EtherCAT-based Platform for Distributed Control in High-Performance Industrial Applications

Dalimir Orfanus, Reidar Indergaard, Gunnar Prytz, Tormod Wien
ABB Corporate Research
Bergerveien 12, 1397
Billingstad, Norway
{dalimir.orfanus,reidar.indergaard,gunnar.prytz,tormod.wien}@no.abb.com

Abstract

Industrial Ethernet is gradually becoming pervasive in the automation field. EtherCAT is one such technology, combining high real-time performance with a rich functionality set. Besides that, EtherCAT offers short cycle times, accurate time synchronization, redundancy and high data throughput. Most EtherCAT applications are in the 500-1000μs domain. There are only few EtherCAT master implementations in the sub-100 or sub-50μs range for high-performance applications. This paper presents EtherCAT based distributed controller which is capable of less than 30μs cycle times and was implemented on P2020 processor. Total HW and SW overhead per cycle is 9.6μs including jitter of 5μs. Achieved preciseness of the distributed time is below 20ns. The stack was designed to be modular and also reused on different platforms.

1. Introduction

Use of Ethernet for industrial applications and factory automation is steadily gaining more and more popularity. There are several reasons behind it. Just to name few of them: (i) low cost of infrastructure components (network interface cards, switches, routers) due to its pervasiveness in the commercial area; (ii) variable topologies allowing various distributed control schemes, (iii) large number of silicon vendors (thus avoiding vendor-locking); and most importantly (iv) much higher bandwidth and speed than traditional fieldbus solutions. Ethernet has a very attractive price-performance ratio when compared to the most fieldbus solutions. However, Ethernet is basically a communication medium but not the complete communication solution. In the Open Standard Interconnection (OSI) model it is located at the lowest two layers: the physical layer and partially the data link layer.

The family of *Industrial Ethernet* (IE) protocols has in recent years evolved in to a large number of solutions and standards. Two of the most prominent representatives of this group are EtherCAT [3] and Profinet [5] where the earlier is believed to be the highest performing [6].

EtherCAT is widely used in the automation market and there exist a large number of master stack implementations. However, only a very few of those are designed to reach cycle times significantly below 100μs. In automation there are number of applications where there is a need for very fast control loops with data update rates below 100μs. Two examples are high-end motion control applications [9] and certain power electronics applications [8, 10]. Cycle times in such applications may go down to 30μs or even below.

In EtherCAT networks the devices (slaves) can be very accurately synchronized but the master may not be such accurately synchronized with the devices. This is mainly due to the fact that the EtherCAT slaves have special hardware support for time synchronization, something the master hardware normally does not have. However, this can be for example solved by integrating an EtherCAT slave into the master device (two integrated slaves are needed in the redundant ring network) [7].

It was a further objective of the work to test the hypothesis whether it is possible to implement the EtherCAT stack with sub-30μs cycle times and at the same time could easily be moved to a different hardware platform and operating system. This is particularly challenging for very high-end implementations as the push for generality means that platform specific customizing should be avoided as much as possible. Generally speaking, making a system more generic and platform independent typically introduces some overhead which in turn cause additional latencies and jitter. The goal of this work was also to investigate whether a generic design could reach the desired performance levels.

The goal is to use EtherCAT in variety of embedded system with different requirements. To have a very specific implementation, tightly connected with the platform, is not a very feasible and maintainable solution. Therefore, we had to address another challenge to have a common stack for different platforms and at the same time to provide high performance in case of the devised distributed controller. Additionally, we would like to also

978-1-4799-0864-6/13/$31.00 ©2013 IEEE

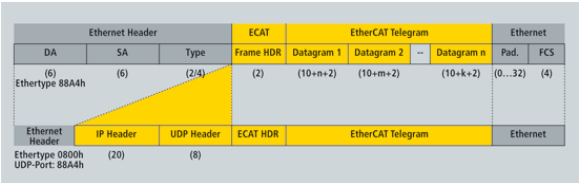

Figure 1. EtherCAT protocol format [3]

support different implementations of the stack with respect to the timing requirements, footprint, concurrency, etc. For example, in some embedded applications the footprint in terms of memory usage may be more critical than latency or jitter of the time synchronization and vice versa. The objective was thus to achieve functional modularity and platform independence while keeping the cycle times below 30μs.

The paper is organized as follows: Section 2 gives a short introduction to EtherCAT whereas Section 3 summarizes the relevant previous work. Section 4 outlines the design of the solution and Section 5 describes the implementation in detail. Finally, Section 6 presents some experimental results from the tests that have been done before the conclusion is found in Section 7.

2. EtherCAT: Brief overview

EtherCAT [3] is Ethernet based industrial real-time protocol. It has a ring topology consisting of a EtherCAT Master (ECM) and up to 65535 EtherCAT Slaves (ECSs). Physically, the master is connected with ECSs into a daisy-chain topology. Topology can be arranged either into a line or a full ring. The ring arrangement provides a redundant path for increased safety and availability. From the application perspective, only the master is allowed to create and send a frame. Slaves can only read/write into passing frames at specified position based on the commands contained in the currently passing frame. The EtherCAT frame does not stop when passing through the slave, instead it gets slowed down (being buffered, typically from 4-16 bytes) and the read/write is done on the fly. The latency per slave is typically not greater than 1μs.

From the protocol perspective, EtherCAT telegrams can be used at OSI layers 2, 3, or 4. Format of the EtherCAT's frame (see Figure 1) always consists from the EtherCAT header and one or more *telegrams*. Currently there are defined three types of telegrams: *Protocol Data Unit* (PDU), *Mailbox*, and *Network Variable* (NetVar). Each of the telegram type has its own structure consisting from the header and data payload. The PDU is used to configure slaves and for very fast data exchange. Remaining telegram types are slower and used for (un)buffered hand-shake type of communication.

One of the important feature is the *Distributed Clock* which is EtherCAT's service to provide precise, low jitter system time for the whole network. Thus, each slave can very precisely trigger local activities or measure some processes based on its local copy of the system time. Resolution of the system time is 10ns and practically achievable smallest jitter is about 11ns. Process of the synchronization of the DC has two phases. In the first one, delays between all slaves are measured. Based on measurements, the master calculates time offsets for every slave and writes this information to particular configuration registers on every slave (using PDU telegrams). In the second phase, called *Time Control Loop* the master periodically sends *Drift Compensation Telegram* DCT which help slaves to detect and compensate a drift in their local copy of the system time. Rate of the DCT has significant impact on the jitter. It is therefore crucial to send DCT regularly and at the rate high enough to ensure desired level of the jitter in the distributed clock.

The DC is tuned to a selected *time reference source* which is typically the first slave (in theory can be any slave in the ring or even the master). In some low-performance, high-jitter tolerant applications, the master can provide the reference time as well. Reason to use slaves for time reference rather than the master is due to the master's processing jitter which is much greater than slave's. Therefore, not every master's implementation supports precise synchronization of the master with slaves' DC. Implementations, which support such precise synchronization are part of so called *Class A* stack implementation category.

DC can be also combined together with redundancy. In case of a failure, two logical rings are created. Slave which was used before as a time reference is not anymore contained in the second network. There are some workarounds but they demand specific arrangement [7]. To our current knowledge, there is no reliable implementation of the DC with redundancy in sub-50μs.

3. Related work

ECM stack is available commercially as well as the open source. In the open source, there are currently available two stacks: *Simple Open-Source EtherCAT Master* (SOEM) and *IgH EtherCAT Master for Linux* as a part of the *EtherLab* suite. The earlier one is meant to be highly portable on variety of embedded platforms (HW and RTOSes). It was reported by several users that SOEM was successfully used in the 200-100μs. However, due to its simplicity and generality is not feasible in sub-100μs domain distributed controller. Its code uses *memcpy* instead of DMA, no zero-copy buffer (to avoid repeated copy of the same data in the memory) which introduces a great jitter. It also does not support synchronization between master and slaves, and does not contain the scheduler.

EtherLab requires real-time extension of the Linux. For low latency operation it has its own network driver. Determinism is "guaranteed" only with specific two network controllers (they provide a patch for a driver) and specific version of the Linux kernel. Regardless the good quality implementation, tight dependency on the PC archi-

tecture, specific network controller, specific Linux kernel, and specific real-time extension version makes use of this stack extremely limited. Contrary to the SOEM, EtherLab provides scheduler and is more comprehensive. From the performance perspective, the EtherLab can provide only 250μs cycle time with jitter of 13 μs [2]. It also does not provide support for tight synchronization between slaves and the master.

From commercially available we will list only those which achieved 50μs cycle time. Koening-PA provides standard master on various embedded platforms. Benchmarks show that it can achieve 50μs cycle times on very high-performance dual-core Intel CPU with 2.66 GHz clock and with use of network controller attached via PCI-express bus. Nevertheless, it is not clear whether they support Class A stack implementation in such time domain. Dependency on such specific high-performance CPU architecture, which also demands active cooling, makes it not very portable and suitable for fan-less embedded applications as indicated in Section 1.

Beckhoff, the company behind the EtherCAT technology, provides also EtherCAT master for very low cycle time called *eXtreme Fast Control* (XFC). It is tightly bound with the TwinCAT and their customized PLC based on Intel's high-performance CPU. In 2012 they have demonstrated cycle time with 12.5μs. From the description it is not clear whether they support distributed clocks and Class A master in such time domain.

To achieve cycle time well below 50μs requires very delicate HW and SW design, with optimized programs, stack, and drivers (as in the case of Beckhoff). There are also attempts to cut the overhead with introducing EtherCAT accelerators [4]. Authors implemented EtherCAT as partially SW running on the CPU and partially as an FPGA design to achieve desired speed of 20 kHz cycle rate. Their evaluation shows 19.2μs processing time in the SW part and 18.1μs in the FPGA accelerator, in total 37.3μs. Counting the transmission time of the smallest frame 5.76 μs plus Inter-Frame Gap of 0.96 μs, the authors in [4] can achieve the best cycle time of 45μs. However, their solution neither provides distributed clock nor a support for Class A implementation.

To our best knowledge, there is no such EtherCAT master implementation which is capable of sub-30μs cycle time, being portable on different embedded platforms (preferably fan-less), and at the same time supports very precise time synchronization of distributed clock in slaves and the master.

4. Design of scheduler and precise time

In order to successfully design a platform for distributed controller in the sub-30 μs cycle time, we have divided a design into the three logical parts which are in a certain degree interleaving. We fully followed the EtherCAT standard and used only standard-compliant components. There are neither hacks in the protocol nor in slaves.

4.1. Common system time

Precise common system time with low jitter (less than 40ns) is the fundamental building block of the distributed controller, especially in the sub-30μs. The question is, where to locate the source of the reference time. The problem is, that the value of the reference time is used to measure delays between slaves as well as is used later for the drift compensation (see Section 2). Thus, latency between acquiring the current value of the system time, sending it over the EtherCAT and time of arrival into a particular slave must have as lowest jitter as possible. Due to that, locating the time source in the master is not a feasible option because of quite high jitter (few microseconds) introduced by the platform (HW, RTOS, SW) itself (see measurements in Section 6).

Another option is to locate the source into one of the slaves. In such case given slave should be the first in the network so other slaves can directly synchronize to it without need to go via ECM. Slaves, contrary to the master, produces very small jitter, well below the 1μs scale. Nevertheless, the problem is how to synchronize the master and the time reference slave.

The proposed solution is in Figure 4 where we have introduced a new device into the network called *Time Reference Device* (TRD) which has embedded standard EtherCAT slave together with high quality clock source. From TRD there is a synchronization signal connected directly to the external interrupt pin of the CPU. The synchronization signal inside the TRD attached to the ECS's *Sync0* output signal. The *Sync* signal is a regular and standard feature of every ECS and its timing is based on the local copy of the common system time. In case of the TRD, Sync generation is based on the source of the reference time. When Sync0 is triggered, interrupt service routine in the master updates local copy of the system time in the master with a new value. The value is incremented by the period of the Sync0 from the TRD. Drawback of this solution is coarse-grained copy of the system time since it is not feasible to interrupt the master every 10ns in order to achieve the same clock resolution as in slaves. Nevertheless, system time on the master is used to feed both task and ECM schedulers which require periodic operation only at the microsecond level granularity.

4.2. Global scheduling

Scheduling of tasks in both the master and slaves has to be based on the common system time because tasks have certain data and control dependency which shall be guaranteed. In other words, the order of execution, starts, and deadlines of tasks have to be synchronized for the whole system based on the same precisely synchronized system time. Example of a schedule for the distributed controller is in Figure 2. Arrow up denotes start of a task's execution while arrow down marks deadline.

Tasks Y in slaves have data and control dependency on

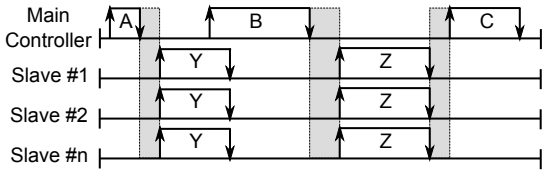

Figure 2. Example of a global schedule in a hard real-time system in one control cycle.

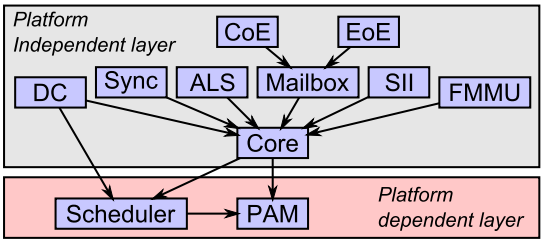

Figure 3. Devised EtherCAT's stack architecture

the master's task A. The same for tasks Z and B. Master's task C is data and control dependent on slaves' tasks Z. The gray color in Figure 2 denotes a region, when communication between the master and slaves has to occur (in our case EtherCAT). In case of A to Z communication, deadline of task A is also launch of the communication over the EtherCAT. Start of execution of tasks Y is also a deadline for EtherCAT to distribute data to all slaves in the system. All these activities shall happen within one period which can be less than 100μs.

To trigger tasks in slaves we use both *Sync0* and *Sync1* which are configured to run periodically with period equal to the cycle time but with different offsets. On the master's side we have to be able to trigger several control tasks as well as to schedule the communication on EtherCAT. Thus, we need more flexibility than in slaves to trigger several tasks and activities. For this reason we have defined *scheduler's base rate* which has lower period than is the control period. Then, all tasks and communication on EtherCAT are derived as an integer multiplication of the base rate together with their offsets w.r.t. start of the cycle. Source of the base rate is the synchronization signal (from the TRD), i.e. Sync0 of the TRD and has period equal or lower than the control period. The base rate purely depends on the application's scheduling resolution and the level of acceptable jitter in the system.

4.3. Master's scheduler

In order to support different types of communication on EtherCAT w.r.t. determinism, periodicity and time criticalness (very low latency), we have created an abstraction called *channel*. Channel has guaranteed communication bandwidth (rate and size of frames) and has associated certain channel properties. Based on this information, the scheduler determines from which channel will be at the given moment transmitted frame. This guarantee determinism of the whole schedule including deadlines.

Properties of a channel are: rate of sent (as a multiplication of the base rate), maximum frame size, offset w.r.t. start of the cycle, activate or disabled, one-shot mode or periodic mode. This allows us to cover following communication types:

- Periodic, deterministic: Deterministic send of a frame with given rate derived from the base rate and offset and with guaranteed maximal latency.
- Aperiodic, deterministic: One-shot send with guaranteed maximal latency, used in case of system failure (e.g. to order slaves into safe shutdown).
- Aperiodic, non-deterministic: One-shot best effort transmission, no guarantee on latency, used for system debugging or time uncritical diagnostic.

Property *Active* of a channel can be together with one-shot deterministic mode combined into various failure-recovery schemes. When a task does not finish on time, a watchdog will deactivate given periodic channel and activates preselected one-shot deterministic channel with preallocated telegram containing commands and data for slaves for the correct failure-recovery.

Latency in the system depends on the HW/SW overhead, jitter, as well as length of a frame. Overhead and jitter can be precisely measured (Section 6). Based on this information, the EtherCAT scheduler can perform simple schedulability test when a new channel is being registered (so called admission test). In order to guarantee maximal latency, each channel has associated maximum length of a frame. When a frame in the current channel has greater length then registered with the given channel, the send is rejected. Deeper details of the devised EtherCAT's scheduler are out of the paper's scope.

4.4. EtherCAT Master's Stack Architecture

To address portability requirements, we have proposed modular and layered EtherCAT's master stack architecture shown in Figure 3, where arrows denotes direction of dependency between modules. The architecture consists of two layers: platform dependent (PD) and platform independent (PI). The idea is that to deploy the EtherCAT master into a new platform would mean to update only modules in the PD layer while the rest of the stack remains untouched. Each layer has then its own structure consisting of modules with well-defined interfaces (services). Because of that, we can choose among different implementations of the given module depending on the requirements (small footprint, SMP safe functions etc.).

The PD layer contains two essential modules: EtherCAT scheduler and *Platform Abstraction Module* (PAM). The later one hosts implementation of platform dependent functions such as management of zero-copy

buffers, memory allocation, network interface card operations, DMA management, debug and logging functions, etc. *Scheduler* contains certain amount of generic and platform independent functionality. However, it also contains some platform specific code for interrupt service routine for TRD's synchronization signal.

The PI layer has one central module *Core* which performs the essential EtherCAT functions. For example, it parses frames, manages callback handlers for telegrams, creates fundamental datagrams and frames, provides wrappers for synchronous send operations, etc. Other modules use these functions in order to perform their own operations. For the sake of example: The *Slave Information Interface* (SII) needs to perform several read and write operations in slaves' registers in order to get the content of the EEPROM attached to the slave. *SII* module then uses services of the *Core* to create needed frames with proper PDU telegrams. The same for other modules such as *DC* (responsible for initial configuration of the system time), mailbox (which is used to run Ethernet-over-EtherCAT, CAN-over-EtherCAT, etc.), application layer services (ALS), and many more. Such modularity allows including into the target platform only modules whose functionality is required in the application. This is favorable feature for optimizing the footprint, safety purposes, or for debugging and unit testing.

5. Implementation

Based on the design briefly described in Section 4 we have implemented prototype of the distributed controller into an experimental platform shown in Figure 4. Each part of the platform is described in following subsections.

5.1. EtherCAT Slaves

For EtherCAT slaves (on the figure as Slave #n) we used FPGA development boards Spartan-6 LX150T from Avnet. In FPGA we have deployed Beckhoff's EtherCAT slave IP (version 2.04a) and via FMC connector attached networking modules (called ISMNET FMC) from Avnet which contains circuits for PHY (incl RJ45 connectors), EEPROM, and source of clock signal from a crystal oscillator. The networking modules provide additional connectors where we have routed out from EtherCAT IP both *Sync0* and *Sync1* signals. These signals can be then attached to the oscilloscope for debugging and measurements. EtherCAT IPs were then attached via bus to the *MicroBlaze* softcore. Both Sync signals were also routed to the MicroBlaze's external interrupts to trigger slave's control tasks. In our case these tasks performed simple operations just to emulate real control system.

5.2. Time reference device

Prototype of the TRD was also built on the same FPGA development board as the regular slave described above. Here we have deployed just one EtherCAT IP with its own networking module (the same as above) and routed out the Sync0 pulse.

5.3. Master

As the master for the experimental platform we selected Freescale's P2020RDB-PCA reference design board with P2020 communication processor with two (2.5 DMIPS/GHz each) 32-bit big-endian e500v2 cores, both clocked at 1.2 GHz. The processor has passive cooling which is very favorable in most industrial applications. P2020 has on-chip three high-speed advanced configuration *Medium Access Controllers* (MAC) including separate DMA controller with cache coherency. This tight integration of MACs with the CPU helps to minimize the HW latency which could be much higher if MACs are connected to the CPU by an external bus such as PCIexpress.

As the real-time operating system we decided to use VxWorks due to its very low latency, good determinism, and minimal software overhead [1]. The version deployed in our platform is 6.8 with SMP. Whole EtherCAT stack was implemented in ANSI-C. Initially the master used VxWorks' network stack architecture and generic Freescale's MAC driver. This setup yielded communication jitter 50μs with maximum cycle time 200μs. Therefore, we implemented our own MAC driver for VxWorks which provides desired determinism, very small overhear, small jitter (as shown later in Section 6), and supports zero-copy buffers. Because the P2020RDB board does not provide connectors to attach external signals, we did simple modifications (patching and soldering) in order to route out and to connect synchronization signal from TRD to the P2020 processor. The signal was attached to the external HW interrupt 0, i.e. the highest priority. Given interrupt service routine handles updates of the local copy of the system time as well as controls the scheduler.

To further minimize the latency, all frames for deterministic channels have preallocated memory. Application tasks can map their variables directly into telegrams' payload field. When frames returns back from the network, they are parsed into telegrams and each telegram has associated a callback handler. Zero-copy buffer design and redesigned driver of the MAC controller avoids time-costly repeated copy of the data in the memory.

6. Experiments and Results

6.1. Common system time preciseness

To measure preciseness of the common system time we let all slaves to synchronously and periodically generate pulses on Sync0 signal. In Figure 5 is a snapshot from the oscilloscope showing the reference Sync0 signal from the TRD. Other three signals are taken from regular slaves. Resulted jitter in the synchronization is at the level of 11 ns for each slave, system jitter is around 14ns which very close to the theoretical minimal jitter (time resolution of the EtherCAT) 10ns.

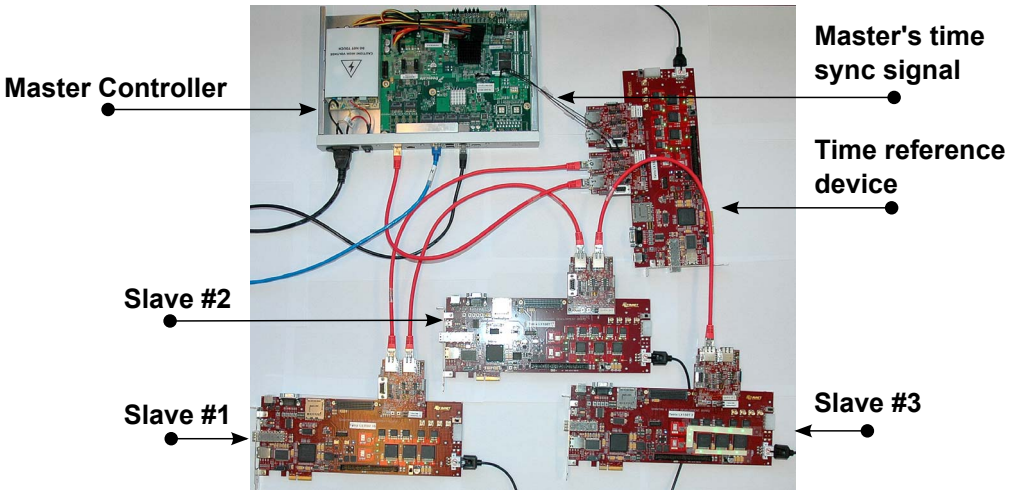

Figure 4. View on the experimental setup. Red ethernet cables denote EtherCAT network.

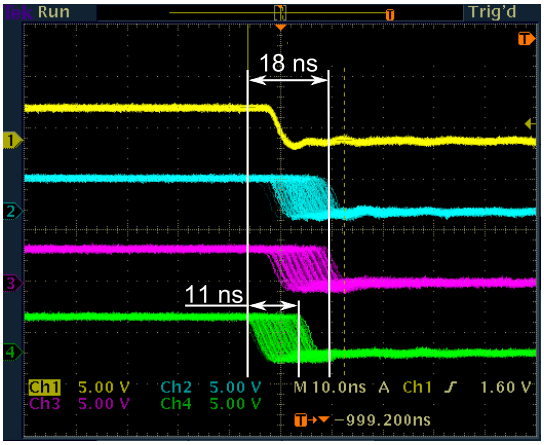

Figure 5. Common system time jitter between TRD (the top) and three slaves

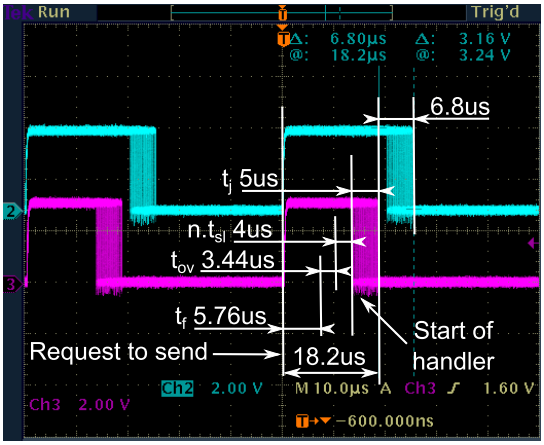

Figure 6. Latency and jitter on two consecutively sent frames to four slaves

6.2. Measuring HW/SW overhead

Total overhead on our experimental platform has three components: i) HW itself, ii) RTOS, and iii) EtherCAT stack. The overhead determines lowest achievable control cycle time. Knowing the overhead also helps to properly configure the EtherCAT scheduler (Section 4) to guarantee channels' bandwidth and deterministic behavior. To measure overhead together with jitter we can use P2020's internal timer (SW measurement) or general purpose I/O (GPIO) signals on the CPU. The first method is easier to use but is less precise due to the timer's granularity (16ns) and overhead of 272ns for value's read-out. The HW method is far more precise with granularity at the instruction level (less than 16ns). However, used experimental board P2020RDB-PCA uses GPIOs to control some peripheral circuits. Thus, we disconnected those devices and found a place on the PCB where we can pick up the GPIO signals directly with the oscilloscope's probe.

After that, we created few routines that each time the EtherCAT stack is ordered to send a frame set the GPIO pin. Once the frame is received and parsed, registered telegram handler is called that in turn clears the GPIO. Clearing the GPIO happens at the same time as the user defined processing of received data starts. Overhead to set and clear the GPIO pin is less than 200ns. To measure also the delay, which system needs to send another consecutive frame, we ordered the stack to send 2 frames in a row. Measurements from the periodic send of these two frames are shown in Figure 6.

Different GPIOs were used for each frame. Used frames had the minimal length, i.e. 46 bytes in the payload which yields 32 bytes data field for EtherCAT's PDU telegram. The bottom signal is related to the first frame and the upper to the second one. For the duration of the first frame's pulse on the scope holds Eq. (1) where t_f is time needed to send out the frame, t_{sl} is latency of a single ECS, t_{ov} stands for total overhead, and n is number of slaves in the network.

$$t_{pulse} = t_j + t_{oh} + t_f + n.t_{sl} \quad (1)$$
$$t_{oh} = t_{pulse} - t_j - t_f - n.t_{sl} \quad (2)$$

From the Eq. (1) we can easily derive Eq. (2) to calculate the overhead. All values on the right are know: pulse length is 18.2μs, jitter can be clearly read as 5μs, number of slaves is 4. Latency on used slaves was before measured as 760ns but for easier calculation and to cover the slaves' jitter (which is negligible $<<$ 1μs) we used conservative value of 1μs. Time to send out the frame on the 100Mbps Ethernet is obtained by multiplying its total length (64 B + 7 B preamble + 1 B start of frame delimiter = 72 B) with the bit's time length (10ns) which yields 5.76μs. Filling these values into the Eq. 2 returns total system overhead in one cycle equal to only 3.44μs.

In order to estimate the lowest possible cycle time, we have to add to the Eq. 1 *Inter Frame Gap* (IFG) which is Ethernet's mandatory minimal idle time before a new transmission can occur. Its value is defined by the standard to 12B, i.e. 0.96μs. Thus, the lowest cycle time of frame of a given size is in Eq. (3). After filling known values from the current P2020 based platform we get the lowest cycle time in Eq. (4).

$$t_{cycle}(size) = t_j + t_{ov} + t_f(size) + n.t_{sl} + t_{ifg} \quad (3)$$
$$t_{cycle}(size) = 9.4\mu s + t_f(size) + n.t_{sl} \quad (4)$$

The upper signal in Figure 6 has falling edge postponed by 6.8μs w.r.t. the end of the first signal. Subtracting from this value t_f (5.76μs) and t_{ifg} (0.96μs) we get only 80ns overhead. This overhead is caused only on the receiving side because all ready frames are pre-fetcheed by P2020's MAC into its transmission buffer. As seen here, devised EtherCAT stack is highly efficient even though we used only C-code and no inline assembler.

To measure the latency in the scheduler, we performed another measurement where we measure the time between the synchronization pulse (generated by the TRD) and the moment when a frame from the active channel is sent out. Results are shown in Figure 7 where the synchronization pulse (on top, attached directly to Sync0 of the TRD) activates the interrupt service routine on falling edge. The GPIO is set immediately after the P2020's MAC was ordered to send a frame. The delay between the synchronization pulse and the set of the GPIO is a little bit less than 500ns, where 20ns is latency of the GPIO command. Part of the latency is consumed inside the VxWorks' interrupt dispatcher. That is, the scheduler has 0.5μs latency.

6.3. Experiments with 30μs cycle

Previous measurements show that our devised EtherCAT based platform is able to handle sub-50μs cycle times. For this reason we set up an experiment with distributed control running at 30μs cycle time. Our experimental control application requires periodic

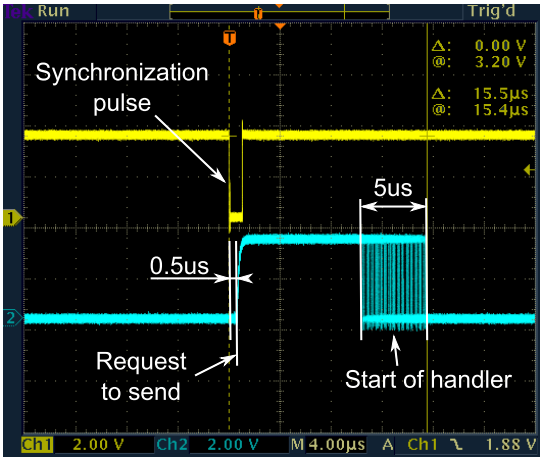

Figure 7. Latency and jitter on the master scheduler w.r.t. common system time. Network with one slave.

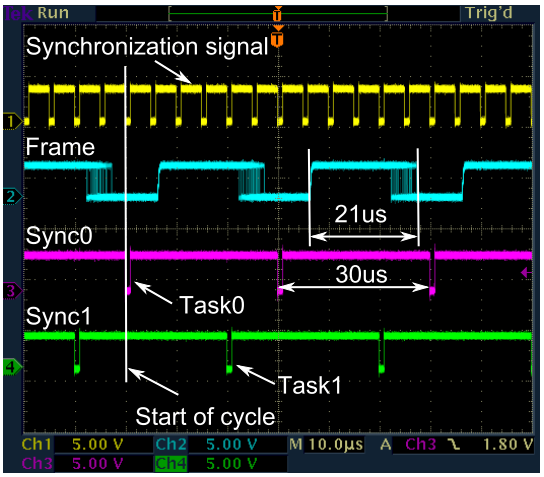

Figure 8. Experiments with 30μs cycle time with two telegrams in a single frame

triggering of two tasks in slaves (Task0 and Task1). One task shall start at the beginning of the cycle and the second one after receiving data from the master. Data from the master shall have 50B size. Results are show in Figure 8.

The top signal is the synchronization signal generated by the TRD. The second signal shows start of the frame send and the moment when the telegram handler is called. The third signal is the Sync0 which is triggered always at the start of the cycle in all slaves. The last signal is the Sync1 triggered at the moment when data from the master should be written in slaves and slaves can launch next task. However, we have to also send regularly drift compensation telegram in order to compensate drift in the system time (Sec. 2). We can launch a separate frame but to preserve the bandwidth, we included this telegram (20B) into the same single frame with the control data telegram.

This is also the reason why Sync1 is triggered even though when the frame is not fully received in the master. When the first telegram passes the last slave, the tasks in slaves can be launched with Sync1 signal. Both Sync signals are taken from different slaves to demonstrate the precise synchronization.

Based on the performance measurements we have calculated that this frame has to be sent 5μs after start of the control cycle. For this reason we set the granularity of the scheduler to 5μs, i.e. synchronization signal generated by the TRD has period of 5μs. The offset of the Sync1 w.r.t. start of the control cycle was calculated to be 20μs. The experiment run for two days without any interruption or error, i.e. the platform had stable and deterministic performance.

Here we can also validate Eq. (4) which yields cycle time 22.2μs (9.4 + 8.8 + 4*1). Size of the frame is 110 bytes (payload 50B + PDU header 12B + EtherCAT header 2B + Ethernet header 14B + Ethernet checksum 4B + preamble 7B + start of frame delimiter 1B + drift compensation telegram 20B). By subtracting IFG (0.96μs) we get 21.24μs which is very close to the oscilloscope's reading 21μs. Difference 240 ns is due to the pessimistic estimation of the latency in slaves (1μs).

6.4. Measurements summary

Achieved system jitter in the common system time is 18.8ns while jitters on individual slaves are 11ns, which is very near to the EtherCAT's clock resolution of 10ns. Total HW, RTOS, and SW overhead on the experimental platform is 3.4μs where: parsing of the EtherCAT frame in our stack takes 80ns and the latency of the EtherCAT scheduler is below 500ns. This yields less than 0.6μs overhead in the SW, the rest (2.9μs) is spent in the HW which performs DMA, and prepares the frame for transmission. However, jitter in HW, RTOS, and SW is 5μs, i.e. greater than the overhead. Such relatively great jitter w.r.t. the overhead is caused by the DMA and cache (in case of the transmission). Another source of the jitter on the master is caused by interrupts. P2020's MAC controllers have relatively low priority (the highest MACs IRQ is 28), thus there is an interference with other devices.

7. Conclusions and future work

This paper has documented the design and implementation of a high-end EtherCAT master for distributed controller capable of cycle times below 30μs in combination with time synchronization with an accuracy better than 20ns. The EtherCAT master has been implemented to make porting to other platforms as easy as possible. The common system time shared by the master and the devices enables highly precise scheduling of all real-time tasks in the system (on the master as well as slaves).

This paper has therefore demonstrated the feasibility of EtherCAT in the most demanding real-time applications by presenting an implementation platform based on the Freescale's P2020 dual-core processor that combines generality and ease-of-porting with the top-level performance.

Further work is to port the proposed EtherCAT master to various platforms and also to cope with the jitter in order to even decrease the cycle time at the level of 12.5μs (80kHz). At the moment we have successfully ported the master on the Xilinx's Zynq SoC platform and initial measurements shows 17μs cycle times. Once the deterministic driver for the Zynq's network controller was in place, it took less than a day to prepare the fully fledged stack (update of the Platform Dependent Layer, Figure 3). Besides that, we want to also introduce reliable redundancy together with the distributed clock in sub-30μs time domain. This has not yet been successfully achieved and we believe that proposed stack and the platform provide solid foundations for such implementation.

References

[1] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio. Performance comparison of vxworks, linux, rtai, and xenomai in a hard real-time application. *IEEE Transactions on Nuclear Science*, 55(1):435 –439, feb. 2008.

[2] M. Cereia, I. Bertolotti, and S. Scanzio. Performance of a real-time ethercat master under linux. *IEEE Transactions on Industrial Informatics*, 7(4):679–687, nov. 2011.

[3] EtherCAT Technology Group. http://www.ethercat.org/. Accessed on 28 March 2013.

[4] T. Maruyama and T. Yamada. Hardware acceleration architecture for ethercat master controller. In *9th IEEE International Workshop on Factory Communication Systems (WFCS), 2012*, pages 223–232, may 2012.

[5] PROFIBUS & PROFINET International. http://www.profibus.com/. Accessed on 28 March 2013.

[6] G. Prytz. A performance analysis of ethercat and profinet irt. In *IEEE International Conference on Emerging Technologies and Factory Automation, 2008. ETFA 2008.*, pages 408–415, sept. 2008.

[7] G. Prytz and J. Skaalvik. Redundant and synchronized ethercat network. In *International Symposium on Industrial Embedded Systems (SIES), 2010*, pages 201–204, july 2010.

[8] C. Toh and L. Norum. A performance analysis of three potential control network for monitoring and control in power electronics converter. In *IEEE International Conference on Industrial Technology (ICIT), 2012*, pages 224–229, march 2012.

[9] L. Wang, H. Jia, J. Qi, and B. Fang. The construction of soft servo networked motion control system based on ethercat. In *International Conference on Environmental Science and Information Application Technology (ESIAT), 2010*, volume 3, pages 356–358, july 2010.

[10] W. Zheng, H. Ma, and X. He. Modeling, analysis, and implementation of real time network controlled parallel multi-inverter systems. In *7th International Power Electronics and Motion Control Conference (IPEMC), 2012*, volume 2, pages 1125–1130, june 2012.