available to synchronisation is the duration between two *ticks*. The actual length between two ticks as described in Section 2.2.2 is fairly arbitrary but all nodes in the same system must be capable of using the same granule size. A significantly slower MCU *(Microcontroller unit)* could limit the lower bound of granules due to slow processing speed. The nodes have to also be capable of working with the same epoch size epoch. The epoch is in turn depends on the duration between two ticks. The epoch should be designed for the longest relevant transient in a system that itself is highly system dependent.

The number of clock cycles per tick adjust the total time for a tick, if the tick length is designed to correspond to 80 000 cycles the result is 1 tick/ms, or 1000 tick/s. With 1 tick/ms and an epoch counter of 1 MiB (2^{20}bits) the total duration of the epoch would be approximately 17.5 min before the LT-counter would roll over.

In Section 2.2.3 the concept of sparse and dense time was introduced. Since the system designed is to function for a prolonged time and restrict its impact upon the communication bus, sparse time is the selected method where only one out of several ticks are used to synchronise the nodes. To decrease the likelihood of unnecessary master arbitration due to intermittent connection, multiple resynchronisation attempts must be allowed while still retaining a sufficiently low upstart transient of an uninitialised system. The initial design will hence use a synchronisation interval (sparse tick) of 10 000 ticks between synchronisation messages.

For 256 KiB (2^{16}bits) the total duration of the epoch would be approximately 1.1 min before the LT-counter would roll over. Similarly, for 16 MiB (2^{24}bits) the total duration of the epoch would be approximately 5 h before the LT-counter would roll over.

3.7 The Node-level Protocol

All nodes and types of nodes are to have the same basic behaviour which is described in this section.

Each node uses HDL-blocks to enable both local time-keeping and the ability to synchronise to an external source. This consists of a tick-generator and registers, state-machines, error-correcting-calculation, control-logic and I/O-interface to a CAN-bus.

3.7.1 Startup State

A node will from cold-start or reset listen to the bus for the presence of an existing synchronised system. The node listens to the bus for a period to evaluate if there exists an existing synchronisation, and if there is, or if another node in startup-state claims master, it will quietly join as a slave. If no such communication is detected the node will announce that it will take on the role of master. For there to be no accidental interruption of an existing system synchronisation from a newly started node, the period the newly started node should wait and listen on the bus should equal at least two expected duration of synchronisation messages. This duration is calculated from the preexisting knowledge of the LO frequency and the system granule, as stated in Section 3.6.

In the case of all nodes being in the startup phase, or in the case of a master being lost, eventually, a node will try and take the role of master. However, in a period between listening for an existing system and renegotiating a new master, nodes will actively reject any other node that tries to take on the role of master. This is to prohibit nodes that are too fast to take the role. After the rejecting period has passed the node tries to take the role of the master itself. If the other nodes reject it goes back to the starting state, called **S0**. If the node instead accepts it, they will quietly join in as slaves, and the winning node carries on as master. To avoid a situation where multiple nodes simultaneously try to take on the role of master, the arbitration phase of CAN is utilised. Each master-capable node has a unique CAN-ID associated with CDCAN, and the node with the lowest ID will win access to the bus, and also subsequently the master-arbitration.

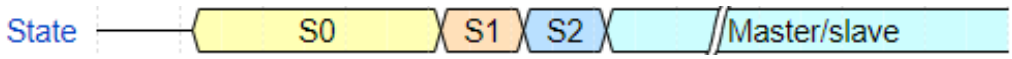


Figure 3.4: Overview of the start-up states in time in a node. At **S0**, a node quietly joins in as a slave in an existing system given that there is one active. At **S1**, a node rejects all other nodes' attempts to take the role of master. At **S2**, a node tries to take the role of master, but if unsuccessful takes on the role of slave. Duration of states not to scale.

The starting or initial procedure for a node can be seen in Figure 3.4 where **S0** is the initial passive state where the node listens for an active system. **S1** is the state where the nodes in an initial system count for a predetermined time to reject fault nodes running too fast or too slow. **S2**-state is where the node once again will accept other nodes assuming the master role and halfway through **S2** the node will itself try to assume the master role. The duration of **S2** limits which nodes are accepted. The start and end of **S2** can be considered the upper and lower bound as described in Figure 2.6. The total time between **S0** to the end of **S2** is the start-up transient

of an uninitialised system.

3.7.2 Running State

When a node is synchronised to the GT, it will do one out of two things. It is either the master node, which it will periodically send out synchronisation messages to update the slaves GT perception in accordance with the master's LT. Or it is a slave node, which it will listen to the master and adjust its LT to the GT given by the master.

A node with a master-role will periodically send out new synchronisation messages. The maximum periodicity of the synchronisation messages depends on the needed precision of the system and the worst-case drift. If multiple synchronisation messages are not sent out in time, the system will fall into a desynchronised state and need to resynchronise. The master is responsible to keep the system synchronised. The GT is time-kept in the master's LT-counter.

A slave node will passively listen to the bus for synchronisation messages and update accordingly. It will also continuously make adjustments internally to how many LO-cycles a tick consists of to lower the drift to the master's tick which is derived from the master's LO. In the case of a desynchronisation, nodes that have been synchronised and tweaked the tick-cadence will have a cadence similar to those other nodes in the system. This is done to decrease the jump needed in LT when synchronisation is reacquired.

Unless a fault occurs, nodes will stay in the running-state with their assigned role indefinitely.

3.7.3 Recovery State

In the event of faults, nodes must be able to handle them. If the master node for any reason stops sending synchronisation messages, then the slave nodes will have to renegotiate a master. For the individual node, this means that whenever it detects that the master is gone it will go back to the startup-phase.

A node may detect that an incoming synchronisation is either too fast or too slow to seem reasonable. But as the node can not know if it is itself that is too slow/fast, or if the master node is, it will send out a query to other slave nodes in the system to see if any of these agree that the synchronisation is too slow/fast. If no such response is received in some time, meaning no other node agrees, the node will go into a running-silent mode, and will not send another query. If a master node sees the query and an agreeing response, it knows that it has been demoted from the role of master. It will go back to the startup-phase but will be prevented to claim the role of master again.

3.8 Node Block Design

The protocol described in Section 3.7 is implemented in hardware, and this section will elaborate on different components of the node design by describing the blocks and their functions.

3.8.1 Local Time Counter

Essential for the timekeeping-ability is the LT-counter *(Local Time-counter)*. This counter is implemented as a simple clock-dividing counter with a predefined vector size as described in Section 3.6.1. While the time vector itself is constant, the clock-dividing factor (clock cycles per tick) is not. The dividing factor is controlled by a regulator. This counter should reflect and track the GT and is updated through the process described in Section 3.4 which the system master is responsible for.

3.8.2 Clock-per-Tick Regulator

To be able to compensate for inconsistencies between nodes' LOs *(Local Oscillators)*, the cadence of the LT-counter needs to be varied. This is done by changing the number of clock cycles from the LO it takes to make one increment of the LT-counter (tick). To update the value used, a regulator is used. Multiple regulator types can be used such as PID-regulators, Kalman filters and moving averages.

The regulator is fed with the current error in LT compared to the GT and returns how much the node should compensate in ticks this next period between synchronisations. The parameters in the regulator are updated once every synchronisation sequence as described in Section 3.4. The result of the regulator in turn will alter the amount of LO-cycles it will take for a tick, slowing or speeding up the cadence of the LT with it.

Because the value tick per period that the Clock-per-tick regulator gives is an absolute value of ticks, it is not directly usable by any other device. Intuitively one might be tempted to use division to get the percentage difference between the intended period and the measured, but this was decided against since division is a bothersome operation to do in hardware. Instead, a LUT *(Look-up table)* is indexed by looking at the result of the regulator and comparing that to predefined ranges of PID-results. The result from the LUT is the value *clocks per tick* that is then fed to the LT-counter.

3.8.3 Timeout

The timeout module ensures that the node is resynchronised at a set frequency. When a predetermined counter laps a timeout-pulse is generated. This pulse triggers a sync-message to be sent. A counter keeps track of the number of timeout pulses since the last successful assert message from the system master. If no update has been received, i.e there is no master in the system, for multiple timeouts, the timeout-block triggers the node to restart the master arbitration. The duration between

syncs can be adjusted here.

In addition, the timeout works with relative time between syncs, hence, this node ensuring that no errors occur from the circular properties of the fixed-length time vector in the LT-counter while comparing the elapsed time since the last synchronisation.

The functionality is identical for all nodes in a system. The master flag blocks some of the functions and the slaves have slightly slower pacing of the timeout-pulses.

3.8.4 Difference Calculator

The difference calculator compares the received GT timestamp in an Assert_2-message and the internal sampled LT at corresponding Sync_1-message. The difference is used to calculate a new LT-value to update the local LT-counter if needed. In addition, the node is responsible for handling the wrap-around of the counter. This is done by switching which value (GT or LT) is used as the first term and/or sign for the operation.

3.8.5 The CAN Bus Controller

Nodes in the CDCAN-network communicate over CAN, and as such, each node needs an interface controller. The interface controller uses a component of the Kvaser supplied IP, *KCAN*. The component, the BSP *(Bitstream Signal Processor)*, handles both receiving and sending of data, as well as error-handling and other CAN-protocol specifics. The choice of using the BSP instead of the entire IP-block, *KCAN*, supplied from Kvaser is due to signals being abstracted from the system with the entire KCAN-IP and not accessible by external HDL-blocks.

To enable the BSP additional signals are needed and as such two additional blocks are designed, the *RX Controller* and the *TX Controller*. The RX Controller interprets the incoming messages and ignores all messages on the CAN-bus not used for our protocol. A small subset of CAN-IDs are reserved for this purpose and the translation table is part of the RX Controller. The BSP itself is a "dumb" module and processes all packages regardless of Message-ID. The received messages are passed on to the relevant block downstream in the FPGA.

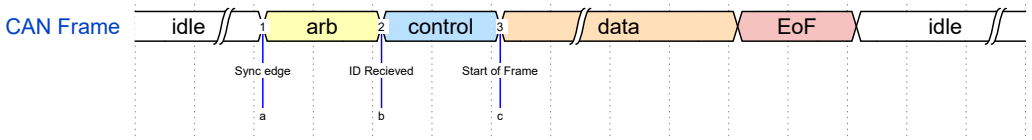


Figure 3.5: The CAN-frame consists of several parts, the Sync Edge is the first detectable change on the bus that a Frame is being transmitted on the bus. Other parts of the frame are subject to stuff-bits and the timestamp of the Start-of-Frame can change depending on the sample point of the BSP.

A critical part of the CDCAN-protocol is to accurately timestamp when a CAN-frame is received. In CAN there is a start-of-frame point, positioned in the frame where the frame-data starts as displayed in Figure 3.5. This point is not at a fixed

distance from the start and is subject to the BSPs re-timing attempts and actual sample point which can vary in accordance to the CAN-protocol. Instead, the RX-Controller pre-emptily saves the local counter data in LT at the sync edge from the CAN-bus which triggers when the BSP detects a change on the CAN-bus (from High to Low). If the ID matches a *Sync*-message or a *Assert*-message the relevant control signals are issued; else the saved timestamp is discarded. The Sync Flank is one of the abstracted signals in the KCAN-IP.

The TX Controller handles the configuration of the BSP such as bit rate, and other registers to enable the BSP to interact correctly with the CAN-bus. In addition, the outbound messages are collected in a buffer and prioritised to ensure the most important message is sent first rather than acting as a FIFO. In addition, the buffer in the TX Controller can be emptied without sending if the node changes role or if a conflict occurs in the system.

3.8.6 Timestamp

The timestamp module is triggered whenever a CAN-message or external signal is sent which requests the current time be sent through the CAN-interface. This is done to save the LT-counter content at the triggering and keeping it until transmitted. The block also keeps the time for the first request if the CAN-bus is busy and ensures that the data is transmitted when the bus is ready. The LT continues to run uninterrupted and multiple physical nodes can be triggered with the same external signal.

This module will be used for debugging and data collection, but is not necessary for the CDCAN-protocol itself.

3.8.7 Node Controller

The node needs to be able to handle multiple different situations and as such needs an overlying controller which steers the program flow in the intended directions. The controller is responsible for the internal state. In addition, the controller keeps track of the role the node has assumed and some signals will be limited or disabled by the master/slave-role of the node.

The controller has the same three primary states as the system described in Section 3.2: 1) startup-state; 2) running-state; and 3) fault-handling-state.

The startup-state is the initial state the controller starts a node in at boot. The master-arbitration process is an FSM *(Finite state machine)* run at start-up until the steady-state "Running" is reached.

The controller triggers tasks in other blocks from conditions and signals, from the CAN-bus and internal blocks. Nodes running with the master flag transmits Assert_2-messages after a Sync_1-message while the slave has the Assert-process disabled.

The fault-handling state handles a node that is rejected by the higher-level syn-

chronisation system to ensure that a faulty node does not disrupt the higher-level system.

3.9 Design of Tests

To verify our system can solve our described problems, tests needs to be designed.

3.9.1 Precision with Different Time-Scales

To show how different tick-lengths and synchronisation intervals affect the precision of the demonstrator system, the nodes are set to several different tick-lengths and synchronisation intervals.

3.9.2 System Functionality at High Busload

The system should be able to remain synchronised even at high bus-load. This test will use a setup from the precision tests but also load the bus as much as possible with higher-priority messages. The load is limited by the transmission rate of one Kvaser Leaf Light V2 USB-CAN interface [22].

3.9.3 Merging of Separate Systems

One design goal was to have the ability to merge two separate systems into one common system. This is done by having free-running systems which are subsequently connected.

4

Implementation

The design envisioned in Chapter 3 was implemented in hardware. Part of the project was to select a hardware platform and realise the design through HDL-blocks. The following sections will describe the hardware selected and any disparities from the design in the previous chapter.

4.1 Hardware Selection

The hardware platform selected for the demonstrator project was the Terasic DE0 Nano [9], depicted in Figure 4.1. This selection was based partly due to in-house knowledge but also due to additional hardware such as CAN-transceiver boards and IP-blocks for interfacing with these boards was available.

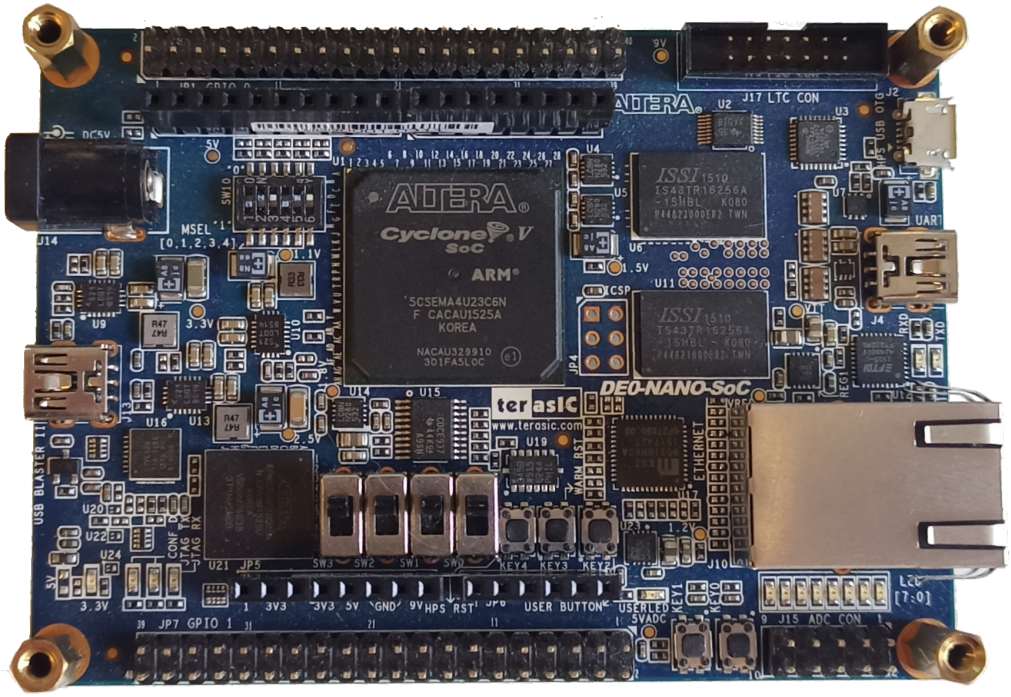


Figure 4.1: The Terasic DE0 Nano development board is to be connected to a board equipped with CAN-transceivers (not pictured).

The demonstrator system uses three of these boards connected to a CAN-bus. The demonstrator HDL-blocks are included in a bigger FPGA-project containing additional blocks for integration with a Linux system on the DE0-boards. The systems are network accessible and the FPGA-bitstream can be updated through this interface.

To control and read the bus from a lab computer, two USB-CAN interfaces of the model Kvaser Leaf Light V2 are used.

4.2 System Architecture

The system is subdivided into multiple blocks based on the envisioned design described in Chapter 3. However, some of these blocks and functions are not present or fully implemented in the demonstrator system. The missing or incomplete functions and components are the ones described in Section 3.3, 3.7.3 and 3.8.2. In the following sections, the differences from the initial vision will be described. Functions and blocks not mentioned in this section are implemented as described in Chapter 3.

4.2.1 Start-up Transient

The arbitration process described in Section 3.3 uses a predetermined duration of time to accept or reject nodes with LOs *(Local Oscillator)* which are significantly faster or slower.

This startup timer should be selected for a short time to limit the startup-transient, i.e., the time the system takes to resume or start the CDCAN-protocol. The demonstrator system reuses already available timekeeping blocks to decrease verification complexity of the system. The result is that the time used for synchronisation messages was reused and the start-up transient is directly proportional to the synchronisation interval. Hence systems with few synchronisation messages have a longer start-up transient.

The longer start-up transient was deemed an acceptable trade-off to save time on implementing an adaptable alternative. The time was instead spent on implementing more critical HDL-components and verifying these.

4.2.2 Fault Management

It was envisioned that in addition to the nodes being able to recover from a situation where the master disappears out of the system, the nodes should be able to determine if a master's cadence is too slow or too fast. The slaves should be able to detect this, revoke the current master's role and arbitrate a new master. This feature is not implemented, and the consequence is that once a node has gotten the role of master it will never lose this status unless it is unable to see sync-assert message chains on the bus. The system can therefore start to drift wildly if for any reason the master's clock becomes unstable. The case of a drifting LO was regarded unlikely and the

process of silencing or killing an existing master node can be performed through the CAN-connection manually.

4.2.3 Cadence Regulator

To increase the accuracy of the slave nodes and combat LO-drift, a self-adjusting regulator for the LT *(Local time)*-counter was envisioned. Different methods including a PID-regulator and a moving-average type regulator were unsuccessfully tried in the design. In the interest of time, the regulator was omitted entirely in favour of completing critical rework of other HDL-blocks. Due to the lack of the cadence regulator, given a node is not affected by external interference, slave nodes' LT will linearly drift away from GT according to the difference between the slave's clock and the master's. In other words, the worst precision as described in Equation 2.2 can never improve given a specific system with a set sync-interval and tick-length. Further, because the slaves' cadence can not be altered, in the case of GT being less than LT, nodes will turn their LT backwards to match GT.

Since the cadence regulator was not implemented the system lacks the awareness necessary to handle such an algorithm, as described in Section 2.2.4, which considers time in a circular fashion.

5

System Performance

The demonstrator system, shown in Figure 5.1, consists of three nodes connected to a CAN-bus which can be split to separate one of the nodes to an own network. Each side of this bus has a CAN-interface connected to a lab computer used to log the traffic and request the LT from each node. Because of a limited amount of platforms, we were able to construct only three nodes.

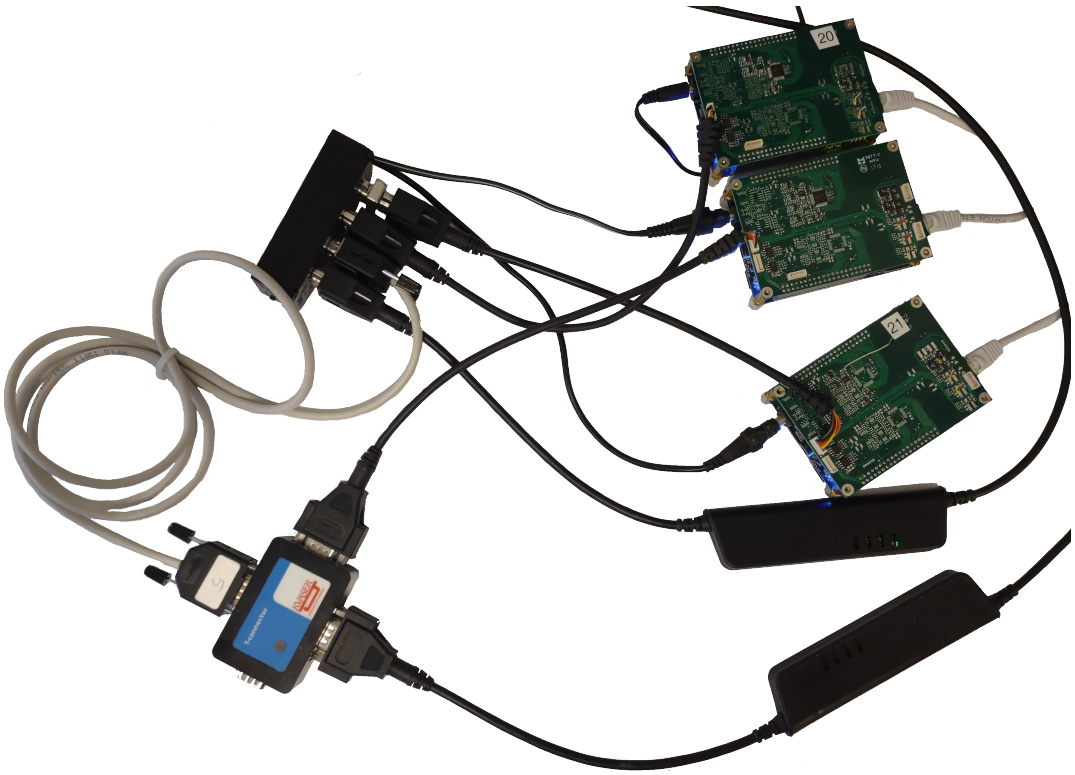
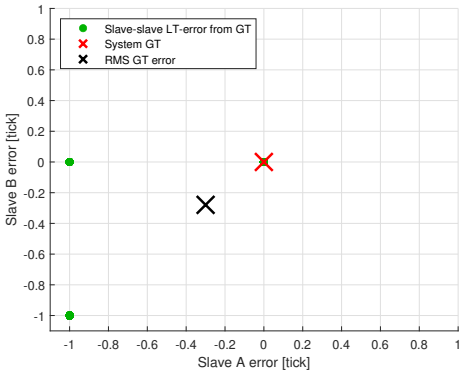


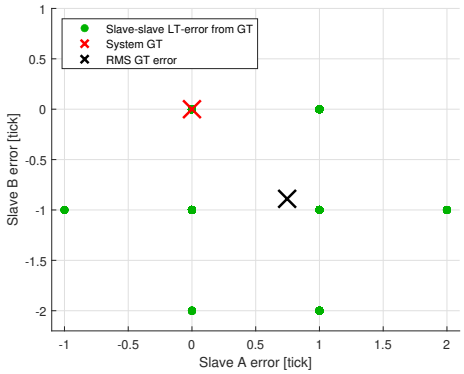
Figure 5.1: A picture of the test system. It consists of three DE0 Nano equipped with CAN-transceivers, two Kvaser Leaf Light, two DSUB connection blocks and an interconnecting wire. The wire may be re-/disconnected to shift between one and two systems.

5.1 Precision Tests

In this section, the results of the conducted Precision Tests are presented and is collated in Table 5.3. Each node will have the same settings, but because the DE0 Nano boards are very alike, they will not drift measurably in a reasonable test time-frame, and the worst-precision of different tests would be measured to be indistinguishable and close to zero. To show that the system works, the slaves will be configured in such a way that they drift from the master by having the slaves' counters being slightly slower. The LT of each node is sampled through the CAN-bus at a frequency of 5 Hz.



(a) A system of three nodes with identical hardware and settings. Local oscillators are 100 µs tick-length, 1 s/sync, no induced drift.



(b) Test of system with three nodes. The tick-length is 100 µs and the synchronisation interval is 1 s/sync, slaves has an induced drift of ±125ppm.

Figure 5.2: An ideal system (left) gives the CDCAN-protocol the best conditions, a system bound by the lowest time-unit (right) simulates a system in steady state but with different LO-frequencies and hints at what the demonstrator system can achieve. without alterations.

The data is then processed and the distance for each slave is measured to the master's perception of time, the system GT. The error is then plotted against each other. The RMS *(Root-mean-square)* for each slave is then used to place a systems error in the plot. This RMS value can then be compared to other configurations where a lower value denotes a better performing system. The RMS value can be considered the system-wide perception of GT, over a longer period, offset from the system GT kept by the system master.

Representing an ideal system with identical LOs and close to zero drift, Figure 5.2a (only inherent LO-drift) shows that the system performed admirably but not perfect. With no drift compensation slave A achieved 91 % accuracy, slave B achieved 92.2 % accuracy while the system as a whole had an accuracy of 91 %. The RMS-error vector was 0.4 tick.

A system with an induced drift of ±125ppm of the slave nodes performed significantly worse and is shown in Figure 5.2b. Slave A reached 56 % accuracy, slave B

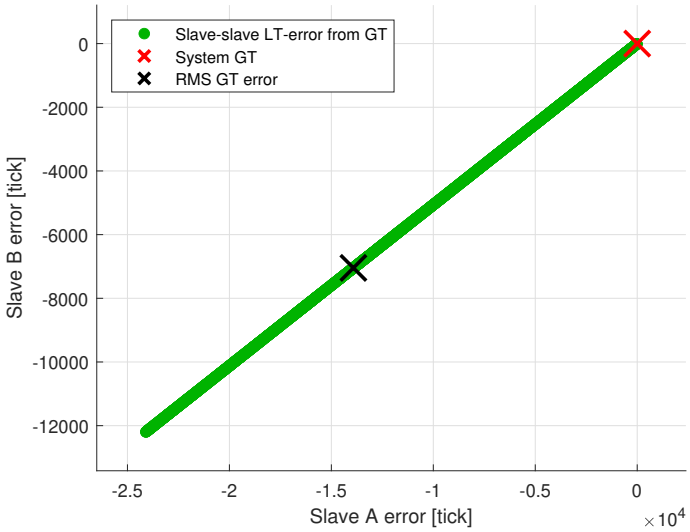


Figure 5.3: Test with 1 µs tick-length and a synchronisation interval of 1 s/sync. Induced drift in the slaves was 2.5 % and 1.25 % respectively.

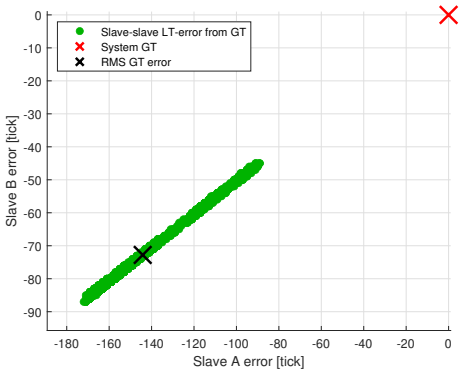
reached 34 % accuracy and the system as a whole only reached 19.2 % accuracy. The RMS-error vector was 1.16 tick, close to three times worse than the system with no induced drift.

The test with the greatest difference between tick-length and synchronisation interval had a 1 µs tick-length and 1 s/sync, as plotted in Figure 5.3 or 1 tick–to–1 000 000 tick per sync ratio. This resulted in large errors between slaves and the GT. A tick length of 1 µs results in a counter of only 80 clock cycles, the slaves work with one, respectively two ticks longer counters resulting in an induced drift of 2.5 % and 1.25 % respectively. The combined RMS-error vector of both slaves to the master's GT had a length of 15 600 ticks.

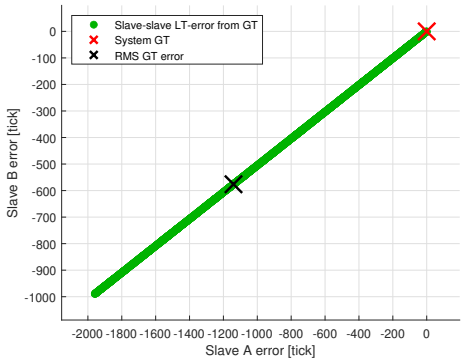
5.1.1 Performance with 10 µs Tick-Length

The induced drift in each slave is 2 % and 1 % respectively. The induced drift of the slaves can be observed in the axes where the X-axes has an error of close to twice that of the corresponding Y-axis.

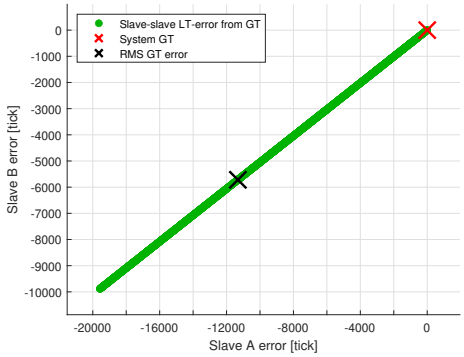
Three systems with tick-lengths of 10 µs but different synchronisation periods are tested and resulting errors are displayed in Figure 5.4.



(a) Test with 10 µs tick-length and a synchronisation interval of 100 ms/sync.



(b) Test with 10 µs tick-length and a synchronisation interval of 1 s/sync.



(c) Test with 10 µs tick-length and a synchronisation interval of 10 s/sync.

Figure 5.4: Three tests with 10 µs tick-interval and different synchronisation intervals. The error increases almost linearly to the increase in time between two synchronisation events. A higher synchronisation rate results in a higher precision within the system. All three tests included an induced drift in each slave with 2 % and 1 % respectively.

The black cross in each graph in Figure 5.4 is the combined RMS-error of the slaves in the system compared to the master's GT. The impact of the synchronisation interval on the precision can be observed as almost linear by comparing the magnitude of the X- and Y-axes between graphs. A tenfold increase in synchronisation frequency results in an almost tenfold decrease in worst error. In addition to the increase in graph-axes, the distance between the RMS-error vector and the master's GT increases.

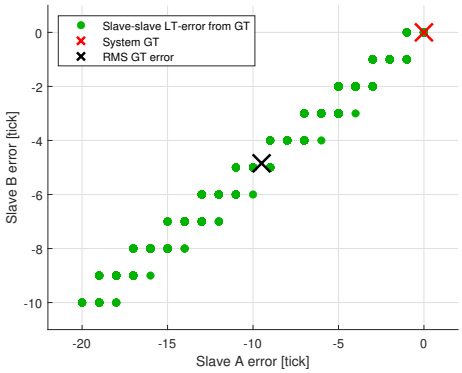
The worst precision of each system is calculated with Equation 2.2, as well as an RMS-error vector, and collated in Table 5.1. Increasing the synchronisation period degrades the performance of the system.

Table 5.1: Collated results from test conducted on 10 µs tick-length systems in Figure 5.4

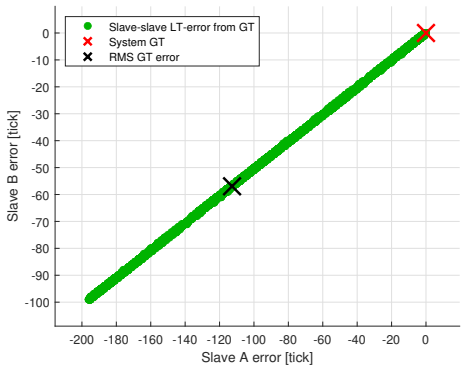
System configuration	Worst precision [ticks]	RMS-vector length [ticks]
10 µs/100 ms	172	161.5
10 µs/1 s	1959	1277.7
10 µs/10 s	19573	12684.6

5.1.2 Performance with 100 µs Tick-Length

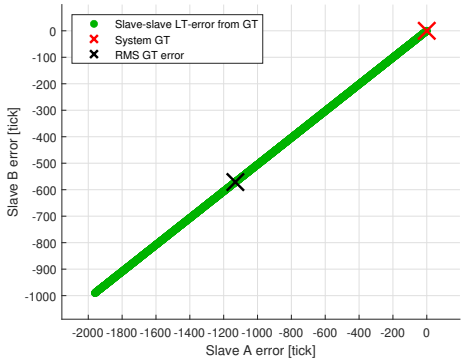
The same test is performed as in Section 5.1.1 but with an increase tick-length of 100 µs.



(a) Test with 100 µs tick-length and a synchronisation interval of 100 ms/sync. Induced drift in the slaves was, 2 % and 1 %.



(b) Test with 100 µs tick-length and a synchronisation interval of 1 s/sync. Induced drift in the slaves was, 2 % and 1 %.



(c) Test with 100 µs tick-length and a synchronisation interval of 10 s/sync. Induced drift in the slaves was, 2 % and 1 %.

Figure 5.5: Three tests with 100 µs tick-interval and different synchronisation intervals. The error increases almost linearly to the increase in time between two synchronisation events. A higher synchronisation rate results in a higher precision within the system. All three tests included an induced drift in each slave with 2 % and 1 % respectively.

Table 5.2: Collated results from test conducted on 100 µs tick-length systems in Figure 5.5

System configuration	Worst precision [ticks]	RMS-vector length [ticks]
100 µs/100 ms	20	10.66
100 µs/1 s	196	126.3
100 µs/10 s	1960	1266.9

The combined errors from the two slaves in each system compared to the systems master's GT is shown in Figure 5.5. The behaviour of the systems with 100 µs tick-length is the same as in the systems with the shorter 10 µs tick-length. The size of the errors is smaller by a factor of ten, corresponding to the increase in tick-length compared to the tests in Section 5.1.1.

The worst precision of each system is calculated with Equation 2.2, as well as an RMS-error vector, and collated in Table 5.2. Like the tests in Section 5.1.1, an increase in synchronisation interval degrades the accuracy of the system.

5.1.3 Synchronisation Summary

Here the results presented in Sections 5.1, 5.1.1 and 5.1.2 are summarised for ease of access and comparison.

To more easily visualise the relation between synchronisation interval and tick-lengths, in Figure 5.6 the RMS-error vector results from the tests are combined in one graph. Shorter resynchronisation intervals have a positive impact and decrease the error as well as longer tick-lengths. The best value corresponds to approximately 1 % error compared to the total length of a synchronisation interval.

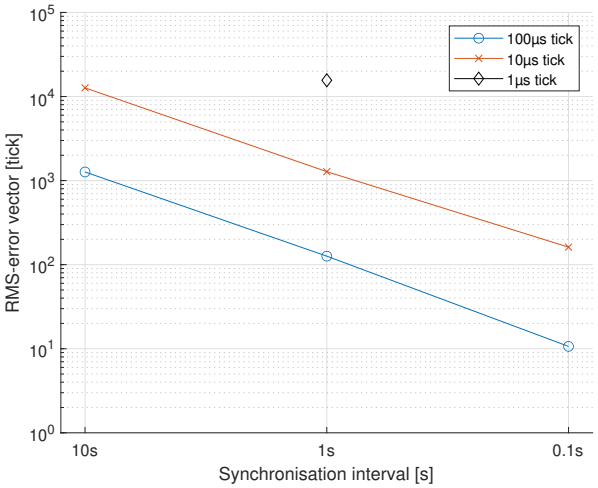


Figure 5.6: Graph combining RMS-error from each test, collated after tick-length.

The tests with the same ratio of ticks per sync, have similar performance and are not due to the absolute length of each tick as seen in Table 5.3. I.e. a system with a 100 µs

Table 5.3: Performance results of precision tests conducted in Section 5.1

Tick-Length [µs]	Sync-Interval [s]	Drift A [%]	Drift B [%]	Worst Precision [tick]	RMS-Error [tick]
100	1	0	0	2	0.403800
100	1	+0.0125	-0.0125	4	1.161465
1	1	-2.5	-1.25	24096	15600.325997
10	0.1	-2	-1	172	161.516255
10	1	-2	-1	1959	1277.742575
10	10	-2	-1	19573	12684.639162
100	0.1	-2	-1	20	10.663510
100	1	-2	-1	196	126.296176
100	10	-2	-1	1960	1266.934351

tick-length and a synchronisation period of 10 s has a ratio of 100000:1 ticks-per-sync. The same is true for a system with a 10 µs tick-length and a synchronisation period of 1 s. The resulting worst precision, approximately 1960 ticks, and RMS-error, approximately 1270 ticks, for both systems are similar.

5.2 Running System on a Busy CAN-Bus

CAN-messages can occupy the CAN-bus hence blocking new transmissions from be sent for a duration. The system is configured so both master and slave only act when the messages are transmitted. Hence the performance of a busy bus should not adversely impact the system.

A system with 1 µs tick-length and a synchronisation interval of 1 s/sync is tested on an almost fully loaded CAN-bus. The traffic on the CAN-bus is of higher priority than the messages used in the CDCAN-protocol. The smallest amount of drift possible to be induced is 1.25 % and 2.50 % respectively. This is due to integer limitations in the counters of the design.

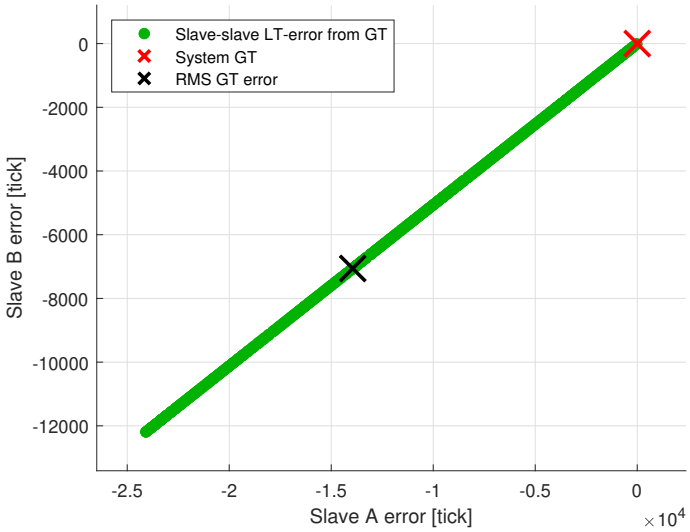


Figure 5.7: A system with 1 µs tick-length, 1 s/sync is tested on a CAN-bus heavily loaded with high-priority messages.

While the RMS-error vector is fairly large, 15 600 ticks, it is within the margin of error to the vector of the unloaded system, 15 602 ticks. This demonstrates the robustness of the devised method of time-stamping the CAN-traffic although not handling the drift of the oscillators in different nodes or reaching the desired accuracy.

5.3 Split of a Running System

A requirement for the project was that nodes disconnected from a running system were to arbitrate a new master and retain the same time perception. Hence a test was devised where an active and synchronised system was split into two. The nodes of the systems were continuously requested.

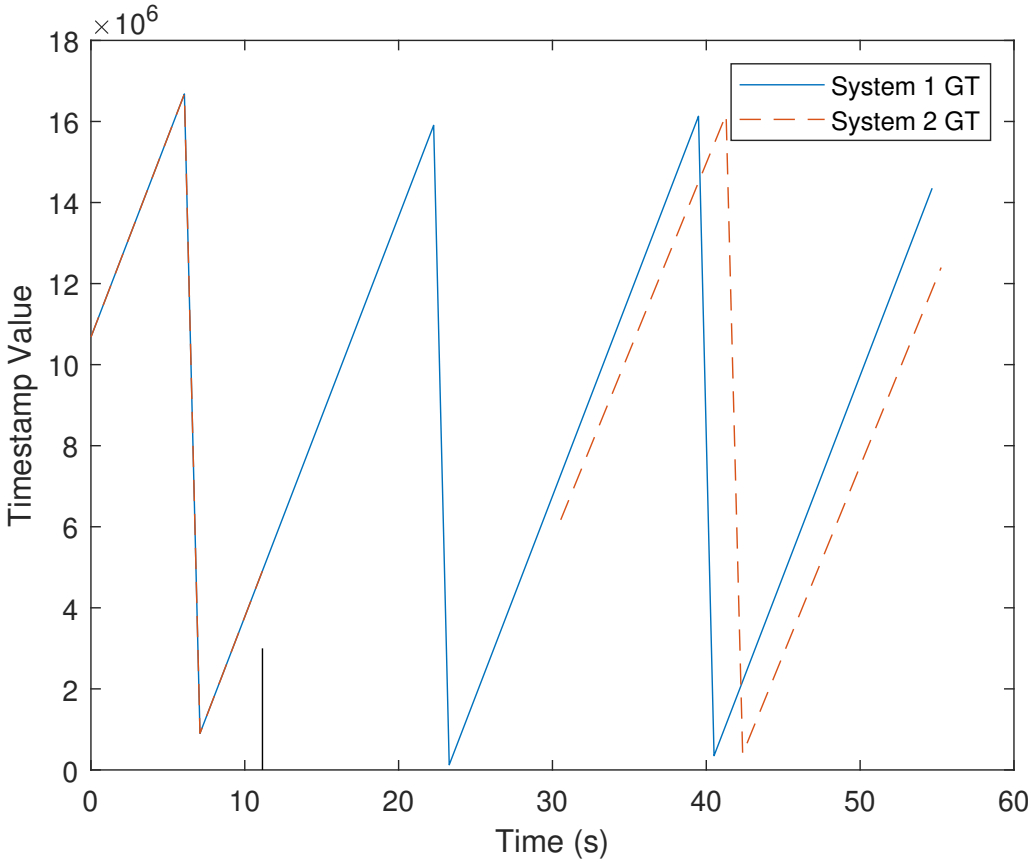


Figure 5.8: Example of a system being split into two separate systems. The black line at 11 seconds corresponds to the instant the system is split in two.

The result of the test is displayed in Figure 5.8 where the black line denotes the physical disconnection of the two system halves. The orange line is not shown during the period 11 s to 31 s since this half is arbitrating a new master. The time is retained while the cadence is not handled. If no drift were to exist or the drift would have been processed the two systems would continue with the same cadence and have the same perception without drifting apart.

5.4 Merging Systems

An additional requirement was for the system to be capable of handling systems being connected while already active with the CDCAN-protocol. Two free-running systems with separate time perceptions were connected after a time and the resynchronisation was observed.

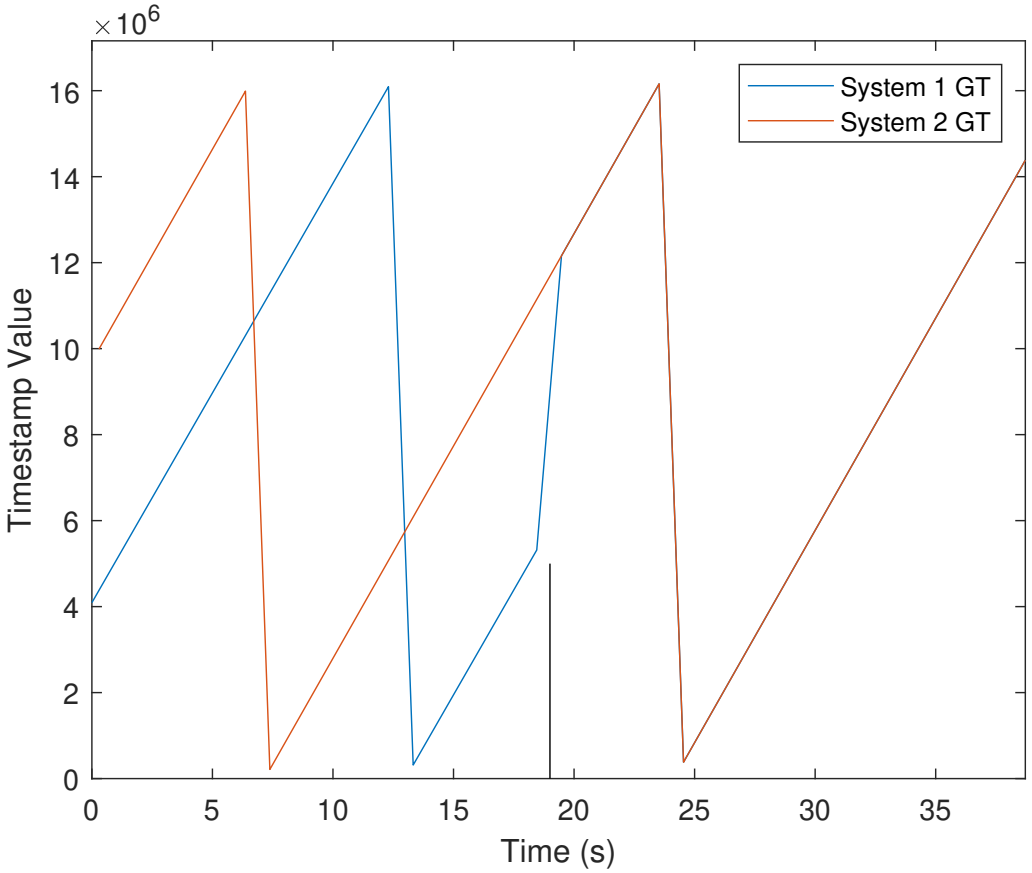


Figure 5.9: Example of two separate systems being connected together. Black line at 18 seconds correspond to when the two separate systems were connected together on a common bus.

The process of two systems reconnecting is plotted in Figure 5.9. At 18 s the two systems are reconnected, at 19 s the systems have combined and one of the masters present has retired. In this case, it is the master in System 1 denoted by the blue line. The sharply sloping blue line at the connection instance is only an artefact of the data processing.

6

Discussion and Future Work

In this chapter, the performance results will be analysed and the progression of the design process discussed. Due to the poor performance of the system some factors contributing to this will be discussed as well as alternative solutions to the design.

6.1 Analysis of the Performance Results

The performance result of the demonstrator system is somewhat mute due to the cadence regulator was not implemented and as such the system is unable to handle drift in LOs or nodes with different LO-frequencies. The system does handle the functional requirements outset in Section 1.1 admirably but the precision is extremely poor and no handling of drifting has been implemented.

With the nodes configured with no induced drift, the synchronisation was maintained for prolonged periods without synchronisation which is down to the precision of the LOs. The accuracy of 91 % could be due to the tick-generators between nodes not being phase-aligned and/or the effect of time-of-flight in the medium. Additionally, paths in the FPGA conducting the information handling might be different between master and slave nodes and as such be subject to slightly different path-delays.

The conducted test with ±125ppm induced drift was expected to be close to what could be achieved with a demonstrator system built out of nodes with different speed LOs, and equipped with cadence regulators. Cadence regulators should make it so that the error would be close, but always behind the master in order to always retain the ability to temporally order events. However, this test showed an unexpectedly poor result. Because the low drift, a very low error was expected. Indeed, even in a system with no induced drift at all there were still errors. This could be because a couple of things, like logic delays, implementation oversights or simply being unlucky. What the exact reason is should be investigated and understood before a real effort in developing the cadence regulator is made.

The tests with induced drift of more than 1 % are most likely beyond the scope of a system using CAN as the medium. As stated in [3] CAN supports LOs with fairly low precision. This does not necessarily, mean that poor precision LOs are used and as such, the induced drift is a somewhat unlikely scenario to encounter.

6.2 Unwise Decisions and World Events

This thesis project failed to meet its design goals. Much complexity arose from the integration and testing of our design. This, in turn, delayed the development and testing of features. Furthermore, other events affected the workflow negatively. In this section, we will in further detail explain decisions and events that negatively impacted the project.

6.2.1 Unnecessary Complex Platform

A big part of the project was to evaluate different potential hardware solutions and implement one of these in a platform. The platform selected was one with an already present HDL-design which had to be integrated with the CDCAN-design. This integration work was greatly underestimated and continued to plague the project with a significant amount of extra work. While the aspiration of using an already available platform for verification was good, the actual implementation did not end up using the tools in the platform. Instead, the much larger project integrating the CDCAN-design greatly increased the time for synthesis and integration without adding any real benefits. In addition, the debugging of external connections with multiple layers of projects was an unanticipated challenge.

In hindsight, it would have been much better to work on *bare metal* for the CDCAN-project due to the amount of time spent on the tools. Had there been less friction the additional tools might have been leveraged for testing and verification but would have required driver programming and creation of multiple software tools. The required development time for these tools was not correctly anticipated in the project plan.

6.2.2 Imprecise Data Collection

The use of the Linux platform would have enabled data collection and preferably the data collection would have been conducted with a known reference time, such as a GPS. Due to the tools not being completed the data collection was conducted through messages on the CAN-bus used by the demonstrator system. The use of the CAN-bus itself should not impact the demonstrator system adversely but is an additional entity that can affect the results negatively.

6.2.3 Computational Complexity of HDL-Simulation

One major hurdle in regards to time was the simulation-testing of the complete HDL-design. Individual HDL-blocks were not time-consuming to simulate, but once the entire system was assembled the simulation environment could not handle the tests promptly. The biggest culprit was the BSP-module. For this component to work properly it needs to be configured correctly. As setting up the BSP is non-trivial, we avoided changing working settings. These settings need, relative to our design, a long real-time transient to perform the CAN-protocol communication. This substantially slowed down all simulation-verification of the whole design of a

single node. In the cases of a system consisting of multiple nodes, simulation times further increased with each added node. For full-scale tests of functionality, the needed simulation times ended up being hours. Instead, resynthesising and loading the test-designs to actual hardware for each test ended up being faster, but with added difficulty in debugging.

6.2.4 Solving Time-Synchronisation and Hot-Swap Simultaneously

We tried to solve both the hot-swap aspect and the robust requirements at the same time as we designed a time-synchronisation protocol. It would have been easier to implement an already proven time-synchronisation protocol and separately handle the master assignment and hot-swap capabilities. This might seem like an obvious solution, at the time it was not.

6.2.5 Global Pandemic

This project was affected by the Covid-19 pandemic negatively. In the early stages of the pandemic, efforts were made to try and work remotely. This, however, led to a breakdown in communication and coordination. Many tasks were either completed twice separately or to two incompatible results. Proper readjustment to the different working styles was only achieved at the very end of the original plan, and this tardiness led to too much lost time and subsequently failure to achieve results in accordance of the original plan. Contingencies and plans on how to coordinate remotely should perhaps have been made beforehand, but we had not foreseen the need for this.

6.2.6 Work Intermission

After the initial time plan was missed, we both became busy with other things. It was first at the start of summer 2021 that there was time to restart the progress of this thesis. However, after about a year of inactivity, we found it hard to get back into understanding our design. This, combined with the numerous odd bugs and workarounds that resulted out of the previously uncoordinated work made it very hard to add and fix required features. A decision was made to keep the parts that worked as intended and remake the ones that did not. Much time was dedicated to rewriting a lot of the modules from the ground up. The decision resulted in modules that worked as intended, but many of the intended features had to be scrapped because of lack of time.

We believe that this decision in the end saved time even though a lot of work was essentially done again. If we instead would have tried to rework and regain an understanding of the original design we might still not have a working design at all.

6.3 Alterations and Omissions

Of the functions listed in Section 1.1.1, this section will discuss the features that were cut or altered, and why.

6.3.1 Assignment of Master-role to Most Suitable Node

The system was envisioned to ignore false masters or handle conflicts between two or more masters in one system with the least effect on GT for the system as a whole. This feature has been omitted in the implementation and instead the current master will relinquish the role if a new master is detected in the system. This means in practice that two independent systems which get connected will lose one master regardless of time. In turn, the system does not guarantee that the time is turned "forward" to ensure coherency in the casual ordering of events at the HDL-level. The lacking handling of master conflicts severely impacts the system's robustness. Due to the system being tested is in a known environment this does not pose a danger to the results collection.

6.3.2 Scalable System

One design goal that was not achieved was to have the developed system infinitely scalable. An arbitrary amount of nodes that all run the CDCAN-protocol were to be able to all be plugged into the same bus and synchronise. This was not achieved and is likely impossible to achieve. It is possible to add an infinite amount of slave-limited nodes, as these nodes usually do not send any information, but even when they do, they would not cause bit errors on the bus. But if there were to be an arbitrary amount of master-capable nodes the sync-assert messages would need to share a common identifier. In the situation of any two nodes simultaneously trying to send a CDCAN-frame they would not be able to distinguish if there is another node on the bus at the same time. As both nodes use the same identifier, both nodes would believe they won the bus after CAN-arbitration. This would lead to error-frames and retransmissions if any of the data-bits do not correlate, or if the transmissions simply drift apart. Because of how the CAN-protocol works, both nodes would immediately again try to access the bus, resulting in another error-frame. A random-delay back-off algorithm like in Ethernet would have to be implemented for CDCAN-frames to better avoid this situation. However, it is not a solution, as a bus with a sufficient amount of nodes would still interfere more than not, thus filling the bus with error-frames. This would result in the LT of each node never being synchronised to a GT, as one would never be agreed upon. Further, it would be denying any lower priority traffic ever getting onto the bus and reduce the possible throughput for higher priority transmissions.

The solution is that each node has a unique and system-wide reserved message-ID for CDCAN-traffic. This has the repercussion that only a limited amount of nodes that may be the master, and each added master-able node will reserve one additional message-ID that may not be used for any other kind of message in the entire system.

6.3.3 Retain Temporal Ordering

A design goal was to retain temporal ordering at all times, meaning that the LT in each node was not to ever be turned backwards unless going from an idle state to a slave state. This is not implemented, as the needed HDL-block to tune the cadence of the nodes were not finished. Instead, the LT is simply set to the new GT-value. The consequence is that if a node is faster than the master, its LT can run ahead of GT. Once synchronisation is done, the LT of this node will have to be corrected backwards to be in line with GT. If two events were to be timestamped just before and just after the synchronisation, it would appear that the order they occurred is reversed to the actual order.

6.3.4 Handle Unsuitable Masters

Because the cadence-tuning HDL-block was not completed in time, the condition-checks envisioned for disqualifying unsuitable masters were unusable. A suitable replacement algorithm was not developed.

6.3.5 Limitations to only one CAN-bus

Throughout the project, only a single bus has been considered. Conceivably multiple separate CAN-busses can be interconnected through bridges. From the topology viewpoint, this compound system constitutes a tree topology. This kind of topology was not considered to be a part of the CDCAN-protocol. Enabling support for bridging multiple buses necessitate rework of the algorithms, state machines and a new perception of the problem at hand.

6.4 Alternative Solutions

This section will discuss alternative solutions and possible improvements which differs from the system design described in Chapter 3 and the implemented system described in Chapter 4.

6.4.1 Improve Time-of-Flight Knowledge

The time-of-flight delays in the medium, Δ in Figure 3.3, is not determinable with only the master transmitting sync-messages. The delay can be different between different nodes in a system and this characteristic can be used by a slave-node to calculate the distance to the master without not introducing additional message types.

In Section 2.4.2 the synchronisation protocol for PTP was described. A similar methodology can be implemented in CDCAN albeit at the cost of busload and worse scaling in big systems. Perhaps this method needs to be limited to the most time-critical applications.

This implementation in CDCAN is envisioned to work similarly as in PTP while reusing the same message-IDs and not introducing additional message types to the CDCAN-protocol already designed. If a slave node transmits a sync-message, the delay between its LT and the master's GT at the message *Sync Flank* will be skewed by the medium delay Δ but in the opposite direction to transmissions from the master node. By noting this delay, Δ, the slave node can subsequently compensate for the medium delay in any future synchronisation.

Even though the delay in a CAN bus is quite stable over time, because of the design goal of hot-plugging being supported, the bus delay can change over time. To ensure the correctness of the medium delay-compensation, periodic tests need to be performed. Furthermore, because of the reuse of the sync-message-id in each node some uncertainty in the measurements could arise in the situation where two or more slave nodes simultaneously attempt to find the medium delay.

6.4.2 Combine Sync-Assert in One Message

In the demonstrator system, the Sync Flank of a Sync-message (a frame with ID and no data) locks the GT-value in the master. This GT-value is transmitted in the data field of a consecutive Assert-message. If instead the Assert data is transmitted in a Sync-message, the number of packages on the bus can be halved compared to our solution. Slaves can request an update by transmitting the same message but with no data-payload.

This solution was not used to simplify the logic needed for the master to know when it needs to send the assert-message. In the current system, whenever a master node receives a sync-message it unconditionally goes to send the assert-message as soon as possible. Sync-messages are sent by each node whenever it reaches a soft-timeout, which includes the master node itself.

6.4.3 Clock Recycling and Phase Aligning

In the protocol White Rabbit, described in Section 2.4.3 the accuracy of the time-keeping is increased by aligning the phases of the clocks themselves in accordance to the master clock. Conceivably the same could be done on a CAN-bus, but only with new and dedicated hardware. Because of the architecture of the BSP-module and, the exact width of a transmitted bit on the bus or the consistent timing of the flanks can not be guaranteed. The analogue nature of the bus could also induce uncertainty in the rise- and fall-times of the signals on the bus. Because of these factors, it would be hard to get consistent results out of clock recycling attempts with current hardware. Instead, additions to the BSP module functionality would be needed to enable this. Strict constraints on the bus itself would also need to be made to further assure accuracy.

6.4.4 Multiple Levels of Timekeeping Units

To increase the total epoch length different counters can be used to denote macro/microticks, where one *macrotick* is N-number of microticks. This would increase the complexity of our demonstrator system without significantly enhancing the current timekeeping capability. It was deemed superfluous for the demonstrator system but might be a worthwhile endeavour for a system that needs timekeeping for a significantly longer time than a single counter or message can contain.

6.4.5 Master Suitability Evaluation

Each node capable of taking on the role of master will do so if able, and given that two nodes start at the same time, the node which has the fastest clock and lowest associated message-ID will take on the role after the negotiation phase. This is beneficial, as the faster clock's cadence will ensure that no node will have to change their LT backwards, potentially misrepresenting the order in which events happened. However, in the case of a staggered start, the node which happened to be turned on first will be much more likely to take on the master role, regardless of that node's relative clock suitability. This may result in a situation where nodes have to change their LTs backwards to stay aligned to GT.

One potential fix to the current system is that a slave node that detect that its LT is ahead of GT during the system-running phase described in Section 3.4 instead takes on the role as master by sending an assert-message. This informs the current master that new master in the system exist, and subsequently back off and itself become subservient to the new master. Care need to be taken in the implementation, however, to avoid a reversed *hot potato game*-situation, where the master role is frequently taken over by another node. This can require a certain threshold between the slave LT and GT, or that nodes are prevented from retaking the role of master within certain amount after being deposed.

6.5 Future Work

The previous sections mentioned several reasons why the system did not perform satisfactorily. The following list is how we recommend conducting and prioritise future work to conclude the project and finish the CDCAN-protocol.

The most pressing issue is that identical nodes are drifting and the causes need to be pinpointed and fixed. If nodes using the same LO-frequencies are unable to keep clocks synchronised, it is hard to see that the system with different LO-frequencies would be feasible to synchronise with any good accuracy.

After fixing the first issue, the missing HDL-module responsible for cadence regulation should be implemented. This module is needed to handle the drift of nodes and enables nodes constructed of different hardware to interact.

The startup procedure is another function that should be modified. With a separate

timer, the startup transient can be made shorter since the transient in the demonstrator system is directly related to the length of the synchronisation interval. This would improve the speed of debugging and verification further.

Some other things to investigate are delays in the transmission medium and the FPGA pipelines. Delays could originate from these sources and high precision is not achievable unless these delays are known.

Furthermore, the unsatisfactory handling of master node-conflicts needs to be evaluated and a system to handle this. Currently, the fastest master wins the system but there is no evaluation if the fastest master is the most suitable node. Hence, a robust system is not within reach until an arbitration or system for handling conflicts is devised.

Finally, if solving the previous tasks are not enough to reach a satisfactory system, the clock synchronisation protocol should be reworked with a known and functional protocol clock synchronisation protocol. The master node assignment should be separated from the synchronisation protocol.

6.6 Ethical Obligation and Environmental Impact

The CDCAN-protocol being compatible with already existing CAN-systems is good from the standpoint that implementing CDCAN is possible without having to redesign entire systems. This lower the material impact of implementing the CDCAN-protocol compared to a completely new system that requires all nodes to be compatible. Nodes communicating using the CDCAN-protocol does however use HDL-blocks which requires hardware compatibility. If the protocol could be lifted to a higher level, such as software, this would be beneficial from an implementation standpoint.

The CDCAN-protocol can also be modified to use several levels of ticks and therefore not use unnecessary traffic on the bus. Power draw of a small network might seem inconsequential, but summed up over many different networks, it does significantly contribute. To keep the traffic to a minimum is therefore not only good from a functional standpoint but also an environmental standpoint.

The use of ASICs or CAN-transceivers specifically built with the CDCAN-protocol in mind might be worthwhile to minimise the power draw of each node while synchronising the system. However, these benefits have to be measured against the environmental impact of manufacturing new circuits to replace the old ones. Systems that use FPGA-chips instead of dedicated ASICs for the transceiver could conceivably be updated to accommodate CDCAN without further manufacturing.

The CDCAN-protocol is only tested during laboratory conditions and even if the performance problems are fixed, it is not ready for use in any safety-critical application, such as medical applications where personal harm may occur if the system fails. The design of a fault-tolerant system requires significant testing and verifica-

tion before deployment, which is yet to be done with the CDCAN-protocol.

7

Conclusion

This project aimed to provide a demonstrator system enabling accurate time synchronisation over a CAN-interface while not adversely affect any other system using the same CAN-network. While the entire design was not implemented fully in hardware the functional behaviour was implemented. The hot-swap capability was demonstrated and robustness of the master assignment reached a satisfactory level. The negative impact of deploying CDCAN in a CAN-system is minimal due to the low bus usage.

The task of designing an algorithm capable of retaining temporal ordering of events in a circular fashion, when counter for timekeeping wraps around, was not finished but during testing. This does not seem to have had a significant negative impact. Perhaps this would have been more important if the tick-lengths approach the limit of what the CAN-bus and LOs can support.

The precision of the demonstrator system is extremely poor and it is unclear if it is down to the missing modules or some other issue at a more fundamental level.

If additional time and resources could be devoted to the task, the authors strongly believe that a robust time-synchronisation protocol with high accuracy is achievable on CAN while meeting the functional requirements outlined in Section 1.1. If the best direction is to continue developing the current design or rework the system from the ground up is uncertain. Additional investigations of where the errors originate from should be conducted before deciding direction.

Bibliography

[1] P. Jansweijer, H. Peek, and E. de Wolf, "White rabbit: Sub-nanosecond timing over ethernet," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, vol. 725, pp. 187–190, 2013, vLVvT 11, Erlangen, Germany, 12 - 14 October, 2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016890021201652X

[2] M. Han, H. Guo, and P. Crossley, "IEEE 1588 time synchronisation performance for IEC 61850 transmission substations." *International Journal of Electrical Power and Energy Systems*, vol. 107, pp. 264 – 272, 2019.

[3] Robert Bosch Gmbh, "Can specification version 2.0," 1991.

[4] M. Peller and J. Berwanger, "Road vehicle controller area network (CAN)-part 4: Time-triggered communication," *ISO IS*, vol. 11898, 2004.

[5] L. Almeida, P. Pedreiras, and J. A. G. Fonseca, "The FTT-CAN protocol: Why and how," *IEEE transactions on industrial electronics*, vol. 49, no. 6, pp. 1189–1201, 2002.

[6] R. Kurachi, H. Takada, N. Adachi, H. Ueda, and Y. Miyashita, "DDCAN: Delay-time deliverable can network," in *2019 IEEE 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, July 2019, pp. 36–41.

[7] Calnex Solution Ltd., "Implementing IEEE 1588v2 for use in the mobile backhaul," Technical Brief, 2009, accessed: 2020-01-02.

[8] Kvaser AB, "Kvaser CAN controller," 2019, unpublished.

[9] Terasic Technologies, "DE0 Nano SoC User manual," 2015, accessed: 2020-02-18.

[10] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed fault-tolerant real-time systems: the mars approach," *IEEE Micro*, vol. 9, no. 1, pp. 25–40, Feb 1989.

[11] H. Kopetz, R. Hexel, A. Krüger, D. Millinger, and A. Schedl, "A

synchronization strategy for a TTP/C controller," in *International Congress & Exposition*. SAE International, feb 1996. [Online]. Available: https://doi.org/10.4271/960120

[12] N. Chauhan, *Principles of Operating Systems*. Oxford University Press, 2014.

[13] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, p. 558–565, Jul. 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[14] J. C. Eidson, *Measurement, Control, and Communication Using IEEE 1588*, ser. Advances in Industrial Control. Springer London, 2006.

[15] M. Kihara, S. Ono, and P. Eskelinen, *Digital clocks for synchronization and communications*. London, United Kingdom, Artech House Publishers, 2003.

[16] D. Paret, *FlexRay and Its Applications : Real Time Multiplexed Network*. New York, John Wiley & Sons Incorporated, 2012.

[17] H. Kopetz, *Real-time systems: Design principles for distributed embedded applications*. Boston: Kluwer Academic Publishers, 1997.

[18] J. Martin, J. Burbank, W. Kasch, and P. D. L. Mills, "Network Time Protocol Version 4: Protocol and Algorithms Specification," RFC 5905, Jun. 2010. [Online]. Available: https://rfc-editor.org/rfc/rfc5905.txt

[19] R. Exel, "Clock synchronization in IEEE 802.11 wireless LANs using physical layer timestamps," in *2012 IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication Proceedings*, Sep. 2012, pp. 1–6.

[20] *Road vehicles — Controller area network (CAN) — Part 1: Data link layer and physical signalling*, ISO Standard 11898-1, 2003.

[21] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014, pp. 305–319. [Online]. Available: https://raft.github.io/raft.pdf

[22] Kvaser AB, "Kvaser leaf light v2 user's guide."