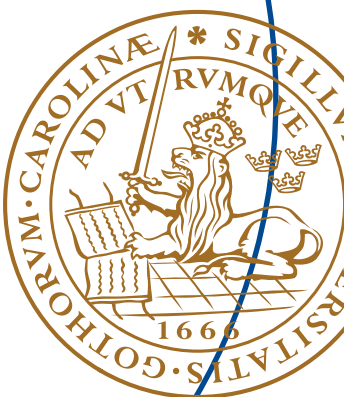
Master's Thesis

Time Synchronization system, Investigation and Implementation Proposal

Fabian Winquist
Omar Abdilameer



Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, August 2015.

Time Synchronization system, Investigation and Implementation Proposal

Mater Thesis

Fabian Winquist
fabian.winquist@gmail.com,
Omar Abdilameer
adi10oab@student.lu.se

Department of Electrical and Information Technology
Lund University

Advisor: Jens A. Andersson, Ander J. Johansson, Stefan Höst

August 21, 2015

Printed in Sweden
E-huset, Lund, 2015

Abstract

Timing and synchronization in time is very important in many types of situations e.g. in most measuring systems, during control of large and complex systems or governance processes. There are different scenarios with different demands on time synchronization accuracy regarding relative time measurements. Some requires sub-nanosecond accuracy other sub-microsecond accuracy. The White Rabbit project addresses this issue by using special hardware and software to achieve sub-nanosecond accuracy. Because specialized hardware for synchronization is not suitable for laptops, the goal of this thesis is to propose one or more software-based solutions for synchronizing computers / laptops in a system with high or with a low demand on the relative time accuracy. In this thesis we investigate the newly defined PTP Ported to Silicon (PPSi) extension as a free software package that supports Cygwin (Windows) and Linux operating systems and even freestanding environments. PPSi is becoming a promising purely software PTP implementation, capable of supporting a range of portable devices and platforms in accelerator projects, with a good performance and scalability.

Acknowledgments

This master thesis project was supported by department of electrical and information technology, faculty of engineering, LTH, Lund university and European Spallation Source (ESS). We thank our supervisors from institution of electronic and information who provided insight and expertise that greatly assisted during this master thesis.

We thank our supervisor - Jens Andersson, an excellent lecturer from faculty of engineering at LTH, for assistance with his full technical experience and knowledge to help us find the current problems and locate the suggestions, and our examinator Stefan Höst, lecturer, faculty of engineering, LTH for comments that greatly improved our work. Also like to thank our supervisor from ESS and LTH, Anders J Johansson for his useful comments, expertise, helping oss getting the equipment we needed.

We also want to thank Professor Alessandro Rubini, programme officer at Fondazione Cariplo for his numerous beneficial discussions during the development of the PPSi Cygwin extension and for his support. Alessandro Rubini was one of the developers of PPSi.

We also want to thank Cosylabs, who offered us two SPEC boards, a SPEC White Rabbit master as synchronizing source, and a SPEC White Rabbit slave. We thank ESS for providing a pair of fiber, a coaxial cable and a media converter, which we need for testing and validate our implementation.

We would also like to show our gratitude to all the staff in the electronic and information department at LTH, for sharing their lab-rooms and provide us the equipments which we needed during the course of this master thesis.

Table of Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
BC	Boundary Clock
BMC	Best Mater Clock
BMCA	Best Mater Clock Algorithm.
CERN	European Organization for Nuclear Research
CPU	Central Processing Unit
DLL	Dynamic-link Library
DMTD	Dual Mixer Time Reference
DTSS	Digital Time Synchronization Service
ESS	European Spallation Source
FFO	Fractional Frequency Offset
FLL	Frequency-locked loop
FPGA	Field-Programmable Gate Array
GMC	Grand Master Clock
GMT	General Machine Timing
GNU	Unix-like operating system developed by GNU project
GPS	Global Positioning System
GSI	The GSI Helmholtz Center for Heavy Ion Research
IEEE	Institute of Electrical and Electronics Engineers
IPv6	Internet Protocol version 6
LAN	Local Area Network
LGPL	Lesser General Public License
LHC	Large Hadron Collider

MAC	Media Access Control
MC	Master Clock
NIST	National Institute of Standards and Technology
NTP	Network Time Protocol
OC	Ordinary Clock
OS	Operating System
P	Precise Code reserved for military use.
PC	Personal Computer
PLL	Phase-locked loop
POSIX	Portable Operating System Interface
PPS	Pulse per Second
PPSi	PTP Ported to silicon
PTP	Precise Time Protocol
PTPd	Precision Time Protocol daemon
RMS	Root-mean Square
SC	Slave Clock
SOC	Slave Only Clock
SPEC	Simple PCIe FMC Carrier
TAI	Temps Atomique International - International Atomic Time
TC	Transparent Clock
UDP	User Datagram Protocol
UTC	Universal Coordinated Time - International Atomic Time
VFO	Variable-frequency Oscillator
WAN	Wide Area Network
WR	White Rabbit
WRPC	White Rabbit PTP Core
WRPTP	White Rabbit extension for PTP

Chapter 1
Introduction

The need to synchronize computer clocks in a network is almost as old as the internet itself. In fact one of the oldest protocol still in use, called Network Time Protocol (NTP), is a protocol for synchronizing computer in a wide area network (WAN) and/or local area network (LAN). The question is what kind of synchronization solutions exist today? Since NTP and DTSS (DTSS is a much simpler time protocol compared to NTP) more complex timing systems has arrived, with a higher demand on time accuracy.

New and more complex timing systems have created new projects and research in the time synchronization area to accommodate the new demands (e.g. high synchronization accuracy). These projects have led development to a new protocol Precision Time Protocol (PTP) and hardware assisted PTP, White Rabbit (WR). As of today there exist a couple of commercial and Open Source implementation of PTP such as PTP daemon (PTPd) and PTP ported to silicon (PPSi).

1.1 Background

Timing and time synchronization is very important in many types of situations, e.g. in most measuring systems, during control of large and complex systems or governance processes. A typical example of a measuring system with a very high demand on timing and time synchronization is accelerators in e.g. European Spallation Source (ESS). Another type of situation that is less demanding is a simple lab set up. Where the goals could be to investigate how disturbance events affect the quality of service at different levels in the OSI model.

The European Spallation Source (ESS) is an association of European nations that interact in the design and manufacture of one of Europe's largest active infrastructure projects that investigate scientific questions using neutron beams.[1] At present, 17 European countries have been committed to construct ESS with Sweden and Denmark as host nations. The facility is being built in Lund (Sweden), while the Data Management and Software Centre (DMSC) are located in Copenhagen.[2] ESS will be built, owned and operated by ESS AB. ESS AB is a public company owned by the Swedish and Danish governments but it will be possible for all other partners to contribute in this company.[2] According to [3], the ESS accelerator is a major user facility known as a neutron source where cascades of neutrons induced by spallation that provides proton beam with high level

requirements at intense of 5 MW. As shown in Figure 1.1, the ESS accelerator produces protons at the ion source. These protons are transported through different sections to the target that accelerates them to an appropriate energy to create neutrons via the spallation process so that they can then be used in future researches.[3]

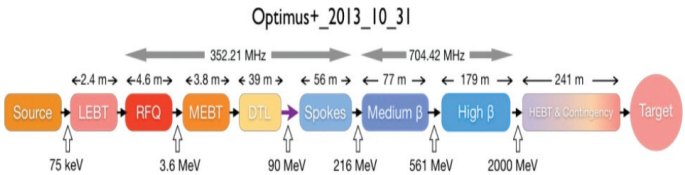


Figure 1.1: A block diagram of ESS accelerator. credit ESS.

ESS will become the world's most powerful neutron source that can be used in a wide range of sciences, such as materials science, structural chemistry, biology and geophysics. ESS will also support future researchs in medicine, environmental science, climate, communication and transportation.

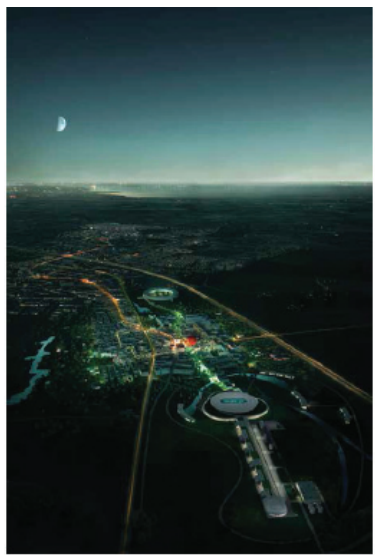


Figure 1.2: ESS overview. credit ESS technical design report.

A problem arises when scientists want to synchronize their laptop with any of the ESS measuring devices (which are considering using White Rabbit hardware for time synchronization). Scientists simply do not want to buy expensive WR

hardware only for this use case, still they want to synchronize their computer with the WR network, with as high accuracy as possible. In this thesis the laptops are considered to run either Windows or UNIX.

As mentioned there exist different scenarios with different demands on accuracy regarding relative time measurements. Some require sub-nanosecond accuracy and other sub-microsecond accuracy. There are a couple of projects that address this issue, like the White Rabbit project. The White Rabbit is an open source project, based on Ethernet. In this project, specialized hardware and software are used to achieve sub-nanosecond accuracy using fiber with minimum link delay, with length up to 10 km. In some cases specialized hardware is impossible/difficult to attach to a laptop and a completely software based solution may be the only alternative.[4][5]

1.2 Scope

The goal with this master thesis is to propose one or more software based solution for synchronizing computers/laptops with system with high or with a low demand on relative time accuracy. A high demand on relative time accuracy is represented by accuracy in scale of nano- or microseconds and a low demand represents time accuracy in scale of milliseconds.

1.3 Thesis outline

Chapter 1 In chapter 1, an introduction to time synchronization in networks is presented as well as an overview of how time synchronization is achieved today.

Chapter 2 In chapter 2, the theory required to understand this master thesis is explained, including how the protocol used today to achieve time synchronization and a general description of the protocols is presented. A small introduction of timing system is also presented in this chapter.

Chapter 3 The PPSi solution is explained in chapter 3, with a detailed description of the software and a declaration of why the software was decided to be the focus in this work. A description of Cygwin is presented in this chapter as well.

Chapter 4 In chapter 4, the result and verification of the suggested software solution are presented, including a comparison between the Cygwin (Windows) configuration and the Linux configuration.

Chapter 5 In the final chapter of the report, a compilation of the discussion and future work are presented.

Chapter 2
Theoretical Background

This chapter presents basic theoretical information that is required to understand the fundamental concepts of this thesis. The reader of this thesis has to understand the general problems with timing system, how does time synchronization work, time offset, frequency offset and jitter. In this chapter, the reader can also find information about various time synchronization protocols such as, NTP, PTP and also a short description of the White Rabbit project.

2.1 Timing system

Time synchronization is necessary in many applications involving widely distributed resources in our life. For example, financial and legal transactions, transportation and distribution systems. The idea of clock synchronization is based on that internal computer clocks may be accurate initially, but after a while there will be a clock error due to clock drift so these drifty clocks must be synchronized to a reference clock. To understand the idea of why time synchronization is needed, an example is presented. In the airline reservation system, a seat can be sold twice or not at all if the distributed computers are not synchronized.[6] It is important that every timing system uses a reference time and that all nodes are synchronized to that timing source. The primary clock source must fulfill rigorous requirements so that it can be traceable to a common time standard such as universal coordinated time (UTC) or international atomic time (TAI). GPS and atomic clocks are examples of primary clock sources.

2.1.1 GPS

GPS can be used as a reference for providing time synchronization in most computer networks used today, by using additional technologies the desired accuracy regarding relative time measurements (e.g. sub-nanoseconds accuracy) can be achieved. GPS is a satellite-based radio navigation system, developed and run by the Department of Defense (DoD) in U.S. for the U.S. military, used to provide estimates of position, velocity and time and jointly controlled by defense and transport ministries. DoD also transmits time signals on public carriers, which can be used by everyone. GPS is referenced to an international time reference and provides very accurate time synchronization and noise immunity but GPS

is scarce in computer synchronization since GPS requires an unobstructed line of sight between transmitter and receiver. A GPS receiver can only be used outdoors. Figure 2.1 shows a simple structure for a network synchronized via GPS. The GPS sends time data messages to the time server which in turns (by using one of the synchronization protocols) sends these time messages to the devices that need an accurate time to synchronize their internal clock.[6]

There is various time synchronization protocols used today in computer networks. These protocols provide time synchronization in different ways depending of accuracy requirements. Either the client sends a request to the server and then the server sends a time message containing its current time, or the server sends its current time to a group of clients needing a reference time to synchronize their local clock using the received time messages. The most important time synchronization protocols used are NTP and PTP.[6]

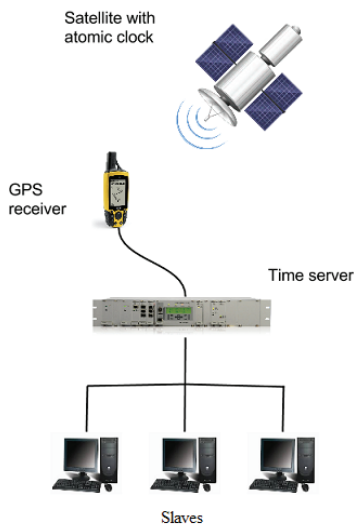


Figure 2.1: Structer of a network synchronized by GPS.

2.1.2 Atomic Clock

An atomic clock is another way of obtaining accurate time and provides very stable and precise timescale. GPS navigation systems use atomic clocks. Without atomic clocks the internet would not be synchronized. An atomic clock measures the time by locking an electronic oscillator to the frequency of an atomic transition where the oscillator is regulated by the vibration frequencies of an atomic system. The frequencies are associated with a transition in cesium-133 atom where 1 second = 9,192, 631,770 cycles of the standard Cs-133 transition. The most accurate atomic

clocks used today are cesium beam atomic clocks and rubidium clocks.[7]

In US, the cesium atomic clocks are maintained by the National Institutes of Standards and Technology (NIST) in Boulder, Colorado. The international second standard published in 1964 was based on the orbital period of the earth. In 1967 a second standard was based on the frequency of a transition in Cs-133 atom (this frequency is exactly 9,192, 631,770 Hz).[7]

A cesium atomic clock operates by locking a crystal oscillator to the principal microwave resonance frequency of the cesium atom. Figure 2.2 shows a simplified structure of a cesium clock.

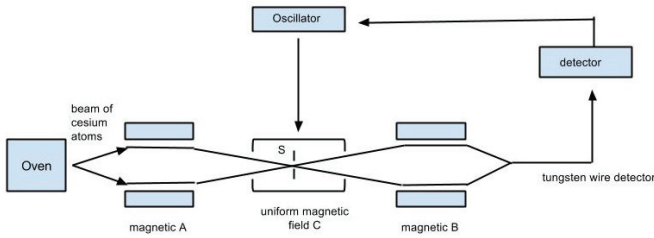


Figure 2.2: A simplified structure of cesium atomic clock.

2.2 Computer clock

In most computers, the clock is represented by quartz or a SAW resonator oscillator and a hardware counter that interrupts the processor at an interval of a few microseconds called a tick. With each tick, the value of the hardware counter is added to the variable representing the clock value. The system clock can be read and set by an application, the readings increment at a nominal rate, depending on the frequency of a tick. The frequency of the tick can be changed by a small amount in some OS i.e. UNIX. It is not possible to change this value in Windows, but there is a function that speeds up or down the system clock.[8] [9]

2.3 Time synchronization

The principal of synchronizing two computers is surprisingly simple and is the same for all time protocols (NTP, DTSS and PTP). The principal is about measuring the time offset between the two computers system clock. According to [9] the time offset can be realized with this expression:

$$T(t) = T(t_0) + R(t - t_0) + D(t - t_0)^2 + x(t) \tag{2.1}$$

T - time offset, t - current time, t_0- the time at the previous measurement update, R - frequency offset (clock wander (jitter) due to temperature change etc...), D - drift due to resonator aging and x - stochastic error term. In most cases, only the two first terms are calculated since they are dominating the last two (this is the case for NTP and PTP, while DTSS only measures the first term). Either PTP or NTP implementations for Windows adjust for frequency offsets since Windows lack support for this feature.[9][12]

2.3.1 Time offset

The principal of calculating the time offset is trivial and is the same for all mentioned time protocols and is based on timestamps.

The fastest way for manually synchronizing your clocks at home would be to use your wristwatch (which you have synchronized with an outside source i.e. bus terminal) aka master clock. Then you would one by one adjust each clock in your house aka slave clock, to match the time on your wristwatch. There will be a small but acceptable offset between the master clock/wristwatch and the slave clock, depending on how long time it took to adjust the slave clock.

This principle is the same for synchronizing computer(s) and server(s), except there can be several master clocks. They are all stationary and the offset varies a lot more between the master and the slave clock as a product of the one-way delay. The basic concept for estimating the offset is shown in Figure 2.3.

In order to calculate the time offset between the master and the slave, the slave sends a request synchronization message to the master. Before sending the message a timestamp t_1 of the slave clock is captured. On arrival the master captures a timestamp t_2 from its system clock and sends a response message, including t_2 and t_3, $t_3$is captured before sending response message. Once the slave got the response

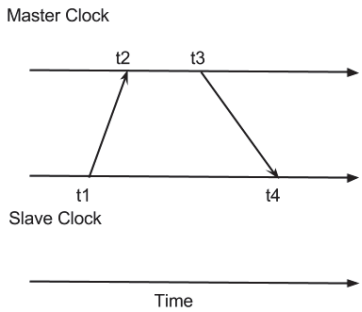


Figure 2.3: Concept of estimating time offset.

message it captures a timestamp t_4. After this procedure the slave has acquired all four timestamps and can estimate the time offset.[9]

$$T = \frac{1}{2}[(t_2 - t_1) + T\,(t_4 - t_3)] \tag{2.2}$$

As should be clear, the offset is half of the round-trip delay and the precision of this algorithm is heavily affected by two factors. The precision of the timestamps, the closer to the hardware the timestamps are captured the better, to eliminate as much OS latencies (caused by interrupts and delays in the kernel) as possible. OS latencies are particularly harmful for this algorithm due to its high variance. The second factor is the variance of the one-way delay calculation. Since the time offset/one-way delay is calculated as the half of the round-trip delay and due to the high latency variance in today's network, this will impact the precision of this algorithm.[10]

2.3.2 Asymmetric delay

Asymmetric delays as mentioned before is undoubtedly the major contributor for time offset and synchronization error, compared to symmetric delay which is much easier to account for. Asymmetric delays can manifest in several different ways: variation in queuing delays, variance in the network traffic, OS latencies, etc.... The problem is that there is no good way to accommodate asymmetric delays, due to its nature. White Rabbit solves most of the asymmetric delays by using superfast WR switches, which has a deterministic delay and by capturing the timestamps in hardware. Both NTP and PTP have ways for coping with asymmetric delay. The first factor is to eliminate OS latencies, this can be done by time stamping closer to physical layer e.g. MAC-layer. This will accord to [9] decrease OS latencies from mille- or microseconds to nanoseconds.

To cope with asymmetry latencies on the path between master and slave, NTP uses the Huff-n'-Puff Filter algorithm. This algorithm is designed to correct the time offset in scenarios where either the upload or download link is considerably congested compared to the other. This can occurs because of high possibility for

difference in traffic volume on download or uploads direction of the transmission. Huff-n'-Puff Filter algorithm measures the delay between the upload and download link for several hours and remembers the minimum delay. In later stages with large delays, this algorithm corrects the current offset using the sign of the offset and the difference between the current offset and the minimum offset. Depending on the sign of the offset, a negative or positive correction is performed.[9] [11]

For PTP on the other hand, four different methods have been suggested as a way of coping with asymmetric delays.

One solution is to employ devices that have PTP support and have capabilities to act as a PTP transparent clock. A transparent clock measures the time a PTP packet is spent in the network device and this time is added in the correctionField in the PTP frame, which is later used in the slave node for calculating the offset. This method is probably the most accurate way of dealing with asymmetric delay compared to the other methods, but has a side effect of being costly and not everyone is willing to update their devices with PTP support.[10] [23]

Another method is to use boundary clocks (network device that can act as a master and a slave, but on different PTP ports). According to [24], in the case where the same number of network devices is used, adding a boundary clock between them can be helpful in reducing the asymmetric delay. As for the transparent clock method, the boundary clock method needs PTP support in switches/routers, but is less reliable compared to the first method.[10] [23]

In the last two methods, PTP supported routers/switches are not needed. The first of the two is based on packet priority. In theory a packet with high priority suffers less from queuing latencies and congestion.[10]

The final method is to configure switches/routers to have a fixed delay for PTP frames. If PTP packet has a fixed delay in each network device the impact of asymmetric delay is reduced.[10]

According to the institute of electrical and electronics engineers (IEEE) PTP Power Profile default configuration values, the PTP packet has a default priority of 4 which is a bit higher than "normal" packets, with transparent clock enabled and boundary clock disabled.[16]

2.3.3 Jitter and Wander

Jitter between two clocks occurs when a variation occurs in the clock cycle, due to the fact that the synchronization period between two clocks can never be perfectly maintained. Jitter is quantified as the root-mean-square (RMS) between series of time offsets. Clock wander is a product of the frequency offset between two clock and is quantified as the RMS between a series of frequency offsets.[9]

2.3.4 Frequency offset

The second part of synchronizing slave and master is by measuring the frequency offset/error. The frequency drift/error can manifest in several different ways, but is often caused by environment issue on the oscillator, e.g. temperature, pressure and voltage fluctuation. [9][10]

Frequency errors are usually measured in ppm (parts per million) and can be as large as several hundred parts per million. If there is no means of correcting frequency error, an error of 100 ppm will introduce a time error of 8 seconds per day.[9]

According to [9, p.6] the frequency error due to oscillator wander is a function of averaging time, which in turns depends on the rate of time updates. Rate of time updates less than around 15 min, frequency errors are dominated by network jitter and time updates above 15 min, the oscillator wander dominates. NTP uses a clock discipline algorithm, used to minimize the time offset and frequency offsets and are described in more detail later.[9]

Calculating the frequency offset (FFO) between master (with frequency F_M) and slave (with frequency F_S) is trivial and can be calculated as:

$$FFO = \frac{F_M - F_S}{F_S} \tag{2.3}$$

The values of R and D from (2.1) can be derived from the equation (2.3). In the last term of (2.1) x includes all errors not covered by the other error terms, such as temporary and unobvious frequency errors.[9] [10]

2.3.5 Clock discipline methods

To accommodate jitter and clock wander, a hybrid phase/frequency-lock feedback loop algorithm is used. These algorithms which can operate in two modes; PLL and FLL are used in the NTP discipline (PLL is used in PTPd and PPSi). These algorithms are not defined in PTP since they are implementation specific.[12]

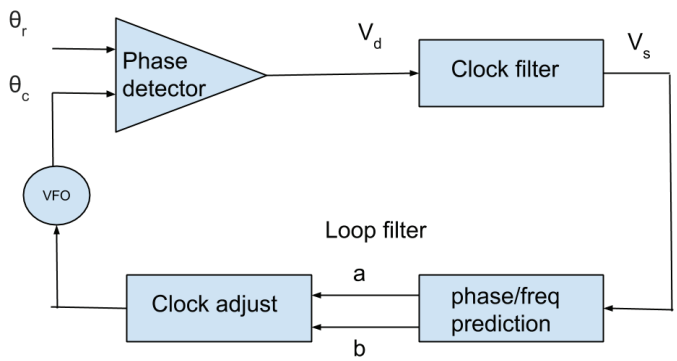


Figure 2.4: PLL.

Figure 2.4 shows a block diagram of the NTP discipline where the kernel operates as a hybrid phase/frequency-lock feedback loop. The timestamps of a reference clock is compared to the timestamps of the system clock which represented as a variable-frequency oscillator (VFO) and the phase difference is determined by the synchronization protocol. In this case NTP is used to produce a raw offset

sample V_d and delay measurement. All offset samples are passed through the clock filter to produce a filtered update V_s, the result is a pulse per second (PPS) signal transitions at intervals of 1 s. These updates are ranging from 16 to 1024 s.[9]

The phase and frequency predictions computed in the loop filter produce a phase adjustment a and frequency adjustment b respectively. The clock adjust method samples these predictors once each second when using daemon discipline, or once each tick interrupt when using kernel discipline, to produce the system clock update V_c. In PLL mode the frequency predictor is not used, however b is computed by integrating the past values of V_s. The phase predictor is the offset amortized over time to minimize the time offset. In FLL mode the phase predictor is not used, however, b is computed directly as the average of the past values of V_s with weight that is determined by poll interval, update interval and Allan intercept (Allan intercept defines the intersection of the jitter and wander whereas the coordinate of this intersection defines the average interval and poll interval). The frequency predictor is computed as a fraction of the current offset, divided by the time to minimize the frequency offset.[9]

According to [9], to demonstrate how clock discipline algorithm works, a client sends message to a server at an interval of μ and a server responds at interval of μ (usually). In PLL, a periodic phase is updated in intervals in order to minimize the time offset and indirect the frequency error. In FLL the frequency is instead periodically updated to minimize the frequency offset and indirect the time offset. A PLL works better in the case where jitter dominates, FLL when clock/oscillator wander dominates the time offset and/or frequency offset.

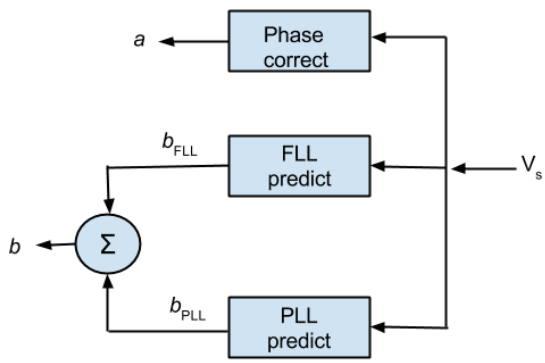


Figure 2.5: PLL/FLL prediction function.

As shown in Figure 2.5, a and b predictors are developed from the phase update V_s where V_s is the phase offset produced by the clock filter. The phase correction (a) represents the value of V_s, b_{FLL} represents the frequency prediction that is computed by taking the average of the past values of V_s. Finally, b_{PLL} represents the frequency prediction that is computed by integrating the past values of V_s. Both b_{FLL} and b_{PLL} are combined with a weight, determined according to the

factors mentioned above to produce the frequency correction (b).

2.4 Time Synchronization Protocols

Time synchronization protocols are used to determine the time offset of a local clock relative to one or more remote clocks. There are several different synchronization protocols in use today, which all uses one general model to time synchronize computer(s)/server(s). The model works by letting the slaves send a request to the master(s) which can also be a grandmaster. The grandmaster has a clock that is synchronized to an external source e.g. GPS receiver. A master has the most precise clock in its domain i.e. is not synchronized to an external reference clock.

The use of server/client terms instead of master/slave may seem equivalent, but the relations between these terms are different. In the server/client relationship, the client always sends a request to a server and the server responds while in master/slave model, the slave is more passive and waits for the master to initiate. Therefore it would be more suitable to use a master/slave model when describing time protocols (NTP, PTP, WR).

2.4.1 Network Time Protocol

NTP was originally created and developed by David L. Mills at University of Delaware and was further developed by some universities and companies. NTP is the most known time synchronization protocol used on the internet. NTP is free of charge and can be downloaded from [13] and can be used in all major OS. NTP can be used both on local networks (LAN) and over the internet. According to [9] NTP consist of three parts: a suit of algorithms, used to process the time values; the protocol, used for exchanging time values between master and a slave; and last the software program called NTP daemon. As of today 5 version of NTP exist, the latest is NTP version 4. The four latest versions are interoperable and can exchange time values and synchronize to another. NTP is a protocol designed to synchronize clocks over the internet. This implies that NTP has higher demand on security and infrastructure compared to PTP, which is design for clock synchronization in LANs.

2.4.1.1 NTP performance

NTP is purely software based, to support as many network devices as possible, which make NTP more sensitive for OS latencies and all timestamps suffers from software latencies. Expected performance of NTP can vary a lot depending on several factors e.g. OS, network traffic and hardware issues. In the scope of this thesis, the main interest is the NTP performance on LANs. Unfortunately no measurements of NTP performance on LANs exist (at least nothing that has been published). According to [13], the performance (accuracy) of NTP over the internet ranges from 5 ms to 100 ms and according to [14] a NTP client synchronized with a GPS signal, accomplish a accuracy of tens of microseconds. NTP accuracy on LAN should range between the two, closer to the first. The performance values stated above is based on NTP running on Linux. The Windows version of NTP

is less accurate; since Windows has less support for time operations and lower timestamp resolution. NTP is suitable for timing systems with a low demand on relative time accuracy, described in section 1.2.[15]

2.4.2 Precision Time Protocol

The Precision Time Protocol (PTP) is an IEEE 1588 standard. The first version was released in 2002 and was known as PTPv1. The second IEEE 1588 version was released in 2008 and known as PTPv2 or IEEE 1588-2008. The latest PTP release (PTPv2) is used by most PTP implementations today. PTP is widely used in industrial control and automation systems and is also used as a base synchronization protocol in WR.[10][12]

PTP is used to synchronize independent clocks on separate nodes in distributed applications that require better synchronization performance than NTP. PTP can provides sub microseconds accuracy and precision in Ethernet-based networks and is used in dedicated networks i.e. does not require a separate network infrastructure. This means that PTP is mixing time information with data when transferring messages, meaning less cost since there is no need for extra cables in this approach.

The main problem with NTP is lack of hardware support. The problem with timestamping frames is the same between PTP and NTP and is solved by timestamping the frames as close to the physical layer of the network as possible. This eliminates latencies caused by the operating system and provides a better estimation for the one-way delay. Thus PTP can provides a few microseconds accuracy in networks with standard components consisting of switches and hubs while achieves better than 100 ns in PTP compliant networks.[17]

PTP does not define algorithms for measuring and adjusting for clock frequency errors since these algorithms are implementation specific. Unlike NTP who uses PLL, FLL or a combination of both of them (hybrid) to adjust the clock. PTP is suitable for synchronizing clocks in local area networks (LAN) since routing PTP messages between networks lowers the time accuracy.[10][12]

2.4.2.1 Definition

PTP defines a master-slave hierarchy where the (grandmaster) server provides a reference time to other clients in the network via a transparent clock (TC) as shown in Figure 2.6. The nodes in the PTP network are called clocks. PTP defines different types of clocks:[12]

- Ordinary Clock (OC): Is a device having only one PTP port. OC can be either master or client.
- Boundary Clock (BC): A device with multiple PTP ports. A BC can be a switch that connects two networks by locking its local oscillator to the master and uses it as a reference time for all the nodes in these networks.
- Transparent Clock (TC): Multi PTP port device such as switches, bridges or routers that forwards PTP messages, whether the these PTP messages are modified by the TC hardware for the residence time that messages arrive and

leave TC. The jitter information from TC is stored in the correctionField in the PTP header.

- Master clock (M): A clock that has the most precise time in its domain.
- Slave Clock (S): A SC can acting as a PTP boundary or ordinary clock depending on the PTP configuration. The frequency and phase recovery in PTP slave is based on the received and requested timestamps from a PTP master which can be either grandmaster clock (GMC) or master boundary clock.
- Slave Only Clock (SOC): A PTP ordinary clock which acting as a slave and cannot become a master. This clock always receives its time from the master.
- Preferred Grandmaster (PG): This clock always acts as a master and cannot be a slave.

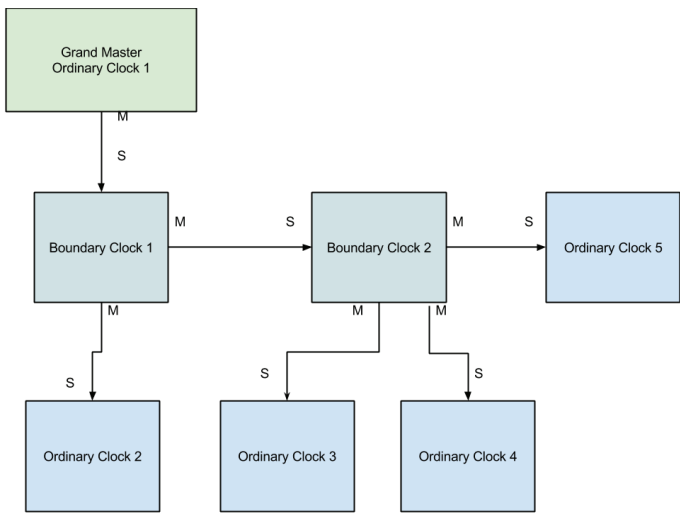


Figure 2.6: A simplified structure of PTP network (M stands for master and S for slave).

PTP selects the role for each node in the network automatically i.e. master or slave defined in the standard as the Best Master Clock (BMC). In the BMC algorithm, each node entering a PTP network compares its local clock to the clocks of all the nodes in network. The node enters automatically master mode if there exist no other nodes in the networks or if all other clocks in that network are worse (less accurate to the reference time). Otherwise the node enters slave mode which means that all the nodes in the network are synchronized to the best available reference time.[10][12]

2.4.2.2 PTP synchronization

The synchronization in PTP is accomplished by exchanging PTP packets between a master and a slave as depicted in Figure 2.7. The slave has to know the values of four timestamps to be able to synchronize to the master. Two kinds of event messages are defined in PTP (*Sync* and *Delay_Req*). The event messages are timestamped both on transmission and on reception. These messages are used for measuring the offset and frequency drift between the master and the slave. The regular messages (*Follow_Up*, *Delay_Resp*, *Announce*, *Signaling* and *Management*) have a timestamp in data field and is used for exchanging timestamps or other parameters.[10][12]

The aim of PTP synchronization is to compute the time offset between the master clock and the local clock of the slave and the round trip delay between them. The propagation time between the slave and its uplink element is called link delay. In one-step mode, t_1 is sent in *Sync* message and is periodically timestamped at the time of transmission from the master to the slave which timestamps it at the time of reception. In two-step mode, t_1 is send in the *Follow_Up* message, which is sent by the master to the slave. The use of one-step mode or two-step mode is depending on the internal implementation of the master. In one-step mode, the *Sync* message is generated in software layer which means less accuracy. In case of two-step mode, the *Follow_Up* message is timestamped in the MAC layer resulting better accuracy compared to SW generated timestamps. The slave sends a *Delay_Req* message that is time stamped both on the master and on the slave. Finally, the master sends *Delay_Resp* message containing t_4. In this way, the slave has a knowledge of all 4 timestamps.[10][12]

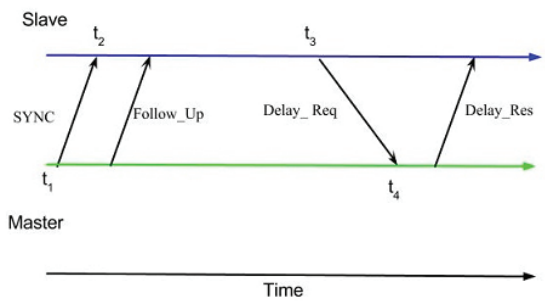


Figure 2.7: PTP synchronization.

$$Round - trip\ delay\,(\delta) = (t_4 - t_1) - (t_3 - t_2) \tag{2.4}$$

$$Time\ offset\,(\Theta) = \frac{1}{2}[(t_2 - t_1) - (t_4 - t_3)] \tag{2.5}$$

An *Announce* message is only generated in the nodes that enter master mode and sent periodically providing information about the master's clock quality and

the data set required by BMC algorithm. *Signaling* message is used to negotiate optional services and *Management* message is used for updating the data set.[10][12]

2.4.2.3 Best master clock (BMC)

The Best Master Clock (BMC) algorithm determines which clock to be the master (has the highest quality properties compared with the other clocks within the PTP domain) and all other clocks to become slaves in the PTP domain. The decision is based on comparing the data sets related to each PTP device within the PTP domain. These data sets include e.g. priority, class, accuracy, variance and distance to the neighboring PTP ports in the entire network. The BMC algorithm is constantly active so that the state of each PTP device enters or leaves the entire network dynamically. If the master clock is removed from the network or if the current master no longer has the highest accuracy clock or in case of failure, the BMC process should redefine a new clock to take the role of the current master and adjusts all other clocks for the new BMC.[17]

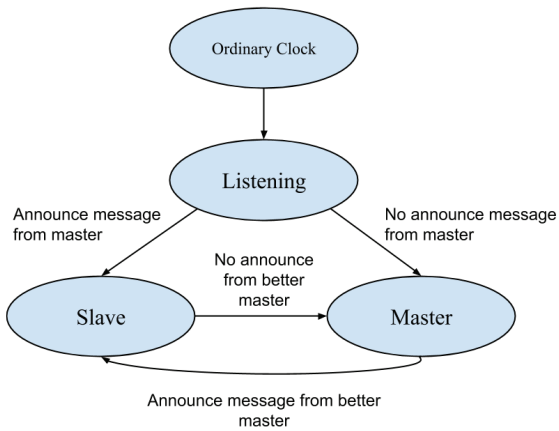


Figure 2.8: The state transition graph for an ordinary clock in BMC.

To understand how BMC works, an example is illustrated in Figure 2.8. An ordinary clock is designed so that it can be a slave or master. When PTP starts, the clock enters the listening mode i.e. listens after *announce* message on the PTP multicast address. The *announce* message contains information about the clock that sent it. The ordinary clock compares its data sets with the information provided in the *announce* message within the *announce* timeout interval. If the *announce* message comes from a better clock, the ordinary clock enters a slave mode. If the *announce* message comes from a worse clock (i.e. having a worse data set in comparison), the ordinary clock enters master mode.[17]

2.4.2.4 PTP performance

PTP was designed to be used in industrial control, automation systems and in measurements that require microsecond accuracy and very low jitter. This is not enough for e.g. particle accelerator, which requires much higher synchronization accuracy. PTP is suitable in local area networks (LAN) and not as much in WAN (as the case for NTP), since forwarding PTP messages through routers and switches increase the jitter which affect the accuracy.[18]

2.5 White Rabbit

This section presents a short introduction to the White Rabbit (WR) project including some basic on General Machine Timing (GMT) and its drawbacks, also presents WR design goals, introduces some theoretical considerations about the requirements of CERN and GSI and an explanation for how WR is intending to meet these requirements. The synchronization scheme in WR and a description of the central components in a WR system structure is presented.

2.5.1 Introduction

Large control systems have many distributed nodes that all needs to be synchronized. The current control and timing system at CERN (General Machine Timing) have many drawbacks; low speed (500 kbps), unidirectional communication and complicated maintenance when using different timing systems in accelerators. Because of these drawbacks of the current timing system (GMT), CERN started thinking about a suitable successor for the timing system of the Large Hadron Collider (LHC) injectors in 2006. CERN is consisting of six accelerators and LHC is the biggest (27 km long) with thousands of devices that serve the accelerators, all of which needs to be precisely synchronized.[20]

At the same time the GSI Laboratory in Germany had already started a brainstorming about the timing system for the FAIR facility. Both of CERN and GSI have the same requirements regarding the need for high bandwidth, full-duplex communication links and the choice of Ethernet for the physical layer, which were important requirements for both CERN and GSI on their timing systems. The White Rabbit project was the solution to solve these problems and was started by CERN. GSI joined the project later and now the WR project is a collaboration of many institutes and companies around the world.[4] This project is both open hardware and open software which means that anyone can download and use WR facilities.

An extension has been done to the standard Ethernet to simplify the node synchronization in the control systems. The new feature added to Ethernet is $Synchronousmode$ ($Sync - E$) where all the nodes in the network use a common clock generated by the master. Then they were encoded in the Ethernet carrier and recovered by using a Phased Locked Loop (PLL) for precise time and frequency transfer. Deterministic routing was done by using PTP where packet transmission delay between two nodes will never exceed a certain boundary.

2.5.2 Synchronization scheme

The aim of WR is to synchronize over 1000 nodes with sub nanoseconds accuracy and picoseconds precision of synchronization over fiber lengths up to 10km. The data network has to be deterministic with very low transmission latency. In order to achieve the above requirements, the WR have to mix three technologies, PTP (IEEE 1558), Sync-E and DDMTD phase tracking, so that WR easily can compensate for environmental fluctuations.[19] In WR, the master provides its clock either directly or via Sync-E compatible switches to the WR nodes. This process is accomplished by defining a timing hierarchy, by naming one of the switch ports as uplink and naming all other ports as downlink. The first switch in the hierarchy is called a grandmaster which has the highest clock accuracy referenced to an external source, such as GPS. The downlink ports are then connected directly to a final node or to an uplink port of another switch resulting in a tree of switches, whereas all switch clocks are derived from the clock of a grandmaster as illustrated in Figure 2.9

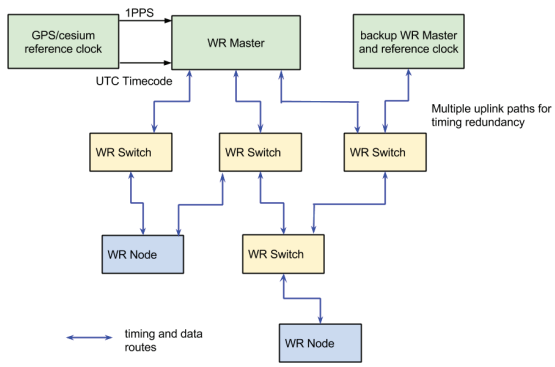


Figure 2.9: WR synchronization hierarchy.

After all timestamps have been transmitted and recovered, there is a problem of compensating for transmission delay from electronics in WR switches and nodes and from propagation delays on the fibers. Delays that are coming from switches and nodes have to first be fixed by manual or automatic calibration, whereas propagation delay on the fibers due to thermal effects is solved by using PTP two-step mode. WR has to use continuous phase measurements instead of direct time stamping since measuring clock phases is easier than measuring time intervals between pulses according to.[4]

The phase measurement process used in WR is called Dual Mixer Time Difference (DMTD) scheme. The aim of using DMTD is to compare two clocks to third clock whose frequency which has very slightly offset to the frequency of these clocks. An analog structure of DMTD is presented in Figure 2.13 [24]. Let's assume that the input clocks $x(t)$ and $y(t)$ are identical in amplitude and frequencies ($= f_{clk}$) and the phases are Θ_x and Θ_y respectively. Both of clocks are multiplied

by mixers with the oscillator signal $z(t)$ with frequency f_{offset} and phase Θ_{offset} as:

$$x(t)z(t) = \cos(2\pi t f_{clk} + \Theta_x)\cos(2\pi t f_{offset} + \Theta_{offset}) = \frac{1}{2}\cos(2\pi t\,(f_{clk} + f_{offset}) + \Theta_x + \Theta_{offset}) + \frac{1}{2}\cos(2\pi t\,(f_{clk} - f_{offset}) + \Theta_x - \Theta_{offset}) \tag{2.6}$$

The result of equation 2.6 is two clock signals, one with high frequency and the other with low frequency. The high frequency clock is removed by low pass filter and the result is only one clock with low frequency. The mixing process done here only affects the frequency of the signal clocks but the phase difference between the two signals remains the same. If the clock offset is very close to the frequency of the input clock f_{clk}, then the phase difference can be measured with higher accuracy using a simple counter. [24]

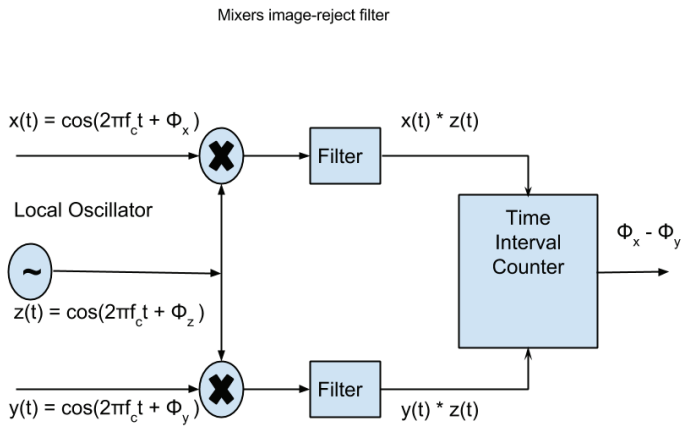


Figure 2.10: Structure of a analog DMTD.

According to [24], the analog DMTD used above provides very accurate phase measurements, but the cost of external mixers and filters is high especially in t.ex. WR switch. However, the analog DMTD is transformed to digital DMTD as shown in Figure 2.11.

2.5.3 White Rabbit hardware

In this section, a description of the White Rabbit Ethernet Switch is introduced; also some information about Simple PCIe FMC carrier (SPEC) and finally White Rabbit PTP Core are presented.

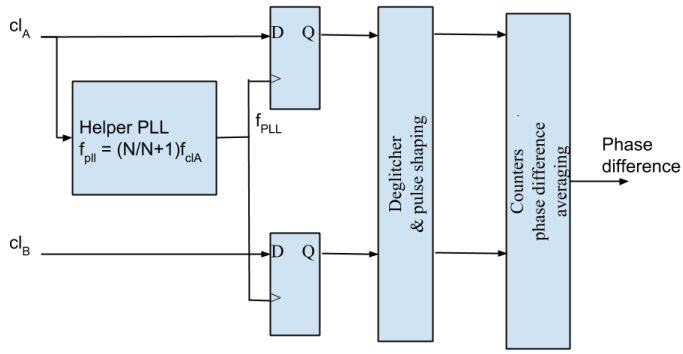


Figure 2.11: Structure of a digital DMTD.

2.5.3.1 White Rabbit switch

The switch is the core in a White Rabbit network and is implementing the actual Ethernet switch (IEEE802.1D Bridge) with WR extensions, including low level timing functions such as PLL, DMTD and fine timestamping. The extensions are available only if the connected device is WR aware. If the connected device is non-WR aware, the switch plays as a standard switch.

2.5.3.2 Simple PCIe FMC carrier (SPEC)

According to [21], the PCIe FMC carrier holds one FMC card and an SFP connector where the PCIe side has a 4-lane interface, and the field-programmable gate array (FPGA) Mezzanine Card (FMC) mezzanine slot uses a low-pin count connector. This card is optimized in cost and supports the White Rabbit timing and control network. A prototype of SPEC is shown in Figure 2.12.

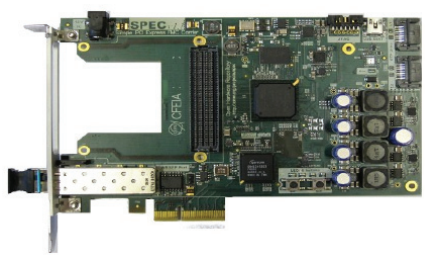


Figure 2.12: SPEC prototype, credit Cosylab.

2.5.3.3 White Rabbit PTP Core (WRPC)

WRPC is an implementation of Ethernet MAC for providing precise timing of synchronization in timing control systems, when sending and receiving normal Ethernet frames between user-defined HDL modules and a physical medium. WRPC implements WR to achieve sub nanoseconds accuracy and picoseconds precision of synchronization. WRPC can be a master or a grandmaster i.e. a WR master referenced to 1PPS and 10 MHZ clock signal and distributes its clock to other WR nodes in the entire network, or as a slave that synchronizes its local clock to the WR master clock. Figure 2.16 shows the WRPC external interfaces.[22]

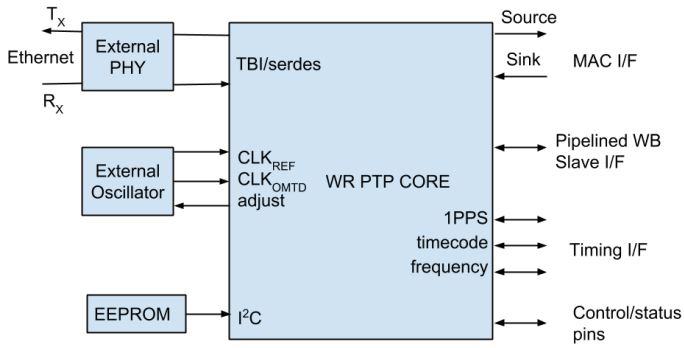


Figure 2.13: WRPC external interface.

2.5.4 WR performance

The WR performance depends on the WR network setup such as how many nodes and switches are available in the network. WR aims to provide sub nanoseconds accuracy and and picoseconds precision of synchronization depending on the network setup. An advantage is that WR can compensate for propagation delay on the fiber (due to variation of temperature on fibers).[35]

Chapter 3
Synchronization with PPSi

In this chapter the suggested solution to satisfy both the high demand and low demand on relative time accuracy is presented. First, a small introduction and an explanation why PPSi was chosen. Then, a detailed description of the PPSi architecture and a comparison between the Cygwin version and the Linux version of PPSi.

3.1 Introduction

PTP Ported to Silicon (PPSi) is a portable PTP implementation, originally developed for the White Rabbit project but is also suitable for different kinds of networks. PPSi is licensed under the GNU Lesser General Public License, meaning it is allowed to copy, distribute and modify the PPSi code with or without fee.[25][29]

PPSi is the resulting effort to provide full support for a WRPTP (White Rabbit PTP) extension. In order to make PPSi portable it is required that the PPSi daemon can work in different environments, hosted environments (e.g. Linux computers) or freestanding environments (e.g. WR nodes with no operating system) and is the reasons it is written in C. To accommodate this demand, the daemon uses a modular design that separates PTP from the run time specific system calls. This design approach allows easier porting for other types of environments. A mayor part of this thesis has been to develop a Cygwin extension for PPSi.

3.1.1 Why PPSi

This section describes why PPSi (PTP ported to silicon) was chosen to be the solution of the problems mentioned in section 1.2 Scope, why PPSi is a good solution in our case and what problem PPSi has. Two problems are considered in this master thesis to be solved. The first problem is to get a PC's internal clock synchronized in a WR-network (the computer has no WR hardware and can be a Linux or Windows computer). The second problem is to get a PC's internal clock synchronized in time in a non-WR network, e.g. a master server/computer is first synchronized with NTP and then let the other nodes/computers in the network to be synchronized to the master (the computers has no WR hardware and can be Linux or Windows computers).

There exists a couple of solutions that addresses one or both of the problems (NTP and several PTP implementations), but PPSi was chosen as the solution. PPSi can achieve sub-microseconds accuracy of synchronization and it is possible to get up to sub-nanoseconds accuracy with hardware assistance, according to one of it's developer Alessandro Rubini. The WR project uses WR frames that are compatible with a normal PTP frame, and therefore PTP will work fine for both of the problems stated above. There are numerous different PTP implementations but none is WR compatible. However, there exist no PTP implementation (free of charge) that works for both Linux and Windows computers (only the one or the other) and is compatible with WR hardware. Before this thesis, PPSi only worked for Linux, WR computers (computer with WR hardware) and WR switches and did not worked for Windows.

Another good reason to choose PPSi is that it is developed for WR, to work with or without WR hardware. This approach will make it easier to add WR hardware in a non-WR network in the future, without changing the software.

3.2 PPSi overview

The PTP protocol code is based on code from the PTPd project which is also an Open Source PTP implementation under GNU LGPL license. Since WR is based on the Ethernet standard, so is also PPSi. PPSi can send packets over UDP or raw Ethernet (where the destination and source address is a MAC address), depending on the architecture specific code (will be discussed later). PPSi can be used as Ordinary Clock (OC) or as a Boundary Clock (BC). The user of PPSi can configure PPSi for different working modes for each PTP port (a PTP port is a virtual input for receiving PTP packets, defined IEEE 1588 protocol), PPSi supports several PTP ports on a single physical port.[25]

There are also several diagnostic options in PPSi. Which can be configured in different levels (providing more or less data) on each port and displays data regarding: state machine, servo loop and frame I/O and can be changed at run time. This feature can be very useful in circumstance where network sniffers are unavailable. Another important build-time option is support for extensions, i.e. WRPTP.[25][26]

In order to get several slaves to be synchronized to one master, PPSi initiates a multicast group, this enables several slaves to synchronize to one master by joining the multicast group. Currently PPSi does not support UDP over IPv6, but a patch to enable UDP over IPv6 is under development.[26]

3.3 General Architecture

The PPSi design goal is to be self-contained and therefore uses Makefile to compile the PPSi code. The top-level Makefile is used to build the object file ppsi.o, which looks like a library for architecture specific code. In order to achieve portability, the core PPSi code has no interaction with the outside world; instead this is up to the architecture specific part of PPSi to provide.[25]

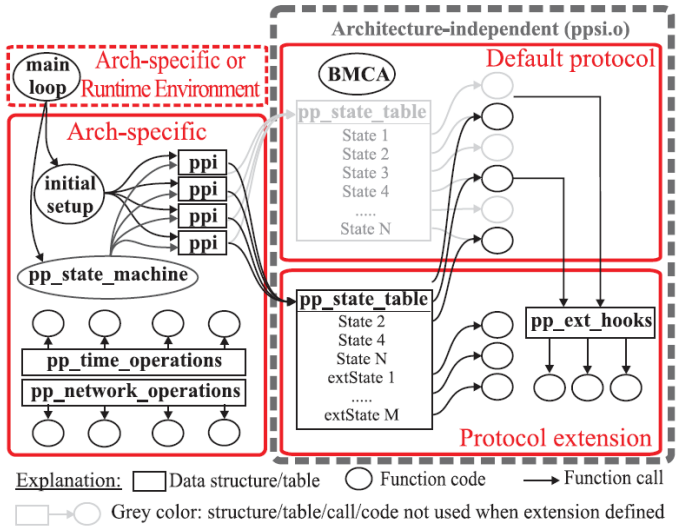


Figure 3.1: PPSi architectur. Credit the PPSi developers.

The PPSi code can be divided in 3 parts (the filled red rectangles in Figure 3.1): The architecture specific code, the architecture-independent default protocol code and the architecture-independent protocol extension code. This architectural split conforms to the layering description in IEEE 1588 standard. The media-independent layer (default protocol in PPSi), the media-dependent (arch-specific in PPSi) and finally option- and profile-specific operation (protocol extension in PPSi).

In order to build PPSi, a configuration file is needed. The configuration file specifies what kind of architecture and possibly extension, to be build/used. This is performed using Kconfig, which is a tool derived from the Linux kernel and can be configured in an interactively or in a non-interactively way. The non-interactive option is achieved by providing pre-build configuration files (called .config) one for each architecture (with extension disabled). The interactive way is graphical interface in the command prompt.

Configuring PPSi at run-time (e.g. enable diagnostic on each port) is achieved by feeding "configuration strings" through the run time environment to PPSi.[26]

3.4 PPSi Internals

The goal of this section is to present deeper look at the PPSi code. The core of PPSi, the state machine is thoroughly explained as well as the time-, network operation, profiles and extensions.

3.4.1 Default Protocol

PTP protocol is implemented in the "default protocol" part of the PPSi code. The most interesting part is the Best Master Clock Algorithm (BMCA) which is the algorithm used to decide what state a PTP port should enter. Another interesting fact is that the servo/PLL implemented in PPSi is based on NTP's PLL algorithm, because the PTP specification does not specify a servo algorithm (for clarification the servo is the PLL in PPSi and is responsible for calculating the clock frequency offset).

```
struct pp_instance {
    int state;
    int next_state, next_delay, is_new_state; /* set by state processing */
    void *arch_data;
    void *ext_data;
    unsigned long d_flags;        /* diagnostics, ppi-specific flags */
    unsigned char flags,    role,   proto;    /* protocol flags */


    /* Operations that may be different in each instance */
    struct pp_network_operations *n_ops;
    struct pp_time_operations *t_ops;


    /* Times, for the various offset computations */
    TimeInternal t1, t2, t3, t4;                /* *the* stamps */
    TimeInternal cField;                /* transp. clocks */
    TimeInternal last_rcv_time, last_snt_time;    /* two temporaries */

};
```

Figure 3.2: Data structure for *pp_instance* (only a collection of the variables is presented).

Every configured PTP port runs a state machine, defined by an *pp_instance* object (called ppi), which runs a *pp_state_table*[] structure shown in Figure 3.3. The content of the state table depends on how PPSi is built (default or profile-extended, a profile-extended can also refer to the default state table). As Figure 3.2 presents, most of the important variables are defined in *pp_instance* structure and contains information about the current state, next state, diagnostic as well as protocol flags and timestamp.[25]

The state machine in PPSi is purely network driven and has nothing PTP-specific in the engine. The BMC algorithm is used to decide in which state each port should enter.

Below is a list of all states a PTP port can enter in PPSi:

- Disabled: indicating the post is disabled and no message can be sent on this port.[12]

```
struct pp_state_table_item {
    int state;
    char *name;
    pp_action *f1;
};
```

Figure 3.3: The state table structure.

- Faulty: in PPSi this state is for troubleshooting purposes and prints an error message before entering initializing state.
- Initializing: in this state all network, hardware and data sets are initiated and the clock identity is derived from the NIC card's (Network Interface card) MAC-address.[12]
- Listening: is a temporary state at startup, before deciding to enter slave or master state.
- Master: in the master state the port sends announce message (announcing that this port is a master) as well as listening for sync and request messages from slaves.
- Passive: the passive state, when a port is neither slave nor master and only responses on incoming messages and sending delay request.[12]
- Slave: in this state the PTP port tries to synchronize to a master.
- Uncalibrated: PTP port enters this state when preparing to synchronize to a master and cannot enter master state, this state is mainly used for initializing servo.[12]
- Pre-master: not yet implemented in PPSi since this state is used in BC situations.

3.4.2 Architecture specific

The architecture dependent code defines network and time operations and provides the operations to each associate PTP port (called ppi). The time and network operation structure is defined in *pp_network_operation* and *pp_time_operation*. In most architecture (especially the hosted) a main loop is provided, which runs the PPSi program. In most freestanding environments, a main loop is not provided, because in those environments all object files are linked together with an externally main loop.[25]

The main function of the architecture specific code refers to a single entry point called *pp_state_machine*(), which refers to the default protocol code. *pp_state_machine*() is a procedure with a network frame as argument and operates synchronously and returns immediately. If available, the procedure returns a wait value, indicating when to call the procedure again.

```
struct pp_time_operations unix_time_ops = {
    .get = unix_time_get,
    .set = unix_time_set,
    .adjust = unix_time_adjust,
    .adjust_offset = unix_time_adjust_offset,
    .adjust_freq = unix_time_adjust_freq,
    .init_servo = unix_time_init_servo,
    .calc_timeout = unix_calc_timeout,
};
```

Figure 3.4: The time operation structure.

In each *pp_instance* object a time operation structure is instantiated, the structure points to all time-related related interaction between PPSi protocol and the run time environment. The structure is presented in Figure 3.4.[25]

The *get*() method returns the time of the system clock. The *adjust*() method is used for small adjustments on time in scale of nanoseconds and frequency (in part per billion), depending on what the underlying architecture supports. In for example Windows (and therefore Cygwin) there is no support for adjusting the clock frequency. The *set*() method is only used for time jumps, when the time offset is too large for a time adjustments. The initialize servo method is used to return the current applied frequency correction, e.g. if the system is already in process of adjusting the clock frequency, PPSi can take that into account.[25]

Timeouts are software based in PPSi and is implemented as inline functions based on monotonic time in millisecond resolution. Since *get*() does not return monotonic time we need a special function for timeouts. The timestamps in PPSi are stored in Time Internal structures which has a high precision field (nano scale). This field is only used by WR profile.[25]

```
struct pp_network_operations unix_net_ops = {
    .init = unix_net_init,
    .exit = unix_net_exit,
    .recv = unix_net_recv,
    .send = unix_net_send,
    .check_packet = unix_net_check_packet,
};
```

Figure 3.5: The network operation structure.

Like the time operations, each PTP port also instantiate a network structure. According to [25] "In case of a device having more physical ports (e.g. a BC), each port has it's own driver, thus handling different hardware interfaces on the same device". In a hosted Linux environment there a possible to tell the kernel to timestamp incoming frames (discussed later).[25]

In the initialization phase, each PTP port is using information in the instance

of the network operation structure. Additionally, the multicast group is also initialized in this phase. Each architecture implements how PTP packets should be sent, either over UDP and/or raw Ethernet, depending what the architecture support. Each architecture can pre-set what the default should be. If both UDP and raw Ethernet is supported, a configuration can set up two PTP ports on a single physical interface.[25]

The *send*() and *recv*() operations has two responsibilities: firstly receiving and sending frames, secondly to collect timestamps for each operation. These timestamps are later used in the PTP calculations. The *check_packet*() method is used to check for frame arrival. Finally, the *exit*() method closes multicast and the socket.[25]

3.4.3 Profiles and extensions

The "protocol extension" part of PPSi contains the PPSi extensions code, currently only WRPTP extension is available.

```
struct pp_ext_hooks {
    int (*init) (struct pp_instance *ppg, unsigned char *pkt, int plen);
    int (*open) (struct pp_globals *ppi, struct pp_runtime_opts *rt_opts);
    int (*close) (struct pp_globals *ppg);
    int (*listening) (struct pp_instance *ppi, unsigned char *pkt, int plen);
    int (*master_msg) (struct pp_instance *ppi, unsigned char *pkt, int plen, int msgtype);
    int (*new_slave) (struct pp_instance *ppi, unsigned char *pkt, int plen);
    int (*handle_resp) (struct pp_instance *ppi);
    void (*s1) (struct pp_instance *ppi, MsgHeader *hdr, MsgAnnounce *ann);
    int (*execute_slave) (struct pp_instance *ppi);
    void (*handle_announce) (struct pp_instance *ppi);
    int (*handle_followup) (struct pp_instance *ppi, TimeInternal *orig, TimeInternal
*correction_field);
    int (*pack_announce) (struct pp_instance *ppi);
    void (*unpack_announce) (void *buf, MsgAnnounce *ann);
};
```

Figure 3.6: Profile and extensions structure.

In PPSi, profiles and extensions are managed by protocol hooks, to modify the state machine table. The available hooks are presented in Figure 3.6. The purpose of profiles is for example to define modified state table, specific action required to establish communication with a peer. As an example, in WR it is required to perform specific action, in order to establish a WR link between two WRPTP daemons after a successful *announce* handshake.[25]

The available hooks fulfills three types of roles:[25]

- Managing extension: *init*(), *open*() and *close*().
- Extending protocols: these hooks are responsible for changing the protocol behavior, these are the hooks *listening*() and *handle_followup*().

- The third role is to handle packet for profile-specific handshake, these are the *pack_announce*() and *unpack_announce*() methods.

The structure can be extended to support new extensions, which is easier since profiles and extensions are cut off from the default protocol.[25]

3.5 Supported architectures

PPSi was originally developed for WR networks and hosted Linux computers. As of today PPSi can be runned on several other architectures. In this section all current supported architectures are presented.

- arch-unix: this is the default configuration and should be used when building PPSi for hosted Unix based environments. This architecture can run as an OC or a BC.
- arch-cygwin: this is the Cygwin (Windows) configuration and should be used when building PPSi for hosted Windows with Cygwin installed based environments. This architecture can run as an OC clock.
- arch-wrs: the White Rabbit switch build, which includes special hardware for time stamping and phase detection. This architecture should be used for building WR switches, include special software and has the possibility to fall back on default PTP when running in non-WR environment.
- arch-wrpc: the build for White Rabbit PTP Core architectures. The run time environment is a soft-core CPU running within an FPGA.
- arch-bare-i386 and arch-bare-x86: these two are for freestanding type environments and "builds a Linux process that does not rely on standard libraries: both process startup and the few system calls are implemented in assembly language by the ports themselves." [25]
- arch-sim: this was made for the purpose of testing PPSi servo (non-WR). PPSi built using this architecture, PPSi implements two PTP instances that communicate: one master, one slave. The two instance communicate through a software-only channel and the perceived flows at a faster pace.[25]

3.6 Cygwin

In this section a brief explanation of what Cygwin is and an explanation of why a Cygwin extension instead of a Windows extension was made.

3.6.1 What is Cygwin?

Cygwin is free software under the GNU GPLv3 license developed by Red Hat (formerly Cygnus Solutions) and is a distribution of popular Open Source tools like GNU, BSD and MinGW tools for Windows. Cygwin consists of two parts: the most important part in scope of this thesis is a dynamic-link library (DLL) as an

application programming interface (API) providing substantial POSIX functionality. POSIX is family of standards that defines the API, the user command line (ls, pwd, clear etc...) and also scripting interface. POSIX is specified by IEEE Computer Society for preserving software application compatibility between operating system (specifically UNIX variants), making it possible to build (and run) Linux programs on Windows.[30] [32] POSIX is not an operative system (OS), instead POSIX acts as an interface between the application and the library (the library contains function calls to the underlying OS). POSIX defines how the communications to the library should be and how the library and the OS should respond.[31]

Cygwin is expected to run on all modern versions of Windows XP, Windows Server 2003 up to the latest version of Windows, Windows 8.1 (Windows 10) and Windows Server 2012R2 and can be run in both 32-bit and 64-bit Windows.[30]

Most of today's operating system is compliant with POSIX and the new version of Windows (Windows 8.1, Windows Server 2012) has added more support for POSIX.[33]

3.6.2 Why Cygwin?

There are a couple of reasons were chosen to build a Cygwin extension for PPSi and not a plain Windows extension. The former reason to build a Cygwin patch is due to the fact that the original PPSi program was written for Linux in POSIX which is not supported in Windows. This means that there are three choices:

- Rewrite most of the PPSi source code to fit Windows.
- Use a POSIX emulator (Cygwin) on Windows and a write PPSi extension for the emulator.
- Run Linux in a virtual machine.

We choose to use Cygwin and write a PPSi extension and leave rewrite PPSi for Windows as future work. The reason why Cygwin was chosen over a virtual machine is because the performance was expected to be worse (at least not better) using a virtual machine compared to using Cygwin.

	Advantage	Disadvantage
Rewrite PPSi source code.	This solution is likely to have best performance compared to the other two. Since, in this solution, PPSi runs directly on the application layer, i.e. less asymmetric latency.	Time consuming, rewriting PPSi source code will take a lot more time compared to the other two. Another downside with this solution is that it will create two versions of PPSi (one for Windows and one for Linux system), which is bad for the future development of PPSi.
Using a POSIX emulator and writing a PPSi extension.	This solution will preserve the portability of PPSi, therefore future development is not a problem. Far less time consuming writing a PPSi extension, than rewriting the source code.	Since Cygwin will run as compatibility layer around PPSi, this will impact the performance, i.e. PPSi will be slower in Cygwin and adding another application layer may give more asymmetric delay.
Run virtual machine.	No changes to PPSi are needed.	This approach is likely to have worse performance, despite the host and guest OS share the same hardware clock, the actual value of the clock is different (the clock value is managed by the OS) see section 2.2 Computer Clock. Running PPSi in a virtual machine will only synchronize the clock in the virtual machine. The host has to then be synchronized with the virtual machine, another way of introducing asymmetric delay.

Table 3.1: Comparison of different approaches to run PPSi in Windows

3.7 PPSi implementation in Cygwin

In this section, our approach of writing Cygwin extension is presented, as well as the major differences between the Cygwin version and the UNIX version of PPSi are discussed.

3.7.1 Approach

Since Cygwin provides a very complete POSIX functionalities as well as GNU library, much of the UNIX code could be re-used. However, all time and network operations have to be rewritten with what is supported in Cygwin (and Windows).

3.7.2 PPSi in Cygwin versus UNIX

The major difference between the Cygwin and UNIX is the time operation code. The *get*() and *set*() methods are almost identical, but the *init_servo*() method in Cygwin returns 0, i.e. no frequency adjustment is currently being applied and the frequency offset is 0. UNIX has a function called *adjtimex*(), which among several thing, returns the current frequency offset. Adjusting the clock frequency is not supported at all in Windows and therefore not in Cygwin. NTP source code was checked (in their Windows service) to see if they have a solution for adjusting the clock frequency in Windows, but no solution was found. According to [34], if the kernel has support to "discipline the clock frequency", then NTP provides a feature for adjusting the clock frequency offsets. Despite having support for small clock adjustments in Windows (which temporarily speeds up or down the clock until the clock adjustment has been met) it is not supported in Cygwin. Instead the *set*() clock method is used to set the clock which is less than optimal, more on this in the chapter 5. Finally, the *calc_timout*() function is identical to the UNIX version.[8] [28]

Another difference is how they receive packet timestamp is generated in UNIX; the receive packet timestamp are generated at the kernel. This is achieved by setting the $SO\_TIMSTAMP$ options on the socket at start up. This will tell the kernel to timestamp incoming packets, the timestamps can later be collected through packet control messages. This feature is not supported in Windows; instead a software timestamp is generated for incoming packet.

There is also some differences in the network operation compared to UNIX. In Cygwin only UDP packets is supported, because raw Ethernet frames is not supported in Windows. Ignoring the differences state above the code is more or less identical between the Cygwin and the UNIX version of PPSi.

3.8 Applications for PPSi

The main application of PPSi is with WR, which is used to provide precise time accuracy for accelerator facilities. PPSi applications provides full support for WRPTP extension and can be used in distributed data control systems to synchronize diverse environments, both hosted (Linux- and Windows with Cygwin

install based WR device) and freestanding such as WR end node without operating system by using a modular design that separates PTP code from the required interaction with a specific environment taking the benefits from WRPTP and WR networks. PPSi can be used as a standalone module that provides precise time synchronization for both standard exiting (non-WR) devices or as a WR compatible devices.[25]

Chapter 4
Result

This chapter shows the test result of the PPSi software in three different scenarios, to test PPSi in different aspect. The calculated performance and verification of the three scenarios are shown below.

4.1 Testing PPSi

In order to evaluate the performance of PPSi two different lab setups were chosen to represent our objective (synchronization in a non-WR or/and WR network) and the third lab setup was made to test WR, so that the software and the hardware approaches for time synchronization could be compared. In order to measure the performance, two clock pulses (1PPS) from the master clock and the slave clock are compared in an oscilloscope. The clock pulse delay or clock pulse deviation is measured using the oscilloscope and the measurements are used to calculate the mean error and standard deviation of the relative time difference between two clock pulses. In each test scenario the setup was running in approximately one hour before measuring, so that the systems had time to stabilize.

4.1.1 SPEC to PPSi lab-setup

In this setup as shown in Figure 4.1 the performance of PPSi was measured by synchronizing a 32-bit Linux (Debian) PC to a WR SPEC card. The SPEC board was configured as a master was inserted into the PCIe slot of another PC. The PC in the test setup was configured as a slave running PPSi. The SPEC board was connected to a media converter (which converts an optical signal to an electrical signal, i.e. fiber to twisted pair) with a 1 meter fiber cable; the slave was connected into the media converter with either a 1 or 50 meter Ethernet cable respectively, to see if the cable length has a significant impact on the synchronization error.

Pre-built binaries were installed in order to build the White Rabbit PTP Core (WRPC), downloaded them to the FPGA and made the configuration of the WRPC. The SPEC was configured as a master and would now send it's timing information to the slave. The 1PPS (clock pulse generated every time the clock value is increased by one second) output of the SPEC board was connected to the CH 1 of the oscilloscope via coaxial cable. The slave were connected to the oscilloscope by a probe connected to the computer's parallel port, in order to

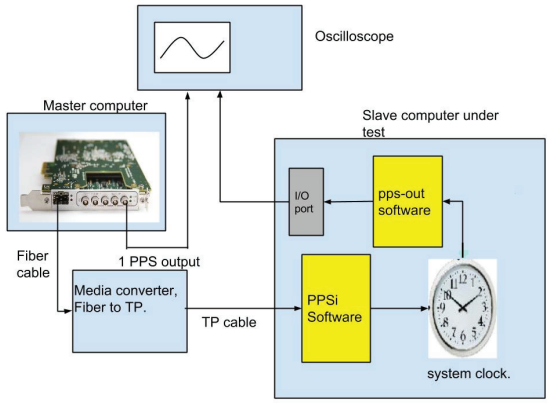


Figure 4.1: Lab setup, SPEC master to PPSi slave

provide a 1PPS signal by running a small software program called $pps - out.c$ was used, available in the PPSi framework. These two signals were compared and the relative time difference was measured. A WRPC can only (currently) send WR PTP frames using raw Ethernet frames, therefore directly connecting a Windows PC running PPSi to a SPEC card will not work (will be discussed in chapter 5).

An interesting note, is that when a media converter (optical/electrical) is used WR is disabled, the reason is there is no direct fiber connected between the WR master and the slave. WR uses information gather from the fiber in order to track the master clock to get a high accuracy timestamps. Instead the SPEC will run normal PTP and the synchronization accuracy will not be high (scale of nanoseconds).

4.1.2 PPSi to PPSi lab-setup

In this setup as shown in Figure 4.2 the performance of PPSi was measured by synchronizing two desktop computers running 32-bit Linux (Debian) operating system and running PPSi. These two PCs were connected by a 1 or 50 meter long Ethernet cable respectively. One PC was configured as a master and the other as slave. Both the master and the slave were connected to the oscilloscope. The 1PPS output of the master was connected to the CH 1 of the oscilloscope and the 1PPS output of the slave was connected to the CH 2 of the oscilloscope. These two signals were compared and the relative time offset was measured and analyzed.

4.1.3 SPEC to SPEC lab-setup

The setup as shown in Figure 4.3 was performed in order to compare how PPSi by software adds up against White Rabbit. Two SPEC cards configured as master and slave was used, a 1 meter fiber cable was used to connect the SPEC cards.

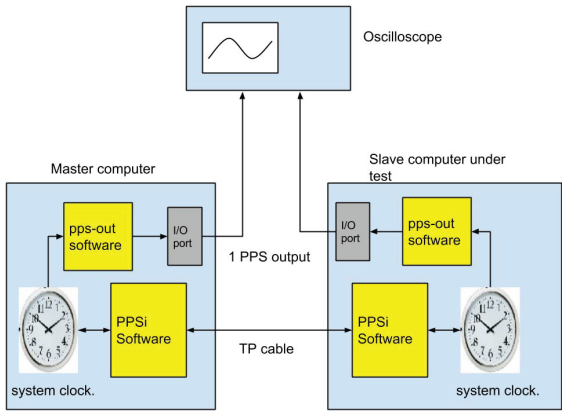


Figure 4.2: Lab setup, PPSi master to PPSi slave

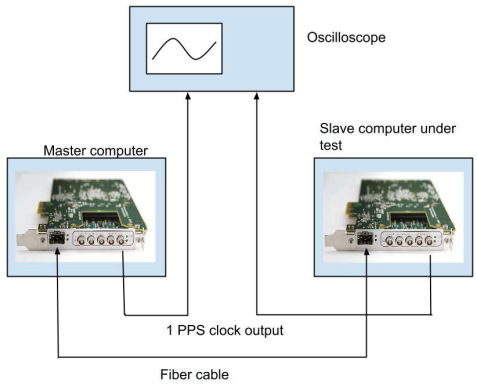


Figure 4.3: Lab setup, SPEC master to SPEC slave

The SPEC cards were connected to the oscilloscope with coaxial cables via the DIO FMC board.

4.2 Measurement and verification

The results are shown in table 4.1 and were calculated using Matlab.

Lab-setup	Mean delay	Standard deviation
WR SPEC card (master) to computer (slave), 50 meter TP cable.	47.5 μs	16.9 μs
WR SPEC card (master) to computer (slave), 1 meter TP cable.	48.2 μs	15.7 μs
PPSi master to computer (slave) 1 meter TP cable.	2.6 μs	13.0 μs
PPSi master to computer (slave) via 50 meter TP cable.	3.6 μs	10.6 μs
WR SPEC (master) to WR SPEC (slave), 1 meter fiber cable.	15.9 ns	2.0 ns

Table 4.1: The performance of PPSi in the three different scenarios

The resolution of the relative time measurements must be taken into account in order to ensure the authenticity of the calculation in table 4.1. According to the LeCroy manual the resolution of the cursor used for measuring the relative time difference between the two clock signals can be $\pm 0.05\%$ full scale for unexpected traces. Therefore only one decimal digit is used in the calculated values in Table 4.1.

Figure 4.4 is an example of how the synchronization deviation/error was measured between the master and slave in each of the three different demonstrations mentioned above.

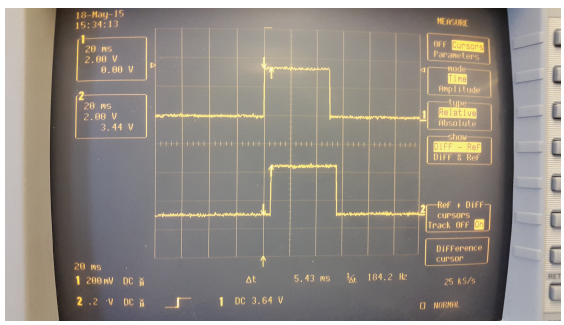


Figure 4.4: Clock pulse deviation: PPSi master to PPSi slave

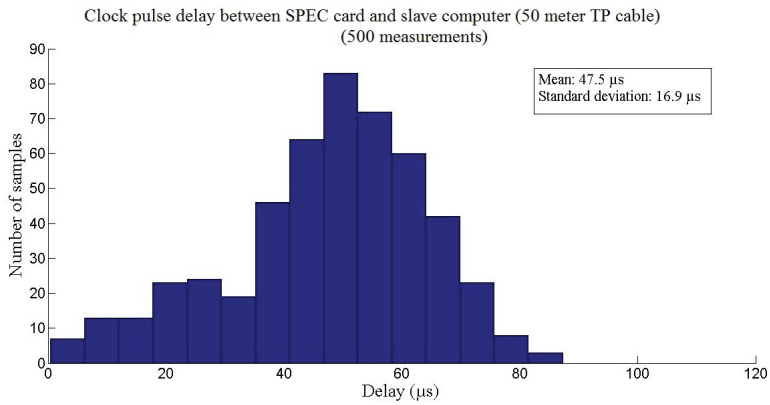


Figure 4.5: Clock pulse delay between SPEC card and slave computer (50 meter TP cable)

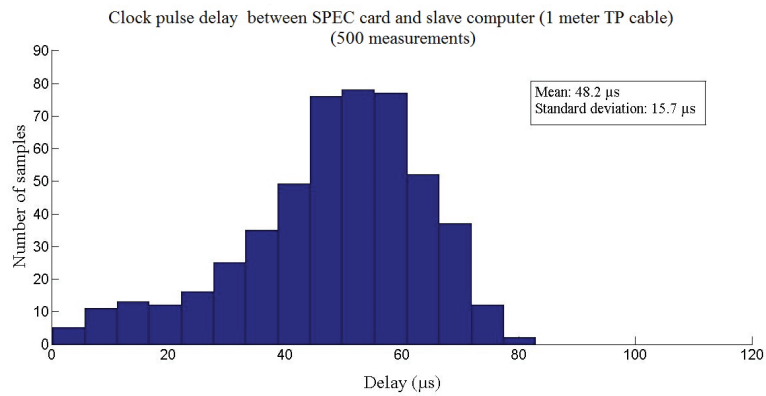


Figure 4.6: Clock pulse delay between SPEC card and slave computer (1 meter TP cable)

In the first scenario (result shown in Figure 4.5 and 4.6) two different cable lengths were used; the probes cable length was 2 meter and the coaxial cable connected to the SPEC card was 1 meter. This will of course introduce some delay between the clock signals but insignificant in this scenario. Assuming the propagation delay in a coaxial cable is 1.57ns/ft the pulse delay will be around 5 ns. [36]

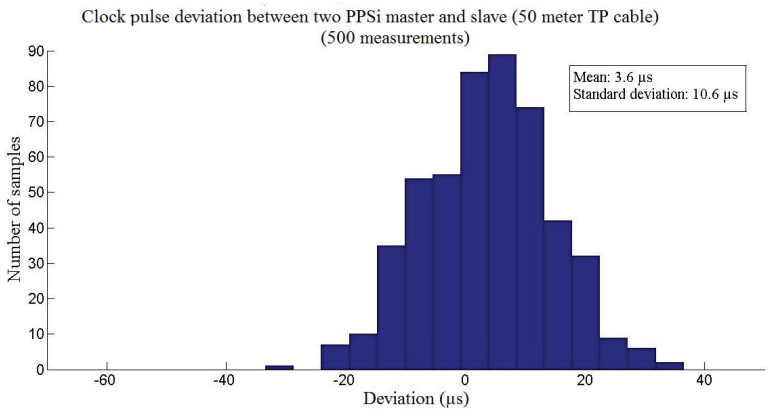


Figure 4.7: Clock pulse deviation between PPSi master and slave computer (50 meter TP cable)

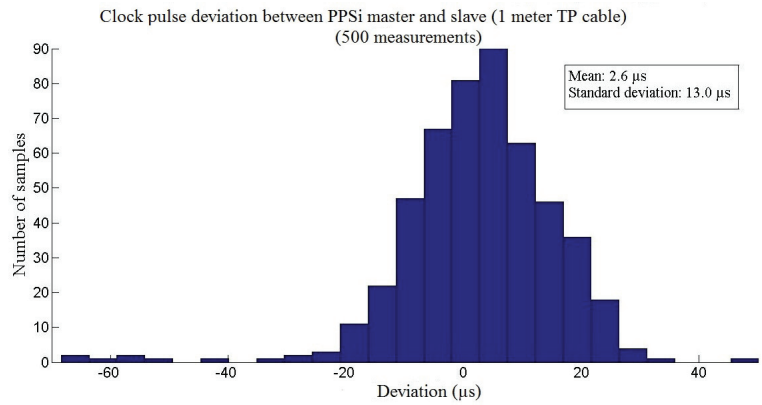


Figure 4.8: Clock pulse deviation between PPSi master and slave computer (1 meter TP cable)

As mentioned in Section 4.1, in order to test PPSi a software program called pps-out (included in the PPSi framework) was used to generate a pulse-per-second signal to a parallel port or a serial port. The parallel port was chosen in our scenarios since it has less delay and jitter than the serial port. However, running a software program to generate 1PPS will introduce couple microseconds of systematic error, due to software delays. The pps-out reports how late it was before and after generating the pulse edge, in our case it was between 0 and 2 microseconds.

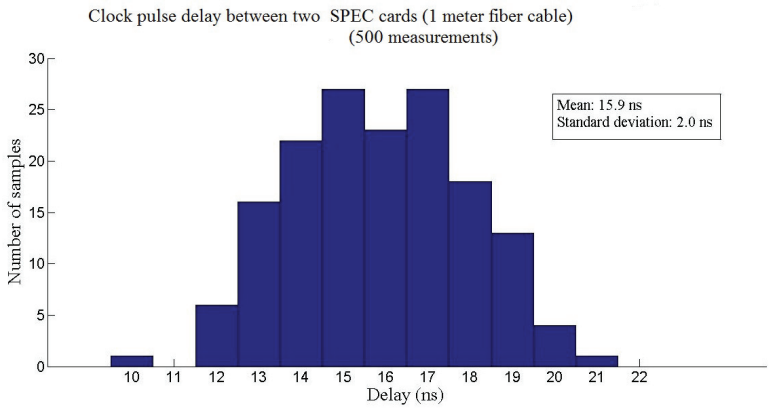


Figure 4.9: Clock pulse delay between two SPEC cards (1 meter fiber cable, notice the scale on the x-axis)

In Figure 4.9 the result obtained by comparing the clock signals between two SPEC cards is not as accurate and the mean is probably lower than Figure 4.9 shows. The reason is that it was difficult to get good samples from our oscilloscopes in such a time scale.

To clarify what is measured in our scenarios, in the first (and the third) scenario the clock offset or clock pulse delay between WR master and slave (running PPSi) is measured by comparing their clock signals. In the second scenario the clock deviation is measured, caused by the computer delay distribution in the slave and master computer. The reason for choosing to call the measurements either clock pulse delay or clock pulse deviation, is because the relative time difference in the first and third scenario is always positive (i.e. there is always a delay between the master clock pulse and the slave clock pulse), whereas in the second scenario the clock pulses deviates from one another (relative time difference can be either negative or positive). In first two lab setups the clock pulse delay or clock pulse deviation was measured in about 1.5 hour (until 500 measurements was reached). In the third lab setup only 158 samples/measurements was taken since there were a lot less variance on the relative time difference between the two clock signals in this case.

The distribution of the delays in the computer is caused by different entities or factors, making the clocks deviate from one another in the second scenario and most likely responsible for clock pulse delay in first scenario. The time it takes to run PPSi and the pps-out software can be seen as a distribution, since the time running the programs can vary (e.g. calculating the time offset, generating timestamps for received and transmitted frames or generating a PPS signal to the parallel port). The PPSi and pps-out program also reads and writes (pps-out program only reads the clock) to the system clock which also takes different amount of time, depending on the kernel. The OS and the kernel delay can also

be represented by a distribution, since interrupts forcing the PPSi and pps-out program to wait some time before running and the time it takes for accessing the hardware and the system clock can vary a lot. There will also be some variations of the delays in the NIC (Network Interface card), since the time for generating a frame to the Ethernet port can aviate and there can also be some queuing delays (in our case the queuing delay is more or less non-existing since there is very low network traffic). Additionally, it takes some time for the hardware to produce an electrical PPS signal to the parallel port. All distributions mentioned above cause the clocks on the master and the slave in the second scenario to deviate from one another.

Another big factor (most likely the biggest) is the internal computer clock, which suffers of clock drift making the clock run too fast or too slow (relatively) and is probably the reason why there can be a negative time difference in scenario 2, more on this in Chapter 5. In PPSi there is a PLL function with main objective to accommodate for clock drift, but it is relatively slow compared to how fast the computer clock drifts.

The delays in SPEC card are caused by the result of SFP transceiver and electronic components as well as the delays inside FPGA chip. Additional reception delay is also caused on both sides by aligning recovered clock signal to the inter-symbol boundaries of data stream as well as the delays caused from generating timestamps. In addition to hardware delays, when running PPSi, the SPEC is affected by the software delays, which is very small due to hardware timestamping of transmitted and received frames. Another important aspect of the SPEC card is that the clock in the card is a lot better compared to the computer system clock, since it less sensitive for e.g. temperature changes and therefore do not suffers as much from clock drift. Also, it more accurate compare to the computer system clock.

In the third scenario, when using SPEC to SPEC, each master and slave has more or less constant transmission and reception delays. As well as the delays that are resulting from the hardware delays mentioned above. The clock pulse delay between SPEC-to-SPEC is always positive since each WR master and WR slave suffers from very small software delays hence, the slave is always being late since the synchronization is never perfect.

Chapter 5
Discussion and conclusion

In this chapter the results of the three test scenarios are discussed. This chapter also presents future work and improvements to the PPSi timing solution. Finally a conclusion of the work and usage of PPSi is presented.

5.1 Discussion and expected result

This section compares the three test scenario and the result is discussed. It also presents the expected performance of PPSi in Cygwin.

5.1.1 Test result comparison

In the first scenario a WR SPEC card was configured as master and connected to the slave computer running PPSi through a media converter. This result is interesting because this is a possible scenario at ESS and is probably closest to reality of the three scenarios.

The result is believed to reflects how accurate the level of synchronization will be and how distribution of the OS and software latencies impact the level of synchronization. Considering how small the PPSi program is, a heavily loaded computer will most likely not affect the synchronization error (having multiple programs used for adjusting the clock while running PPSi will affect the synchronization accuracy, but it is not a good practice and therefore not taken into account). Having a heavily congested network card (and more network devices like switches between the media converter and the slave) will most likely to increase the synchronization error. The result will probably look as graph 4.5 and 4.6 but shifted to the right (higher error) and bigger variance due to asymmetric queuing delays.

The tail in graph 4.5 and 4.6 is believed to be caused by clock drift (causing the clock to tick too fast) which in turn cancels the other delays (OS, software latencies etc...), resulting in a lower offset between the master 's PPS signal and slave 's PPS signal. The tail is not visible in the other two graphs (second scenario) since the clock drift is assumed to be the same on both the master and the server, therefore the clock drift distribution will be centered on 0.

The overall result was expected compared to other PPSi test result and the results in similar scenarios are not expected to be better than this. Connecting a

PC via media converter to WR network, will as mentioned in section 4.1.1 disable WR on the link between the computer and the WR network.[25] The reason that the result between using a 50 meter cable or 1 meter cable in the first two scenario are so similar, is because the propagation delays in this cases are insignificant. The reason it is insignificant in these case, is because the propagation delays is more or less deterministic. In the second scenario, one of the computers ran PPSi as master, that was connected to another computer that ran PPSi as slave. The result is believed to show the computer delay distribution (times two, since the computer delay distribution exists on both master and slave), since the network delay in this scenario is more or less none existing.

If the graphs of the first two scenarios are compared, there is relatively big difference between running SPEC as master or PPSi as master. This because, the combined distribution (called the computer delay distribution) of the OS latencies (caused by interrupts and the OS kernel), software latencies (cased by PPSi and pps-out software) and clock drift, is more or less the same on both master and slave computers in the second scenario. When the PPS clock signals are compared and the clock pulse delay is measured by subtracting the relative time between the PPS signals, the delays will cancel each other out (not always, is why the distribution looks like it does). This is why the graph is centered on a couple of microseconds. The negative values occur when the PPS signal from the slave is generated early compared to the master's PPS signal. This is believed to be caused by clock drift making the clock ticking too fast, canceling the other delays and therefore the PPS signal to be generated sooner compared to the other PPS signal.

It is believed that the reason there is a relatively big difference between using a SPEC card as master instead of a PC, is because the cancelation of the delays mentioned before doesn't occur in this scenario (at least not in the same manner). The SPEC card doesn't suffers from OS latencies and very little from software latencies due to having hardware generated timestamps of received and transmitted package. The difference is believed to be caused by OS and software latencies and clock drift on the slave computer, making the PPS clock signal to be generated much (relatively) later compared to the signal from the SPEC card. The media converter also introduces some delays but those are most likely to be very small and insignificant. An interesting note is that the variances between the graphs in the first two scenarios are the same (a little bit higher in the first scenario, caused by the tail). This suggests that the error distribution is the same and is most likely to be the same in similar scenarios, since the computer delay distribution on the slave are the same.

The third scenario was more as a reference, used to compare to the result of the other two scenarios, the result was very much expected and confirmed with previous White Rabbit test. Comparing this result with the two other scenarios, it is clear that having hardware generated timestamps; no OS and small software related delays have big impact on the level of synchronization achievable.

5.1.2 Expected performance of PPSi in Cygwin

There are downsides with Windows concerning time synchronization, most of which already mentioned earlier in this document. The most sever is that Windows lack functionality for adjusting the clock frequency, making it very hard adjusting for clock drift. Another problem is the accuracy of the timestamps in Windows, since it is possible to get software timestamps in scale of hundreds of nanoseconds, compared to Linux which is in nanoseconds. Since PTP uses timestamps to calculate time offset, this will of course affect the achievable level of synchronization.

As mentioned in section 4.1.1 the WRPC can only send frames using raw Ethernet, making it impossible to directly connect a Windows computer running PPSi to a WR network (PPSi in Windows/Cygwin only can send frames via UDP). In order to solve this issue, we suggest using a Linux computer as a BC between the WR network and the Windows computer (this will affect the accuracy on the synchronization).

Using Cygwin as POSIX emulator also create another application layer which will introduce both asymmetric and symmetric software related delays, making PPSi in Cygwin run slower compared to Linux.

5.2 Future work

This section presents the future improvement of PPSi as well as future work on White Rabbit.

5.2.1 Future improvements of PPSi

Since PPSi is the result of a development effort providing full support for WRPTP extension, support a variety of devices and platforms which means that the path is leaving open for new PTP profiles and new versions of the PPSi standard. The current release of the PPSi standard works for both hosted (e.g. WR switch running Linux, not Windows) and freestanding environments, without a host operating system. The result of this master thesis is building a PPSi extension for Cygwin (Windows). But since Cygwin runs as compatibility layer around PPSi this will impact the performance, i.e. PPSi will be slower in Cygwin and adding another application layer may give more asymmetric delay. However, our proposal for a future work is to write a new standard for native Windows with much better synchronization accuracy and lower forwarding package delays is a new challenge for future developers.

It should be clear that PPSi is software program under development and there are several features to be added in the future (to mention one, IPv6 addresses). There is also the downside that WRPC (White Rabbit PTP core) used in the WR hardware (like SPEC cards or WR switches) only uses raw Ethernet frames as transport protocol, which is not supported in Windows. Adding support for UDP in the WRPC is considered an important feature for future improvement, so it is possible to directly connect a Windows computer to a WR network.

5.2.2 SPEC drivers for Windows

As of today, it does not exist any Windows drivers for SPEC cards, making it hard for Windows users to use White Rabbit. The effect of missing Windows SPEC card drivers we think deserves a future study/development.

5.2.3 Building native Windows PPSi

The design suggested here is centered on the WRPTP standard. The objective is designing a package that works on a native Windows producing better synchronization accuracy. If new requirements appear over time, they should require new solutions, but we cannot predict them at this stage.

5.2.4 Testing and evaluating PPSi in Cygwin

Unfortunately the PPSi Cygwin extension was not tested in this master thesis, because writing a program to generate a 1 PPS output to the parallel port is supposedly to be hard in Windows and no such program existed at the time of this thesis. Therefore the evaluation of the Cygwin extension is left as a future work.

5.2.5 Testing SPEC to computer for internal clock drift estimation

When sending or reception packets, each computer system records its own event using its internal clock, and in order to properly understand the computer system behavior, as reported by the events recorded on each computer, it is important to estimate precisely the clock differences and drift. The results obtained in the previous measurements show that the tail in graph 4.5 and 4.6 was expected caused from the clock drift. In order to ensure if that is correct, an additional testing is required, where two clock signals (1PPS) from the computer clock and the SPEC in an oscilloscope are compared to study the delay characteristics and synchronization time accuracy between them (the SPEC board is attached to the PC's PCIe express slot). In order to get a computer internal clock to be synchronized in time with SPEC, special software is needed for this purpose. Writing of this software and testing this lab-setup are leaved as a future work.

5.3 Conclusion

In this thesis, a purely software time synchronization protocol called PPSi PTP daemon is implemented. PPSi is an ideal timing protocol based on WRPTP. However, by using a combination of a freely software PPSi source code and WRPTP core binaries and any other tools available on the hardware repository website, was very helpful to achieve the target, as well as the purpose of the project. A newly defined PPSi extension as a free software package has been designed that supports Cygwin (Windows).

As a result of this work, using hardware achieves a better result compared to a software only approach. In the case where using hardware is hard and/or too costly a software approach is the only option.

As mentioned in section 1.2 Scope, one problem with two different demands was considered to be solved (A high demand representing time synchronization in scale of nano- or micro seconds and a low demand represents time accuracy in scale of milliseconds). As a result of this thesis, PPSi is the proposed solution to both of the demands considered in section 1.2. However, in Windows the high demand is most likely not to be met, and is left as future improvement of the PPSi software.

References

[1] *ESS Technical Design Report*, April 23, 2013, ESS-doc-274, ISBN 978-91-980173-2-8, http://europeanspallationsource.se/documentation/tdr.pdf

[2] ESS homepage, *E*SS Organisation, http://europeanspallationsource.se/ess-organisation

[3] ESS homepage, *ESS accelerator*, http://europeanspallationsource.se/accelerator

[4] J. Serrano, P. Alvarez, M. Cattin, E. Garcia Cota, J. Lewis, P. Moreira, T. Wlostowski, *THE WHITE RABBIT PROJECT*, https://espace.cern.ch/be-dep/CO/ICALEPCS%202009/1158%20%20The%20White%20Rabbit%20Project/TUC004_FINAL.pdf/

[5] G. Daniluk, T. Wlostowski, *White Rabbit*, http://www.ohwr.org/projects/wr-cores/wiki/Wrpc_core/

[6] M. H. Refan, H. Valizadeh, *Redundant GPS Time Synchronization Boards for Computer Networks*, Published in: Telecommunications Forum (TELFOR), publication year 2011, page:. 904 - 907 , INSPEC IEEE CONFERENCE PUBLICATIONS

[7] D. Dwyer, *How Atomic Clocks Work*, http://science.howstuffworks.com/atomic-clock.htm

[8] Microsoft Developer Network, *Windows time functions*, https://msdn.microsoft.com/en-us/library/windows/desktop/ms725473%28v=vs.85%29.aspx, downloaded 2015-03-23

[9] D. L. Mills., *Computer network time synchronization: the Network Time Protocol. Second*, Boca Raton, FL, USA: Taylor and Francis 2011. ISBN: 1-4398-1463-5

[10] Z. Wensi, *Study and Implementation of IEEE 1588 Precise Time Protocol*, Master's thesis, Faculty Council of the Faculty of Computing and Electrical Engineering on 10 September 2013

[11] University of delaware, *Huff-n'-Puff algorithm*, https://www.eecis.udel.edu/~mills/ntp/html/huffpuff.html downloaded 2015-03-13

[12] *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems.* In, IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002) (2008), pp. c1-269. DOI: 10.1109/IEEESTD.2008.4579760

[13] *Official NTP website, clock synchronization accuracy*, `http://www.ntp.org/ntpfaq/NTP-s-algo.htm#Q-ACCURATE-CLOCK` downloaded 2015-03-12

[14] SyncLab, *RADclock Accuracy and Robustness*, `http://www.synclab.org/radclock/performance.php` downloaded 2015-03-12

[15] NTPd, *NTP FAQ*, `http://www.synclab.org/radclock/performance.php` downloaded 2015-03-12

[16] *Cisco Connected Grid Switches System Management Software Configuration Guide, Configuring Precision Time Protocol*, chapter 5 `http://www.cisco.com/c/en/us/td/docs/switches/connectedgrid/cg-switch-sw-master/software/configuration/guide/sysmgmt/CGS_1000_Sysmgmt/sm_ptp.pdf`, downloaded 2015-03-10

[17] G. Daniluk, *White Rabbit PTP Core the sub-nanosecond time synchronization over Ethernet*, Master thesis, Warsaw University of Technology Faculty of Electronics and Information Technologies Institute of Electronic Systems (2012) `http://www.ohwr.org/documents/174`

[18] R. Lupas Scheiterer, D. Obradovic, C. Na, G. Steindl, F. Goetz, *Synchronization performance of the Precision Time Protocol: Effect of clock frequency drift on the line delay computation*, Publication Year: 2008 , Page(s): 243-246, IEEE CONFERENCE PUBLICATIONS

[19] M. BRUCKNER, R. WISCHNEWSKI, *A White Rabbit setup for sub-nsec synchronization, timestamping and time calibration in large scale astroparticle physics experiments*,33rd International Cosmic Ray Conference, Rio De Janeiro 2013, The Astroparticle Physics Conference, `http://www.ifh.de/~wischnew/talks/cta/icrc2013/icrc2013-1146_final.pdf`

[20] M. Lipinski, *White Rabbit: a next generation synchronization and control network for large distributed systems*, presentation by Rodney Cummings May 2012, Warsaw University of Technology `http://www.ieee802.org/1/files/public/docs2012/new-avb-cummings-white-rabbit-0512-v1.pdf`

[21] *Simple PCIe FMC carrier (SPEC)*, `http://www.ohwr.org/projects/spec/wiki` downloaded 2015-03-18

[22] G. Daniluk, T. Wlostowski, *HDR Core LIB, White Rabbit Core Collection: WRPTP core (WRPC)*, `http://www.ohwr.org/projects/wr-cores/wiki/Wrpc_core`, downloaded 2015-03-23

[23] Geoffrey M. Garner, Michel Ouellette, Michael Johas Teener., *Using an IEEE 802,1AS Network as a Distributed IEEE 1588 Boundary, Ordinary, or Transparent Clock*, IEEE CONFERENCE PUBLICATIONS

[24] T. Wlostowski, *Precise time and frequency transfer in a White Rabbit network*, Master thesis, Warsaw University of Technology, Faculty of Electronics and Information Technologies Institute of Radioelectronics (2011)

[25] A. Rubini, P. Fezzardi, M. Lipinski, A. Colosimo, *PPSi - A Free Software PTP Implementation*, Publication Year: 2014, Italy, IEEE CONFERENCE PUBLICATIONS

[26] A. Rubini, *PPSi Manual*, Mar 2013 `http://www.ohwr.org/attachments/download/1952/ppsi-manual-130311.pdf`

[27] *PPSi's Public Git Repository*, `git://ohwr.org/white-rabbit/ppsi.git`

[28] P. Christias, *Linux Programmer's Manual, adjtimex*, 2004, `http://unixhelp.ed.ac.uk/CGI/man-cgi?adjtimex+2`

[29] *GNU Lesser General Public License*, `http://www.gnu.org/licenses/lgpl.html` downloaded 2015-03-12

[30] Cygwin official website, *FAQ*, `https://cygwin.com/faq.html/`, downloaded 2015-03-06

[31] L. Donald, *POSIX Programmer's Guide*, O'Reilly, 1991. ISBN 0-937175-73-0

[32] *IEEE Guide for Developing User Organization Open System Environment (OSE) Profiles*, IEEE-SA Standards Board, Publication Year 1999, USA. IEEE CONFERENCE PUBLICATIONS pp. vi-63, ISBN 0-7381-1542-8

[33] Wikipedia, *POSIX*, 1 March 2015, `http://en.wikipedia.org/wiki/POSIX`

[34] NTPd, *Documentation of the NTP deamon*. Section: Operation Modes, `http://doc.ntp.org/4.1.0/ntpd.htm`, downloaded 2015-03-23

[35] A. Aulin Soderqvist, N. Claesson *A Timing System Application using White Rabbit*, Master's thesis, Department of Electrical and Information Technology, Faculty of Engineering, LTH, Lund University, January 2014.

[36] Calculating the propagation delay of coaxial cable, *FAQ*, `http://www.gpssource.com/files/Cable-Delay-FAQ.pdf`

Appendix A

Source code

The entire project source code is publicly accessible on the hardware repository website. The links to all the repositories that were required for implementing and testing PPSi are written below and can be downloaded by using git.

White Rabbit core collection
This is the first official release with PPSi as the WR PTP engine. Is a collection of cores in WR nodes and switches.
Git repository: `wr-cores:git@ohwr.org:hdl-core-lib/wr-cores.git`
Project website: `http://www.ohwr.org/projects/wr-cores`.

Software made for White Rabbit PTP core
The software that runs on LatticeMico32 as part of WRPC gateware which is responsible for controlling all HDL modules and carrying out WR time synchronization in WR Slave or WR Master.
Git repository: `wrpc-sw:git@ohwr.org:hdl-core-lib/wr-cores/wrpc-sw.git`.
Project webite: `http://www.ohwr.org/projects/wrpc-sw`.

Software support for the SPEC board
Software for SPEC boards, includes kernel, user space Linux code and FMC-bus driver. It is mainly used to flash the SPECs.
Git repository: `git://ohwr.org/fmc-projects/spec/spec-sw.git`.
Project website: `http://www.ohwr.org/projects/spec-sw`

Software PPSi
PPSi is the software implementation of PTP including the WRPTP extension used for synchronization in White Rabbit networks. The software is located here.
Git repository: `git://ohwr.org/white-rabbit/ppsi.git`.
Project website: `http://www.ohwr.org/projects/ppsi`.

Appendix B

PPSi för realtidssynkronisering (Popular science description in Swedish)

Vad händer om det försvinner eller läggs på några minuter eller sekunder i din datorklocka. De flesta datorer med interna klockor håller dem reglerade med hjälp av vibrerande kristaller. Men även dessa kristaller kan förlora sin noggrannhet p.g.a. yttre faktorer som fukt eller temperatur, vilket kan vara ett problem för vetenskapsmän som vill synkronisera klockan i deras dator till någon mätutrustning med högsta möjliga noggrannhet. I denna artikel beskriver vi PTP Ported to Silicon (PPSi) samt hur mjukvarustödet för PPSi kan användas för att nå en noggrannhet på mikrosekundsnivå.

I de flesta vardagliga tillämpningar spelar datorklockans noggrannhet inte så stor roll, men inom ett flertal tillämpningar kan den vara av central betydelse. Filhantering och arkivering, larm- och säkerhetssystem samt krypteringsnycklar är bara några exempel där det kan vara viktigt att klockorna hos enskilda datorer eller datorsystem är synkroniserade. Ett annat exempel som kräver en mycket hög noggrannhet är acceleratorer i t.ex. ESS och CERN. En enkel laborationsuppsättning är ett exempel på system som inte kräver lika stor noggrannhet.

För att uppnå tidssynkronisering med högre noggrannhet, så skapades PTP (Precision Time Protocol). PTP är framtaget av IEEE (Institute of Electrical and Electronics Engineers) med syfte att synkronisera datorer i ett LAN med hög noggrannhet och kan nå en noggrannhet på mikrosekunden.

PTP kan tyckas vara mer än nog till de flesta tillämpningar, dock är det inte tillräckligt för styrsystemen till acceleratorerna på CERN och ESS. Därför har utvecklare från bl.a. CERN utvecklat ett system kallat White Rabbit (WR). WR utnyttjar PTP och specialiserad hårdvara för att synkronisera över 1000 noder med en noggrannhet på nanosekundsnivå.

Problemet med att använda White Rabbit är att det kräver speciell hårdvara, som är dyrt och passar inte i alla datorer. I fall där extra hårdvara inte är möjligt, är en mjukvarulösning det bästa alternativet. Vår uppgift har varit att undersöka och föreslå en mjukvarulösning som kan användas för att tidssynkronisera datorer i vanliga- och/eller White Rabbit nätverk.

B.1 Principen för tidssynkronisering

Principen för tidssynkronisering är densamma för PTP och White Rabbit, som är baserad på att *master*- och *slave* datorerna tar tidsstämplar på meddelanden de skickar mellan varandra och att de delar dessa tidstämplar med varandra. Tidstämplarna t_1, t_2, t_3 och t_4 (figur 0.1) delas mellan *slave* och *master* för att beräkna tidsförskjutning T. Figur 0.1 visar hur två datorer mäter tidsförskjutningen mellan varandra. Vid tiden t_1 skickar *slave* datorn ett meddelande vilket anländer vid tiden t_2 hos *master*. För att kunna beräkna fördröjningen krävs ytterligare en överföring. Vid tiden t_3 skickar *master* ett svarsmeddelande till *slave* inklusive t_2 och t_3 vilket anländer vid tiden t_4. *Slave* har nu fått fyra tidsstämplar och kan nu beräkna tidsförskjutningen T.

$$T = 1/2[(t_2 - t_1) + (t_4 - t_3)] \tag{B.1}$$

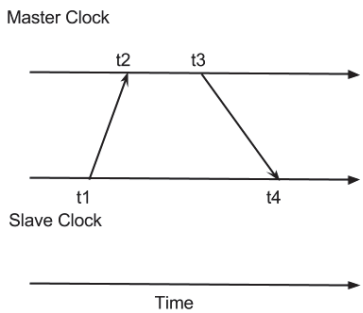


Figure B.1: Tidssynkronisering.

Tidssynkronisera datorer är betydligt mer komplicerat än vad bilden ovan visar, speciellt då det krävs noggrannhet större än ett par millisekunder. Dock ger det en generell bild på hur principen fungerar.

För att hitta en lösning på problemen vi nämnt innan har vi sökt efter nuvarande lösningar som kan fungera i de två scenarion vi satt upp. Vi stötte då på PTP Ported to Silicon (PPSi) som är ett portabel Precision Time Protocol (PTP), vilket betyder att PPSi är ett PTP program som ska fungera i många olika operativsystem (t.ex. Windows eller Linux), dock visade sig att PPSi inte fungerade för Windows. Vår uppgift blev därför att skriva en utökning av PPSi så att det går att köra på en Windows dator.

PPSi är utvecklat för White Rabbit projektet. PPSi synkroniserar nätverksenheter (servrar, switchar etc.) med hög precision genom att använda PTP. PPSi kan användas för att nå noggrannhet på under sub-mikrosekunder och upp till sub-nanosekunder med hjälp av hårdvara. PPSi kan fungera med (och utan) WR hårdvara, vilket gör det enklare att lägga till WR hårdvara i ett icke WR nätverk utan att ändra mjukvaran. PPSi fungerar för närvarande till både Linux/Windows

datorer (och fristående datorer utan operativsystem). Vi såg PPSi som en lösning på våra problem.

B.2 Hur noggrant blir det?

För att utvärdera hur bra tidssynkronisering kan fås i de två mål fallen (dvs. i ett LAN och i ett WR nätverk), testades PPSi i två olika labborationsuppsättningar som skulle motsvara de två fallen.

I den första uppsättningen kopplades två Linux datorer ihop med varandra, PPSi kördes i de två datorerna så att det skulle synkroniseras med varandra. För att mäta tidsförskjutning, användes att annat program som genererade en klockpuls (mätsignal varje gång klockan ökade sekunderna med ett) till ett oscilloskop. Klockpulserna från datorerna kunde jämföras för att mäta tidsförskjutningen. Resultatet visas i figur 0.3.

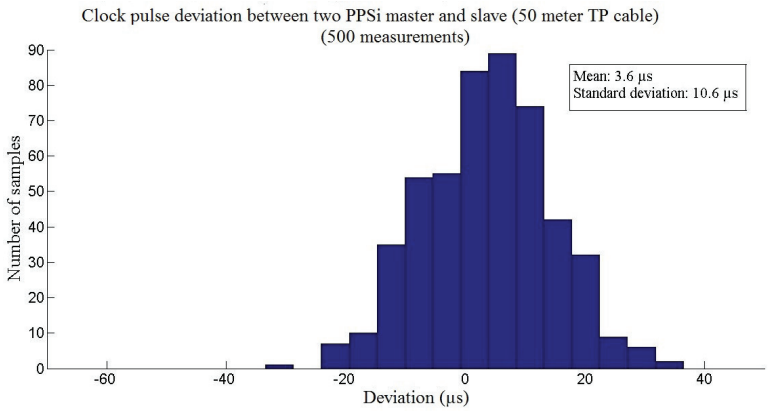


Figure B.2: Tidsförskjutning mellan två Linux datorer. Synkroniserade med hjälp av mjukvara (PPSi).

I den andra uppsättningen användes ett speciellt datorkort kallat WR SPEC kort som är ett speciellt framtaget kort för White Rabbit. SPEC kortet kopplades till en Linux dator (som körde PPSi). Som i förra uppsättningen så kopplades SPEC kortet och Linux datorn till ett oscilloskop så att tidsförskjutning kunde mätas. Resultatet visas i figur 0.3.

I grafen ovan (figur 0.2) kan man se att de flesta mätvärden ligger strax över 0 och att vi har en medelvärdet på ca 3.6 µs, vilket tyder på att tidsförskjutningen bruka vara runt 3.6 µs. Anledningen till att vi har mätvärden över och under medelvärdet beror på flera faktorer. Den mest betydande faktorn är datorklockan. Faktum är att datorklockan är långt ifrån perfekt och kan gå för fort eller för sakta (kallat klockdrift). Detta beror på att oscillatorn som driver klockan är känslig för ex. temperaturskillnader och fukt. Dessutom uppstår det fördröjningar i opera-

tivsystem och i mjukvaran (PPSi programmet). Att köra ett mjukvaruprogram i sig tar en vis tid i operativsystemet (OS), som kan vara upptaget vilket gör att PPSi programmet får vänta. Dessa faktorer tillsammans påverkar fördröjningen och ger upphov till tidsförskjutning mellan datorerna.

I den andra uppsättningen användes ett speciellt datorkort kallat WR SPEC kort som är ett speciellt framtaget kort för White Rabbit. SPEC kortet kopplades till en Linux dator (som körde PPSi). Som i förra uppsättningen så kopplades SPEC kortet och Linux datorn till ett oscilloskop så att tidsförskjutning kunde mätas. Resultatet visas i figur 0.3.

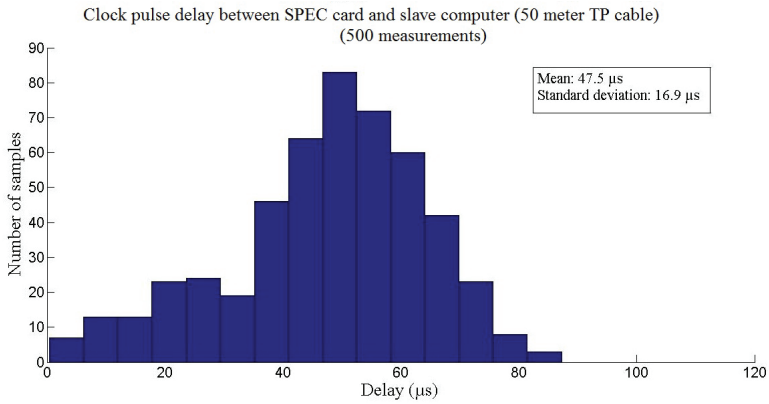


Figure B.3: Tidsförskjutning mellan ett WR SPEC kort och en Linux dator.

Den största skillnaden mellan de två figurerna ovan är dels att grafen är förskjuten det vill säga de flesta mätvärden ligger runt 50 µs, dels att där finns en bakdel eller svans i figur 0.3 bland de lägre fördröjningarna (mellan 0 och 30 µs). Denna bakdel (svans) tror vi beror på klockdrift, som gör att klockan går för fort, på så vis tar ut de andra fördröjningarna (OS och mjukvara fördröjningar) och vi får en lägre fördröjning.

Anledningen till att vi har en förskjutning mellan graferna är på grund av OS och mjukvara relaterade fördröjningar i slaven, som fördröjer klockpulsen genererat från slavens klocka. I ett SPEC kort finns det inget OS och mjukvarurelaterade fördröjningar är väldigt små, vilket gör att klockpulsen från SPEC kortet sällan blir fördröjd, medan fördröjningar från slaven orsakas av OS, mjukvaru fördröjningar och klockavdrift gör att klocksignalen från slaven genereras mycket (relativt) senare jämförelse med signalen från SPEC kortet. Detta är inte fallet i första uppsättningen eftersom OS fördröjningar, programvarufördröjningar och klockdrift, är mer eller mindre densamma på både master- och slavdatorer vilket gör att grafen i figur 0.2 är centrerat runt 0.

PPSi är för närvarande den enda mjukvarulösning som kan användas i både WR och icke WR nätverk och kan enligt våra resultat ge en noggrannhet på 10-

tals mikrosekunder (beroende på hur stor nätverkstrafiken är och hur stort LAN som ska synkroniseras). Vi tror att PPSi öppnar vägen för nya PTP implementationer och stöder ett stort område av apparater, system och plattformar i framtida acceleratorprojekt.

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2015-458

http://www.eit.lth.se