TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

ZHU WENSI

STUDY AND IMPLEMENTATION OF IEEE 1588 PRECISE TIME PROTOCOL

Master’s thesis

Examiner: Professor Markku Renfors
Professor Mikko Valkama
Supervisor: Manager Jouni Kemppainen
Examiner and topic approved by the Faculty Council of the Faculty of Computing and Electrical Engineering on 10 September 2013.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Degree Programme in Information Technology
WENSI ZHU: Study and Implementation of IEEE 1588 Precise Time Protocol
Master of Science Thesis, 70 pages
August 2013
Major subject: Communications Engineering
Examiners: Professor Markku Renfors, Professor Mikko Valkama
Supervisor: Manager Jouni Kemppainen
Keywords: IEEE 1588, PTP, Base Station synchronization, NTP

The synchronization among Base Stations (BSs) in mobile communication systems is such a vital technique that many services rely on it. IEEE 1588, also known as Precise Time Protocol (PTP), is defined to enable precise synchronization of clocks in measurement and control systems. As one of its targeted applications, the synchronization of base stations by PTP has become an important and popular researching topic.

This thesis investigates the feasibility of BS synchronization with the PTP method. It includes simulation based studies, implementation of PTP on practical BS cards, and tests with practical network devices in a laboratory environment. The results indicate that both frequency and time offsets are within the targeted range.

The thesis starts with a background discussion on BSs synchronization where reasons and requirements are introduced. Then the thesis explains the implementation related techniques. At last, the test results are presented and observations are analyzed.

PREFACE

This Master of Science Thesis, Study and Implementation on IEEE 1588 (PTP), has been carried out in the Department of Communication Engineering at Tampere University of Technology, Finland. The work has been done during the second half in year 2012 - 2013 at the System Department at CASSIDIAN, Finland. Supervisors are Jouni Kemppainen (manager) from CASSIDIAN, Finland, Markku Renfors (professor) and Mikko Valkama (professor) from Tampere University of Technology.

Tampere, August 2013.

Zhu Wensi
zhu.wensi@hotmail.com
Orivedenkatu 8 F 134,
33720, Tampere,
Finland
Tel. +358 469096102

ACKNOWLEDGEMENT

I would like to first grant my sincere gratitude to my supervisor – Jouni Kemppainen, an excellent manager from CASSIDIAN. When I was trapped in dilemmas, he was always there ready to help. He used his full technical experience and knowledge to help me locate the problems and gave me suggestions. His patience, experience, knowledge, willingness to listen and understandings towards subordinates make him a very good supervisor and manager. I would also like to show my gratitude to Professor Markku Renfors, for his numerous fruitful discussions during the development of this thesis and invaluable support. Further, thanks to Professor Mikko Valkama from whom I learned to solve problems independently, and no matter how many difficulties I met and how hard they are, being positive, independent and strong.

And I also must express my appreciation to everyone who helped me in CASSIDIAN. Without Dimitri's help, it would have taken me far more time to adapt the kernel and network driver; Antti, Lasse and Balazs helped me on codes design and optimization; Jozsef helped me with algorithm analysis while Jari instructed me on HW. Besides, there are lots of other employees in CASSIDIAN, whose names cannot be listed here one by one due to the space limitation, shared their idea with me, and took care of tools that I used, which made it a lot easier for me to focus on this topic area more effectively. Together with these excellent senior engineers and experts, not only did I learn more on professional skills, e.g., programming and communication system structures, but also gained the attitudes as an IT worker. (For example only start lab work after equipped with ESD protection and write logical codes carefully).

I also want to grant gratitude to OSCILLOQUARTZ company, who offered me a product grandmaster clock (5331) as synchronizing source, and a product level slave clock (5320), which I can test and compare my implementation. And during the observation, the OSCILLOQUARTZ 5320 slave is very stable and its PPS had little time offset from grandmaster PPS. Meanwhile, as an expert from OSCILLOQUARTZ, Gallet Laurent helped me with IEEE 1588 standard understandings which are precious experience gained from his practice.

The last but not the least, thanks to CASSIDIAN offered me such a great chance to contact the industry closely, and learn directly from the engineers and experts.

Thanks to all these people so that I had a wonderful, enjoyable experience when doing this master thesis in CASSIDIAN.

ABBREVIATIONS

ADEV	Allan deviation. One indicator to measure the clock quality.
BMC	Best master clock. A set of algorithms show up in IEEE 1588-2008.
BS	Base station. Offers radio interface for mobile phones.
BSC	BS controller. Controls and coordinates BSs.
CDMA	Code division multiple access.
DCXO	Digitally controlled crystal oscillator.
DDS	Direct Digital Synthesizer.
DHCP	Dynamic host configuration protocol. A host does not assign IP address to itself, instead a DHCP server does this for it.
DSP	Digital signal processing.
eTSEC	Enhanced three-Speed Ethernet controllers. Ethernet controllers in P2020.
FCB	Frame Control Block. Data structure used in P2020.
FDD	Frequency division duplex.
FDMA	Frequency division multiple access.
FIR	Finite impulse response. One kind of digital filter.
FLL	Frequency locked loop.
GPS	Global positioning system.
GSM	Global system for mobile communications. 2nd generation mobile system.
HW	Hardware.
IGMP	Internet group management protocol. Group management for multicast.
IIR	Infinite impulse response. The other kind of digital filter.
IP	Internet protocol. Third layer.
LAN	Local area network.
LTE	Long term evolution. 4th generation mobile system.
MAC	Media access control. Second layer in OSI 7 layered computer network structure.
MIMO	Multiple input and multiple output.
MS	Mobile station. A user equipment such as mobile phone.
NTP	Network time protocol. Synchronize computers through packet exchange.
OCXO	Oven controlled crystal oscillator.
OS	Operating system.
PDV	Path delay variation. Packets transmitted in the computer network suffer.

PLL	Phase locked loop.
ppb	parts per billion.
ppm	parts per million.
PPS	Pulses per second.
PSN	Packet switching network.
PTN	Packet transport network. An optical transport network.
PTP	Precise time protocol.
RF	Radio Frequency.
RTC	Real time clock.
SDH	Synchronous digital hierarchy. Same as SONET, but used elsewhere.
SONET	Synchronous optical networking. Standard for fiber optic transport. Used in North America
STM-N	Synchronous transport module level N. Fiber optic network transmission standard.
TCP	Transmission control protocol. Another transport layer protocol.
TCXO	Temperature compensation crystal oscillator.
TD-SCDMA	Time Division Synchronous code division multiple access.
TDD	Time division duplex.
TDM	Time division multiplexing.
TDMA	Time division multiple access.
TETRA	Terrestrial trunked radio. Professional radio system for security.
ToD	Time of day.
UDP	User datagram protocol. One of the transport layer protocols, above IP.
UTC	Coordinated Universal Time. Widely used time standard.
WCDMA	Wideband code division multiple access.

CONTENTS

1. INTRODUCTION

In mobile communication systems, Base Stations (BSs) are connected together indirectly. The synchronization among them is very important, despite what technique the system uses. They need to be synchronized in such a way that they offer the same carrier frequency and/or precisely aligned carrier phases. This BS synchronization is so critical that once the synchronization fails to meet its requirements, the Mobile Station (MS) connections are dropped or offered a bad quality of service. MS handover, radio framing accuracy and interference control on cell boundaries require the BSs to be highly synchronized. Figure 1.1 shows the three places where the synchronizations are possibly required depending on the employed technique.

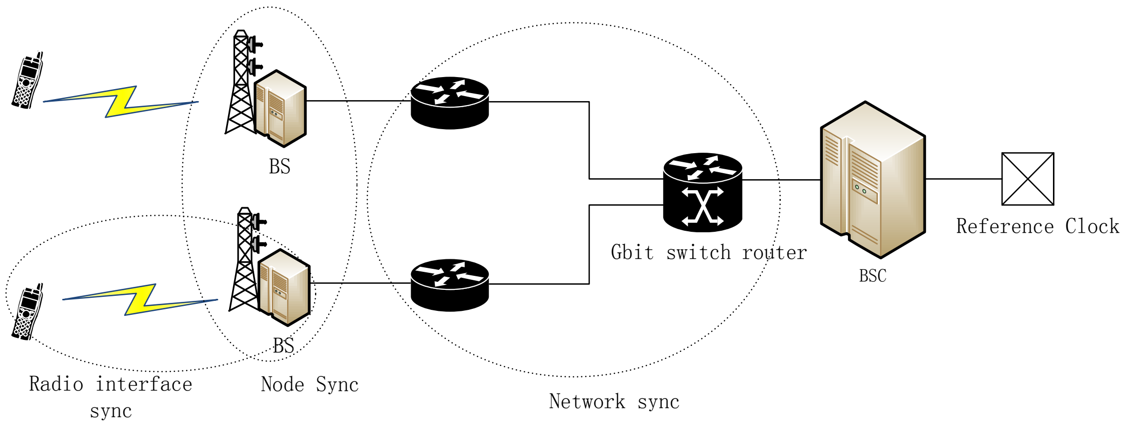


Figure 1.1. *Synchronization positions in a typical system.*

1.1. Background

In the legacy mobile communication system, the backhaul network was constructed using Time Division Multiplexing (TDM) or Synchronous Optical Networking (SONET)/Synchronous Digital Hierarchy (SDH). The TDM input frequency is traceable to a reference clock usually located near the Base Station Controller (BSC) with an error tolerance of 50 ppm. In the Global System for Mobile communications (GSM) system, BSs are synchronized in an end-to-end way, i.e., TDM works as a source for BSs (as a physical layer, SDH employs TDM technique). Later in 3G, depending on the employed access network technique and the willingness of operators, BSs can receive synchronization information through the Global Positioning System (GPS). As what has been commonly known, GPS usually offers time signal with an accuracy of 14 ns in theory [7]. But GPS is not always a reliable time source. Besides, it requires strict line-of-sight connections to the GPS satellites, which limits its application in Pico BSs (small-scale BSs placed in crowded areas).

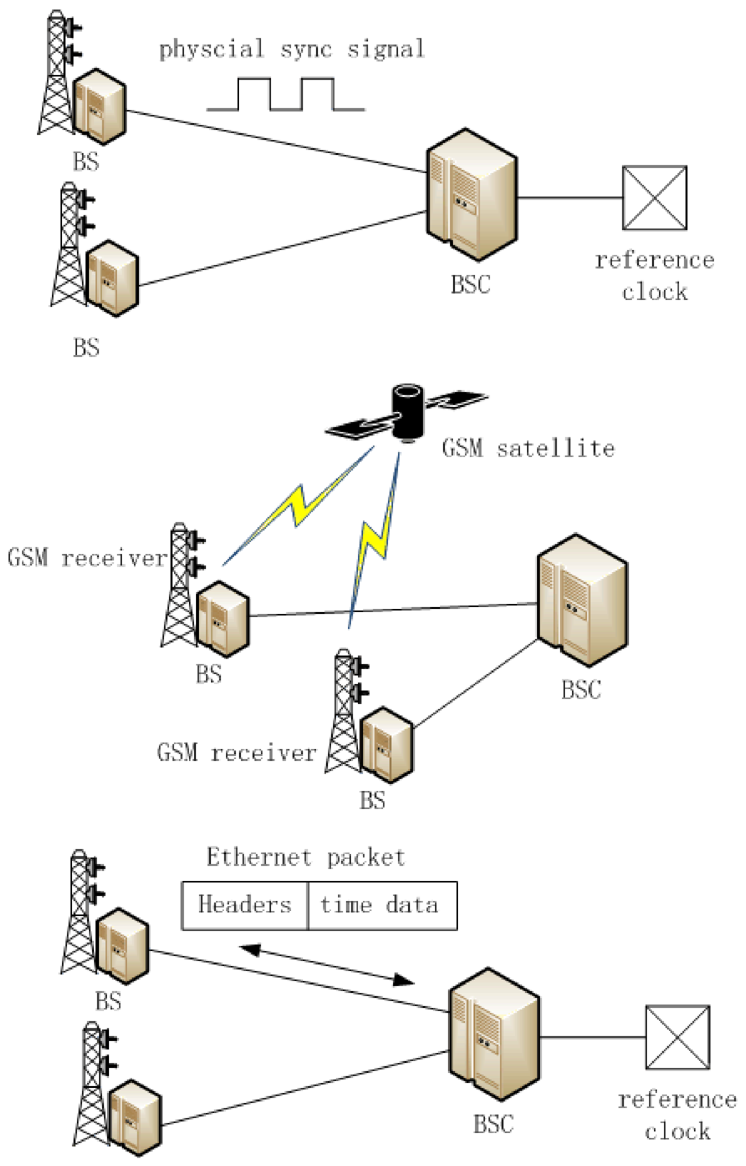


Figure 1.2. *Three methods for BS synchronization – physical layer transport synchronization signal, GPS receiver in each BS, and Ethernet time packets exchange.*

To satisfy the rapidly growing demands on data communication capacity and to eliminate the high expenses on SDH (both construction and maintenance), both operators and solution providers are driven to replace SDH with Packet Switching Network (PSN). Moreover, as PSN is based on the Ethernet technique, it inherits Ethernet's benefits, such as low cost, easy maintenance and fast recovery. As part of the SDH, synchronization signal is needed, as the data transmission relies on the precise frequency. In contrast, the network transmission in PSN does not demand accurate frequency. Figure 1.2 shows the three options for BSs synchronization – GPS, TDM (in SDN) type physical signal and the Ethernet packet-switched time service. There are standardized protocols aiming at the third method, such as Network Time Protocol (NTP) and Precise Time Protocol (PTP).

PTP, also known as IEEE 1588, has been developed for ten years since the first version of IEEE 1588 standard came out in 2002. The second edition was published in 2008. It was developed to offer microsecond to sub-microsecond accuracy that is suitable to apply in the mobile communication system. In fact, the time is actually an accumulation of frequency, therefore in many references (especially ones for NTP) time and phase show up equivalently. Similarly, time offset is often called as phase offset. In a short duration, the time modification reflects frequency change. Many solution providers tune Oven Controlled crystal oscillator (OCXO) according to the PLL input time reference. Based on these ideas, evidently, sharing time packets over Ethernet enables both time and frequency synchronization in BSs.

The PTP software (SW) in BSs builds a slave clock servo, which is in charge of controlling the local oscillator. The positive point for using PTP is that it does not need to change the already existing Ethernet structure. Instead, it only adds SW and hardware (HW), which assists SW to complete the timing service, in devices that need to be synchronized. Besides, with PTP there is no need to install as many antennas as in the GPS method, which makes PTP more cost-efficient. The negative point is that Ethernet is a noisy communication path, and the potential multi-link and unpredictable Path Delay Variation (PDV) lead to difficulties in designing reliable algorithms.

1.2. Requirements

On one hand, based on the way how different users share the communication resources, techniques in mobile communication systems can be categorized as Frequency Division Multiple Access (FDMA), Time Division Multiple Access (TDMA) and Code Division Multiple Access (CDMA). On the other hand, the method that is used to distinguish communication directions, i.e., uplink and downlink, can be divided into Time Division Duplex (TDD) and Frequency Division Duplex (FDD). In TDD, reception and transmission are operated on the same frequency but different timeslots. This requires the inter-site timing to be accurate, so that when MS is talking (traffic on uplink), all BSs are listening. In contrast, FDD uses two carrier frequencies for uplink and downlink respectively, so accurate frequency needs to be guaranteed. [6, p. 2]

Moreover, CDMA has a wide signal bandwidth, i.e., shorter symbol (chip) duration. Therefore CDMA exhibits time dispersive channels. Taking this benefit, Multiple Input & Multiple Output (MIMO) and rake receiver are often used in CDMA, which require precise inter-site timing phase.

The maximum frequency error and the time offset for some techniques are listed in Table 1.1. The frequency accuracy is the tolerance on the frequency error. It is defined as the difference between the actual BS transmission frequency and the assigned frequency [12]. The value (50 ppb) is to ensure that mobile terminals are still connected, even if they are traveling at a speed of 250 km/h (large Doppler shift) and need to handover to another site [3].

Table 1.1 Accuracy requirements in mobile communication systems.[34]

Layer	Sub Items	Freq. accuracy	Timing accuracy
Network *Sync*	E1	50 Parts per Million (ppm)	-
	Synchronous Transport Module level N (STM-N)	4.6 ppm	-
	Packet Transport Network (PTN)	Not so strict	-
Node *Sync*	BSC-BS	50 Parts per Billion (ppb) (if BS picks up time from other BSC)	3 µs (if BS picks up time from other BSC)
	Inter BS	50 ppb (if BS picks up time from other BS)	3 µs (if BS picks up time from other BS)
Radio interface	GSM	50 ppb	-
	Terrestrial Trunked Radio (TETRA) [11]	100 ppb	13.889 µs
	Wideband Code Division Multiple Access (WCDMA)	50 ppb	-
	Time Division Synchronous Code Division Multiple Access (TD-SCDMA)	50 ppb	3 µs
	Time Division Duplex - Long Term Evolution (TDD-LTE)	50 ppb	10 µs
	FDD-LTE	50 ppb	-

Although BSs designed by different solution providers have their own structures, it is common to have a Phase Locked Loop (PLL) controlled OCXO as radio frequency generator in a BS. To ensure that the accuracy of the output Radio Frequency (RF) frequency reaches 50 ppb, the BS frequency reference signal is experimentally required to reach 16 ppb accuracy. This means that if the reference frequency of PLL has 16 ppb accuracy, its control target – oscillator (OCXO) would have some head-room. This is shown in Figure 1.3.

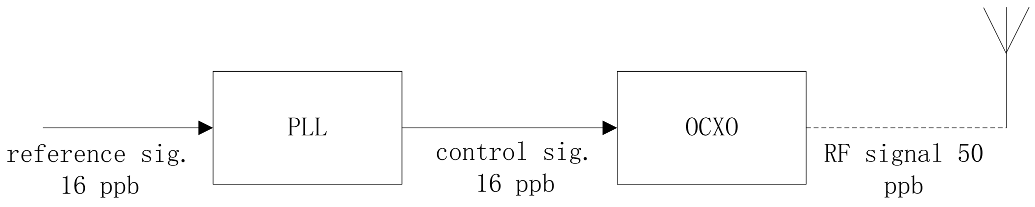


Figure 1.3. *Cascaded frequency accuracy.*

1.3. Crystal oscillators

In order to adjust the frequency in BSs, usually a controllable crystal oscillator is placed in a BS. The main control parameters are the temperature and voltage depending on the crystal oscillator features.

The principle of OCXO is to keep the temperature of OCXO at a predetermined value, at which the XO frequency-temperature curve presents zero slope. This value is beyond the temperature that the XO encounters without the heating oven. If XO is a SC-cut oscillator, the dynamic frequency-temperature stability is 100 times smaller than that of a static type (Table I and II in [32]). The idea of dynamic and static is expressed in [32], and shown in Figure 1.4. Moreover, in order to maintain the oven temperature, a servo circuit is used. The primary device in such a circuit is a thermistor. Therefore, it takes time to warm up the OCXO before it reaches the "zero-slope", which is the operating temperature.

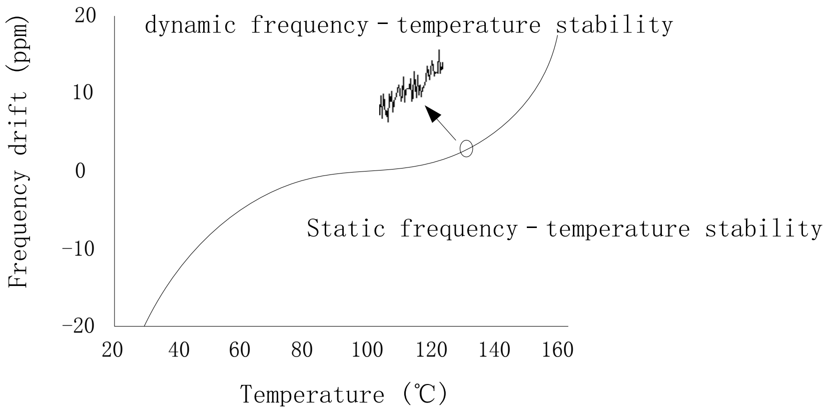


Figure 1.4. *Dynamic and static frequency-temperature stability of an OCXO.*

Besides OC, Digitally Controlled (DC), Voltage Controlled (VC), Temperature Compensation (TC) are widely used in different applications. It is commonly agreed that OCXO is better than DC or VC. The stability of XO is usually expressed by $\sigma(\tau)$, where τ is the observation duration in seconds. The short term stability could reach $\sigma(1) = 10^{-12}$ [1]. Randomly consulting a few OCXO and DCXO product datasheets on the Internet, the impression was that OCXO usually has sub-ppm stability while DCXO has it at about 20 ppm over one year. TCXO is a VCXO plus a temperature sensor [8]. Therefore, from the stability point of view, the best is OCXO, then come TCXO, DCXO and VCXO.

When an OCXO is used in a BS, the temperature of the oven is slightly adjusted in order to keep the frequency as expected.

1.4. Thesis outline

In this chapter, the background for the BSs synchronization and requirements were introduced.

The following chapters focus on the PTP related topics.

In Chapter 2, firstly the PTP theory is illustrated with implementation related notes in the IEEE 1588v2. Based on the time distribution mechanics, the synchronization difficulties and possible solutions are discussed. The last part in this chapter introduces PTP equipment used in this work.

In Chapter 3, HW assisted functions in FREESCALE P2020 processor, and the corresponding Linux kernel modifications are introduced with examples.

Chapter 4 introduces the implementations of the slave clock, including the network handler, data structure and digital control model.

Chapter 5 discusses the performance of digital control model introduced in Chapter 4 using simulated PLL method. The discussion is divided into two cases. The polluted case studies the effect of path delays and the performance of filters. By assuming the oscillator is stable and the network is free of delay variation, the behaviour of digital control model is studied in the ideal case.

In the last two chapters, PTP slave implementations will be listed and explained, followed by the results of experimental evaluations. At the end of Chapter 7, the analysis of the designed PTP slave clock would be concluded.

Appendix 1 is a group of Matlab simulation scripts for PTP slave implemented with the PLL method. Appendix 2 is another Matlab script studying about the behaviour of the PI controller parameter effects on the slave clock.

2. PTP THEORY

PTP is defined to enable precise synchronization of clocks in measurement and control systems. As one of its targeted usage, PTP used in dedicated networks, such as the backhaul of mobile networks, is becoming more and more popular. Before PTP was developed, NTP built a milestone on the computer synchronization in networks. It has been improved and widely employed for 30 years. Published in 2006, [30] serves as a very constructive guide about how to implement NTP. The latest NTP is Version 4 (NTPv4 for short) published in 2010, defined in RFC5905 [21]. The network environment that NTP was designed for was Wide Area Network (WAN), regardless of access technique. Compared with the dedicated network, it suffers from both large delay and delay variation, but has lower accuracy requirements.

2.1. Definitions

A clock in PTP is defined as a node that is capable of measuring the elapsed time from the beginning of a predetermined epoch. This measurement can be done either in HW or SW. Depending on the function and role of devices in a network, PTP categorizes them into four main types:

Ordinary clock: A clock that has a single PTP port in a domain and maintains the timescale used in the domain, e.g., master and slave clocks.

Boundary clock: If an ordinary clock has multiple PTP ports in a domain, it is called boundary clock). Typically, network devices embedded with PTP engines play such a role.

Transparent clock: A device on the communication path, e.g., router and switch, which can give timestamp at the moment that packets arrive and leave. So the delay caused by the jitter in this device can be subtracted eventually. This jitter information is kept in the *correctionField* in the PTP packet header.

Grandmaster clock: A clock that is the ultimate source of time within a domain, e.g., the master clock in this implementation is synchronised to GPS, therefore it is a grandmaster.

Figure 2.1 illustrates the positions of these PTP devices in a network.

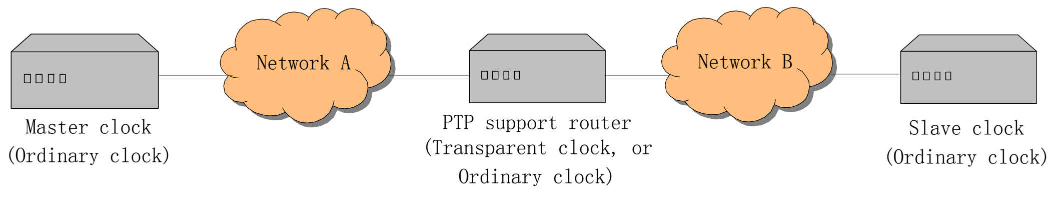


Figure 2.1.*Typical network connection where PTP function is employed.*

The synchronization is completed by exchanging PTP messages between a master clock and a slave clock. The messages can be divided into two categories. The event messages require time to be stamped in both master and slave, and the regular messages are packets that have configuration or timestamp in the data field. These message types are listed in Table 2.1.

Table 2.1 Messages in PTP.

	Type	**Function**
event	*Sync*	Sent by master, contains the egress time in master, and slave tags the ingress time on it.
	Delay_Req	Sent by slave, contains the egress time in slave, and master tags the ingress time on it.
normal	*Follow_Up*	Delivers timestamp in *Sync* but more precise when it is used in two-step mechanism.
	Delay_Resp	Sent by master, it has the ingress time of *Delay_Req*.
	Announce	The characteristics of a clock, e.g., how accurate it is.
	Signaling	Configuration, such as request and grant.
	Management	Configuration.

Follow_Up is only used in two-step mode. The PTP standard gives options on how to deliver timestamp *t1*, one-step or two-step. In one-step, *t1* is only carried by *Sync* message. In two-step mode, *Follow_Up* also carries *t1*, but with a better accuracy. This is because, depending on the specific master internal implementation, *t1* carried by *Sync* might be generated in the SW layer, resulting in a reduced accuracy. In this case, *Follow_Up* has to be used because HW timestamp tagged at Media Access Control (MAC) layer is not inserted into the *Sync* packet. For example, P2020 is such a chip that top layer fills in *Sync* packet with an estimated egress time, so it requires SW to take out timestamp from registers, and fill it in the corresponding *Follow_Up* packet. Comparatively, since one-step does not use *Follow_Up* packet, the timestamp in *Sync* would be the only source for a slave to catch *t1*.

Announce message is only sent by a clock port which is interested in serving as a master [20.p.122]. It contains grandmaster priority 1 and 2, clock quality, etc. For clocks that are only willing to be slave, such as the slave clock implemented in the range of this thesis, this message can be omitted. Instead of transmitting *Announce* message, these slave-only clocks keep an ear on incoming *Announce* messages to choose the master as stated in Best Master Clock (BMC) [20, p.109]. Figure 29 [20, p.116] in standard gives a description of data set comparison algorithm in the form of a flow chart. The *Announce* message contains the grandmaster data set, and it does not report the quality of the clock that sends this message (unless the clock itself is a grandmaster). However, this leads to two problems. The slave clock compares and chooses the master only based on its grandmaster, instead of the quality of the master. If several master clocks use the same grandmaster, which one should be considered as a better one and why the slave clock cannot directly talk to the grandmaster? In [17, p. 17], it is said that if there exists multiple hops from the grandmaster to the slave, the accuracy gained by using a boundary clock in the middle is better than that gained by slave directly treating grandmaster as its master. According to [16, p. 15], network devices, such as switches, could be divided into three types, boundary clocks, end-to-end transparent clocks and peer-to-peer transparent clocks. Taking boundary clocks as an example, and borrowing the conclusion from [17, p.17], reasons for the above questions could be explained in Figure 2.2, where the first situation is better than the second one. Some other constructive application suggestions are given in [17].

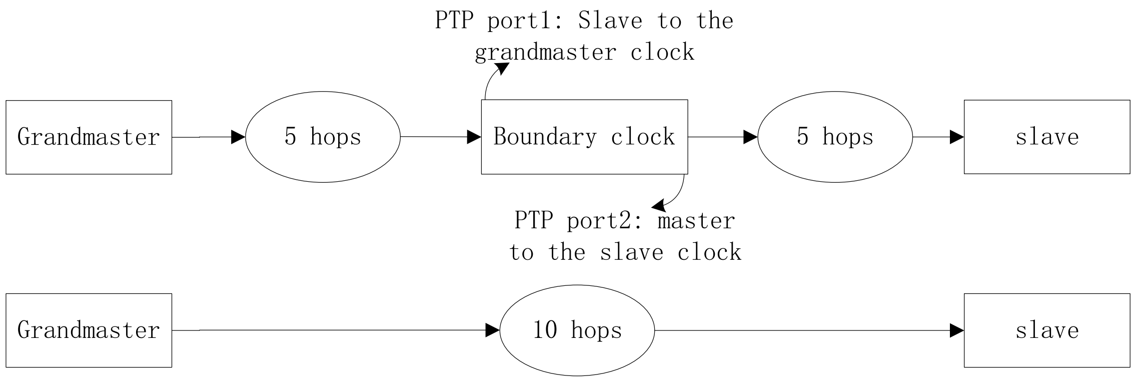

PTP port1: Slave to the grandmaster clock
Grandmaster
5 hops
Boundary clock
5 hops
slave
PTP port2: master to the slave clock
Grandmaster
10 hops
slave

Figure 2.2. *Boundary clock in the middle offers better accuracy than slaves talking directly to the grandmaster.*

Management message is described in [20, p.38] as a way to update the data set. However, In IEEE 1588v2 standard Chapter 13, many data fields are claimed as not part of the data set. Instead, items such as *physicalLayerProtocol* in CLOCK_DESCRIPTION [20, p.161], are self-autonomous configurations. When a dedicated management node is used, it may help to organize and administrate the PTP clock system, even to decide which clock is the most accurate according to the CLOCK_ACCURACY management message. OSCILLOQUARTZ can be configured through web easily by hand so this message is not used. Besides, it is rational to believe that the *Management* messages depend greatly on the communication terminals. Different clock vendors need to negotiate and agree on the message format. Otherwise no one understands the *Management* message from the other end. On the contrary, as NTP is used in large scale computer networks, it is claimed in [28, p. 5] that the autonomous clock hierarchy in NTP is very important. It is too risky to employ an autonomous administration in Telecommunication systems; otherwise a system engineer could not see what is going on in the clock network.

The purpose of a *Signaling* message is to negotiate optional services, such as PTP over unicast. In IEEE 1588v2 standard, both *Signaling* and *Management* message data are called TLV (type, length, value), but not the same. Terminology "TLV entity specification" and "management TLV" both exist. It is rational to believe that the "TLV entity specification" is the data of *Signaling* message while "management TLV" specifies data of *Management* message.

All PTP messages belong to a domain, which is assigned in the PTP common header. Domain is a logically independent region to form clocks clusters. The idea is very similar to computers (i.e., clocks) belonging to a subnet (i.e., independent region) through net masks and Internet Protocol (IP) addresses (i.e., subdomain number in PTP packet header). For every PTP packet destined for this clock, if the domain number is not what has been expected, this PTP messages will be discarded. And the difference between subnet and subdomain is that they are from different network layers, so they work independently. Figure 2.3 shows the domain field in the PTP packet header. This header is common for all PTP packets. In Table 2.2, T.S., M.T., R. and V. means transport specific, message type, reserved and PTP version, respectively.

Table 2.2 PTP common message header.

	0				15	16	31
0	T.S.	M.T.	R.		V.	messageLength	
4	domainNumber		reserved			flagField	
8	correctionField						
12	correctionField (continued)						
16	reserved						
20	sourcePortIdentity						
24	sourcePortIdentity (continued)						
28	sourcePortIdentity (continued)					sequenceID	
32	controlField			logInterval			

2.2. Synchronization mechanism

A slave clock (hereinafter, the slave) synchronizes itself to a master clock by exchanging the packets that carry time information. Based on timestamps, the slave calculates the time offset from master, and eventually adjusts its local time to be the same as that in the master. Indirectly, the frequency of the slave clock thereafter is also adjusted.

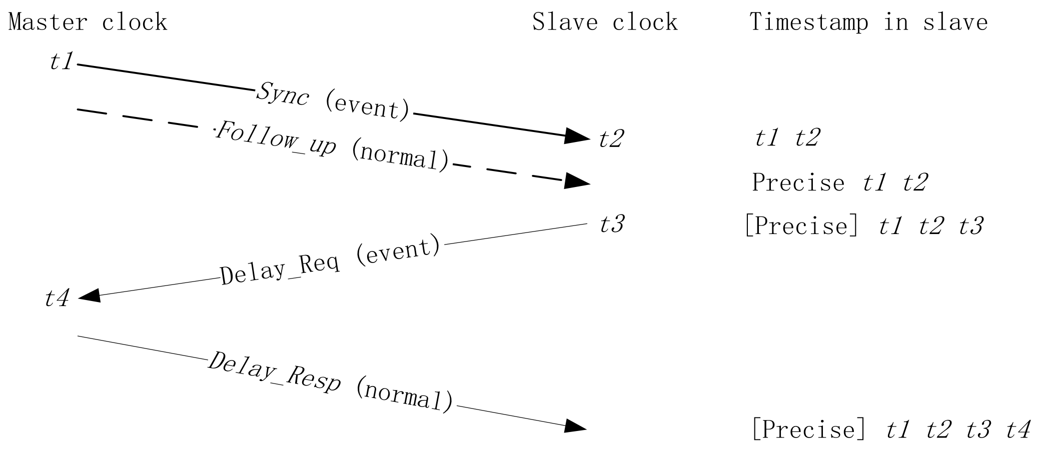


Figure 2.3. *PTP packets exchange.*

The synchronization mechanism is shown in Figure 2.3 where *t1* is the egress moment of *Sync* message in master, *t2* is *Sync* message ingress moment in slave, *t3* is *Delay_Req* message egress moment in slave, and *t4* is the time when the master receives it. All four timestamps are generated according to each clock's local time. Assuming that the time offset in slave is Δ from the master and the master-to-slave delay suffered by a *Sync* message is d_{MS}, we have

$$t_2 = t_1 + \Delta + d_{MS} \tag{2.1}$$

In order to estimate the delay, the slave sends *Delay_Req* messages. Assuming that the *Delay_Req* packet goes through a slave-to-master delay d_{SM}, we obtain

$$t_4 = t_3 - \Delta + d_{SM} \tag{2.2}$$

Combining above two equations, and assuming that delays in both directions are the same, *d*, the time offset from the master can be expressed as

$$\Delta = \mathrm{t}_2 - \mathrm{t}_1 - d, d = \frac{d_{SM} + d_{MS}}{2} \tag{2.3}$$

Based on Δ, the slave knows how much its local time differs from that in the master. Rewriting the above equation, (2.3) turns into

$$\begin{cases} d = \dfrac{\mathrm{t}_2 - \mathrm{t}_1 + \mathrm{t}_4 - \mathrm{t}_3}{2} \\ \Delta = \mathrm{t}_2 - \mathrm{t}_1 - d \end{cases} \tag{2.4}$$

When transparent clock is used, in the most simplified occasion (i.e., master, end-to-end, and slave one-step clocks; no asymmetry correction corresponding to Figure C.1 in [20]), (2.4) becomes

$$\begin{cases} d = \dfrac{\mathrm{t}_2 - \mathrm{t}_1 + \mathrm{t}_4 - \mathrm{t}_3 - c_1 - c_2}{2} \\ \Delta = \mathrm{t}_2 - \mathrm{t}_1 - d \end{cases} \tag{2.5}$$

c_1 – *correctionField* in *Sync* message,
c_2 – *correctionField* in *Delay_Resp* message.

2.3. Time offset in PTP

The time offset from the master clock is affected by two factors, namely delay asymmetry and clock drift.

2.3.1. Delay asymmetry

As part of the time offset, delay asymmetry is introduced by the PTP protocol itself. Packets transmitted through network generally suffer different delays in the two directions. There is no way to exactly measure one way delay, i.e., d_{SM} or d_{MS}. Based on Section 2.3, the master-to-slave delay is assumed as the mean value of the delays in the two directions, and this is unlikely to be the case in a real network. The calculated offset would have an error that is equal to the difference of the mean delay and the master-to-slave delay.

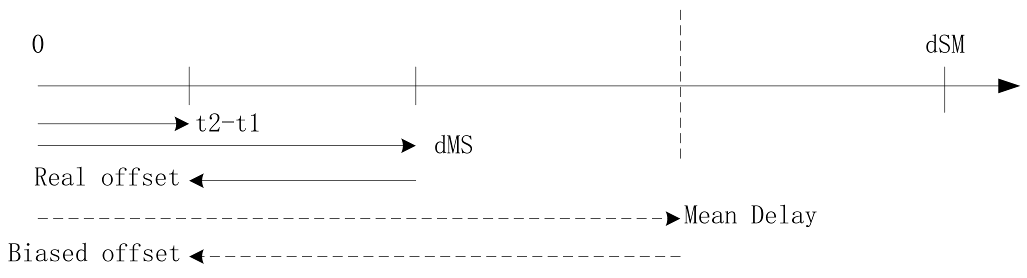


Figure 2.4. *Asymmetric delay affects the calculated offset (arrow shows the vector direction and length is the vector size).*

In Figure 2.4., the biased offset is gained based on (2.4), and its relation with real offset is

$$\text{biased offset} = \text{real offset - mean delay} + d_{ms} \tag{2.6}$$

Since the age of NTP, the effect of asymmetric path delay on offset calculation has been considered. One method claimed to be applicable is the so-called Huff-n'-Puff filter.

There are four main reasons that lead to asymmetric path delays. The first reason is the operating system latency. Between the generation of a PTP event packet and its transmission on the wire, the packet is manipulated and buffered by the protocol stack. Besides, when a packet is received, it is delayed by another interrupt mechanism. In order to avoid the OS latency, the event packets should be timestamped closer to the physical layer, e.g., at the MAC layer. By doing this, the OS latency can be decreased from microsecond or millisecond to nanosecond scale.

The second factor is asymmetric link speeds, and the solution is to use the same link speed for both directions. The third aspect lies in the network devices. No matter which technique is used in a switch, store-and-forward or cut-through, buffering and queuing packets raise delay. It is said that communication between multi-points and a single point suffers from asymmetric delays [5].

To deal with the asymmetric delay, four main methods have been seen so far. The figure in [13, p. 12] gives a very good outline for ITU-T suggestions on synchronization. It includes a portion of how to deal with PDV.

The first possible solution is to employ devices that support PTP and act as transparent clocks. Many router/switch manufacturers have already announced that their equipment is embedded with PTP functions (the newer router of Cisco, e.g., Cisco 7600 can be configured to support PTP). Figures C.1 to C.4 in [20] give very good instructions on how to employ transparent clock in both one step and two-step PTP. It is noteworthy that the time offset of transparent clock does not affect the delay and offset calculation in the slave clock. Compared with the other clocks, *correctionField* in transparent clocks contains not only the sub nanosecond residuals (for example, the egress time of *Sync* message in master is 144 second and 7.4 ns, 0.4 would be in *correctionField*) but also the residence time (during which the packet stays in network devices). This is the easiest way from the point of telecommunication solution providers – they don't need to consider the mechanisms to cancel asymmetry. Besides, the transparent clock method is, in theory, the most accurate way compared with the other three. The negative side is that these devices might be costly and not all customers are willing to update their devices already in use.

The second method is to use boundary clocks. It is claimed in [17] that, if the same number of intermediate devices is used, adding a boundary clock in the middle would be helpful in reducing error. This method is less accurate than the use of transparent clocks, although it still needs switches/routers to support the PTP function.

Compared with employing PTP supported switches/routers, packet priority is supported by most modern network devices. The principle of this method is that the higher priority packet is likely to suffer less switch delays. The adoption of priority shows up in [19]. There is a trend that the PTP header length would be adaptive for the priority usage in the future [18, p. 134].

There is a discussion about the importance of priority in [33]. Their conclusion is “under congestion, Transparent Clocks without prioritization struggle to maintain high-accuracy clock synchronization. Including cut-through Enterprise Ethernet switches with prioritization enabled results in better synchronization performance in congested networks. However, they do not maintain the same sub-microsecond accuracy TCs have when congestion is not present.” [33, p. 68]

The fourth method is called the PTP delay equalizer. It appoints a fixed delay to all PTP event packets. Network devices would buffer the packet into a disciplined queue to ensure that every PTP packet experiences the same known delay. This method is described as first disclosed under the name of "Controlled Delay"[5]. It is also proposed to add delay onto the PTP event packets and make the delays constant in [23].

There are two terms often rising indiscriminately in PTP papers – delay variation and delay. It is claimed that the high delay network usually also has a high delay variation [9]. However, during the procedure of doing this master thesis, what was observed is that the delay variation more likely leads to the frequency offset while absolute delay is closely related with the time offset. In order to characterize the network features better, many papers introduce the idea to use smaller interval between PTP *Sync* messages.

2.3.2. Clock drift

Another time offset is caused by the clock/oscillator itself. Many references depict the time difference between that in a clock which counts the periods of oscillator and the real time as

$$x(t) = x_0 + y_0 t + \frac{D}{2} t^2 + \frac{\emptyset(t)}{2\pi f_{nom}} \tag{2.7}$$

x_0 is the initial time error,
y_0 is the fractional frequency offset between object oscillator and the reference oscillator,
D stands for the linear fractional frequency offset of the measured oscillator,
$\emptyset(t)$ represents the random phase noise,
f_{nom} is the nominal frequency of the reference oscillator.

This is a continuous-time expression. Its sampled version can be easily rewritten by substituting in (2.7) *t* with *kT*, *k* = 0,1,2,...where *T* is the sampling period. The Fractional

Frequency Offset (FFO) between oscillator A (with frequency F_A) and oscillator B is defined as

$$FFO = \frac{F_A - F_B}{F_B} \tag{2.8}$$

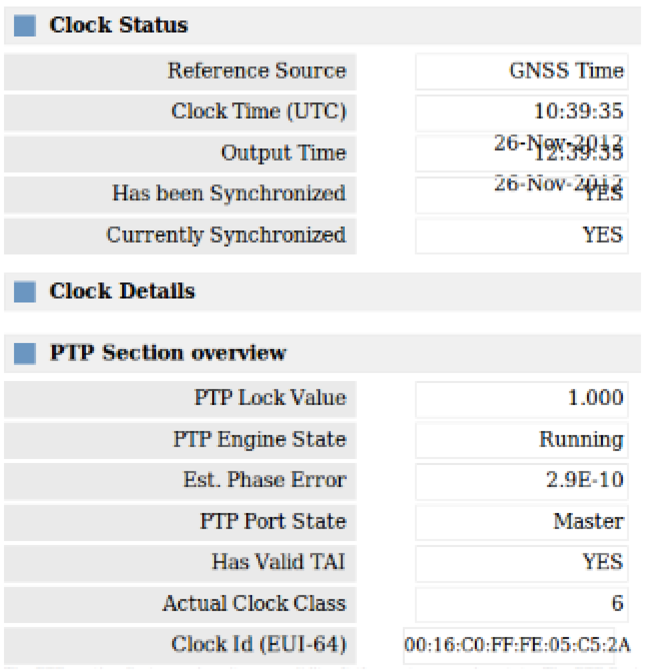
The values of y_0 and D can thereafter be derived based on (2.8).

The frequency drift of an oscillator (represented by D) is affected by the clock itself, such as aging, and environmental issues, e.g., pressure and voltage. In the implementation, the oscillating source is an on-board oscillator in BS and the temperature is the main factor. In [22] the random phase noise $\emptyset(t)$ is valued by 5 indicators, including Allan deviation (ADEV). Basically, $\emptyset(t)$ is a collection of all the temporary and unobvious frequency errors that has not been included in D. According to (2.7) and (2.8), in the thesis simulation (Appendix), random phase noise is considered.

2.4. PTP devices used in this thesis

This thesis got support from the OSCILLOQUARTZ Company. In order to evaluate the quality of the implemented slave, a test was conducted to compare OSCILLOQUARTZ 5320 slave and the designed slave. The slave was implemented in a computer whose processor is FREESCALE P2020. The comparison was based on the fact that the grandmaster should offer trustable reference time information, so a commercial product level grandmaster was used – OSCILLOQUARTZ 5331. In order to better understand the following chapters, the features of these two devices are explained in this section.

2.4.1. Grandmaster – OSCILLOQUARTZ 5331



Clock Status

Reference Source	GNSS Time
Clock Time (UTC)	10:39:35 26-Nov-2012
Output Time	12:39:35 26-Nov-2012
Has been Synchronized	YES
Currently Synchronized	YES

Clock Details

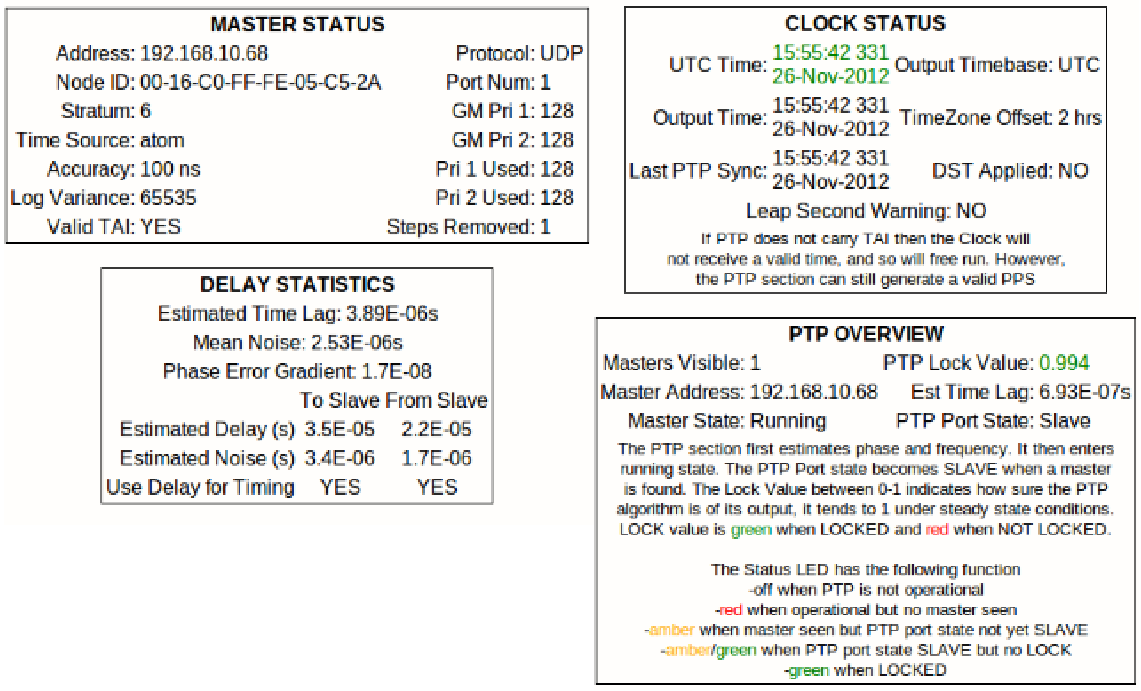
PTP Section overview

PTP Lock Value	1.000
PTP Engine State	Running
Est. Phase Error	2.9E-10
PTP Port State	Master
Has Valid TAI	YES
Actual Clock Class	6
Clock Id (EUI-64)	00:16:C0:FF:FE:05:C5:2A

Figure 2.5. *5331 management – Clock and PTP status screenshot.*

5331 is a grandmaster, with GPS receiver embedded. It can support up to four PTP engines, which means four isolated PTP domains (clock network subnet). It has two Ethernet interfaces, one is for PTP, and the other one is for management. Before starting to manage 5331, one needs to assign an IP address for the management interface on its front panel. The management of 5331 includes two aspects – configuration and status view. In configuration, engineers can log in through web (using the assigned management IP address) to set, for example, transmission method (multicast or unicast), PTP domain number, whether to use two-step mode, etc. In status view, one can see the clock information and PTP overview, as shown in Figure 2.5. The capacity of 5331 depends on the packet rate, it is designed to be, for each PTP engine, 50 PTP slaves at 64 pkt/s, 80 slaves at 32 pkt/s and 128 at 16 pkt/s. 5331 also supports many kinds of outputs, including the 10 MHz standard signal, and Coordinated Universal Time (UTC) second aligned PPS (hereinafter, the GPS PPS), these are used in the test of this thesis.

2.4.2. Product slave – OSCILLOQUARTZ 5320



MASTER STATUS

Address: 192.168.10.68	Protocol: UDP
Node ID: 00-16-C0-FF-FE-05-C5-2A	Port Num: 1
Stratum: 6	GM Pri 1: 128
Time Source: atom	GM Pri 2: 128
Accuracy: 100 ns	Pri 1 Used: 128
Log Variance: 65535	Pri 2 Used: 128
Valid TAI: YES	Steps Removed: 1

CLOCK STATUS

UTC Time:	15:55:42 331 26-Nov-2012	Output Timebase: UTC
Output Time:	15:55:42 331 26-Nov-2012	TimeZone Offset: 2 hrs
Last PTP Sync:	15:55:42 331 26-Nov-2012	DST Applied: NO

Leap Second Warning: NO

If PTP does not carry TAI then the Clock will not receive a valid time, and so will free run. However, the PTP section can still generate a valid PPS

DELAY STATISTICS

Estimated Time Lag: 3.89E-06s
Mean Noise: 2.53E-06s
Phase Error Gradient: 1.7E-08

	To Slave	From Slave
Estimated Delay (s)	3.5E-05	2.2E-05
Estimated Noise (s)	3.4E-06	1.7E-06
Use Delay for Timing	YES	YES

PTP OVERVIEW

Masters Visible: 1	PTP Lock Value: 0.994
Master Address: 192.168.10.68	Est Time Lag: 6.93E-07s
Master State: Running	PTP Port State: Slave

The PTP section first estimates phase and frequency. It then enters running state. The PTP Port state becomes SLAVE when a master is found. The Lock Value between 0-1 indicates how sure the PTP algorithm is of its output, it tends to 1 under steady state conditions. LOCK value is green when LOCKED and red when NOT LOCKED.

The Status LED has the following function
-off when PTP is not operational
-red when operational but no master seen
-amber when master seen but PTP port state not yet SLAVE
-amber/green when PTP port state SLAVE but no LOCK
-green when LOCKED

Figure 2.6. *5320 management – Clock and PTP status screenshot.*

OSCILLOQUARTZ 5320 is the slave. In terms of operation, it differs from 5331 mainly in the way how the management IP address is obtained. The address is detected by a SW which can be found in the product manual. If 5320 and the computer are in the same network, SW would detect the management interface IP address. One could log in 5320 though web interface by using that IP address, and change it later. Like 5331, the management of 5320 also includes two aspects – configuration and status view. In the configuration, the IP address of 5320 could be assigned, and the master clock list can be filled in manually. Besides, there are other configurable features regarding the method used to synchronize. In the status view dialog, not only the slave clock situation but also the status of the traced master is given, as shown in Figure 2.6. In "PTP OVERVIEW" session, if PTP lock value is 1, the slave is best locked. Once slave lost the connection with master, "MASTER STATUS" would be empty. One interesting point is in "DELAY STATISTICS", it displays estimates of the delay and noise. In the two opposite directions, different noise estimates are shown, which was claimed to be impossible in many reference materials. How this is achieved? In the slave manual, a clue was given as "The PLL and DDS designs are capable of smoothing out not only stationary noise (jitter, wander), but they also behave well under all sorts of abnormal transient conditions"[6].

2.4.3. Implemented slave processor – FREESCALE P2020

FREESCALE P2020 supports the HW assisted PTP. It has three sets of Ethernet controllers, called eTSECs. However, only eTSEC1 has the ability to assist the PTP SW, meaning that this interface must be enabled (e.g., in device tree) and write/read operation should be conducted in eTSEC1 registers. These registers cooperate to realize the PTP function, including control and generate timestamp, clock adjustments and PPS generation. By enabling the corresponding flags in a control register, this PPS could also be phase aligned, i.e., the rising edge can be adjusted. This procedure would be explained with examples in Section 3.2 HW support and configuration.

2.5. Summary

In this chapter, PTP was introduced from the practical point of view. Definitions from IEEE 1588v2 standard are illustrated with practical meanings and operations. Therefore, many other points of PTP, such as peer-to-peer mode and BMC, cannot be found here, since they were not part of the thesis implementation.

We first introduced the definition of several key devices, such as boundary clock, ordinary clock and transparent clock. PTP message types were illustrated one by one. Next we discussed the PTP synchronization mechanism, i.e., how slaves know the grandmaster time and calculate their time offset. Thirdly, the frequency wander, mainly due to environmental and aging factors, was described as a reason why oscillators need to be adjusted. Besides, the interferences that greatly affect the accuracy of PTP were explained together with possible solutions. At the end of this chapter, some devices used in this thesis, i.e., OSCILLOQUARTZ 5331, 5320 and FREESCALE P2020 were introduced.

3. HW ASSISTED PTP FUNCTION

In computer communications, after the application layer (a SW) generates the data, it is encapsulated and passed through the TCP (or UDP), IP, MAC and PHY layers. The message journey between different layers in OSs would meet a random and not a small amount of delay. The unpredictable delay variation is called jitter. In order to gain better accuracy, the jitter due to the system operation should be minimized. This requires that the timestamp should be generated as close as possible to the bottom layer of the network. This requires both HW and SW, which also includes OS to support it.

To enhance the readability, the processor register names and the SW components (e.g., functions and structures) are in italic font throughout this and the following chapters. Besides, if an object is followed by a bracket parenthesis ("[]"), a specific field of this object is referred.

3.1. Slave clock structure

A top-down approach to implement the HW assisted slave clock includes several blocks. The first block is a user space SW. It handles the network packet exchanging based on the service offered by the kernel layer; it maintains a data structure where the necessary information to calculate the time offset is stored; it also contains a servo, that uses a set of filters to optimally tune the local clock. The second block is a network driver. It separates the PTP event packets from all the others, and indicates the HW to generate timestamp on them; it maintains a circular buffer, which contains the event packets ingress timestamps that are ready for the user space SW to request. The bottom block is a set of HW registers, which form a clock, accompanied with a set of other services, such as the alignment of output PPS signal, and timestamp storages. Without the user space SW, the network driver and HW registers run freely. But with the user space SW, the time difference between the master and slave could be minimized by controlling the HW registers.

3.2. HW support and configuration

The HW support and its configuration are explained from the point of an ordinary clock, which is a slave using P2020 processor. Its structure is shown in Figure 3.1.

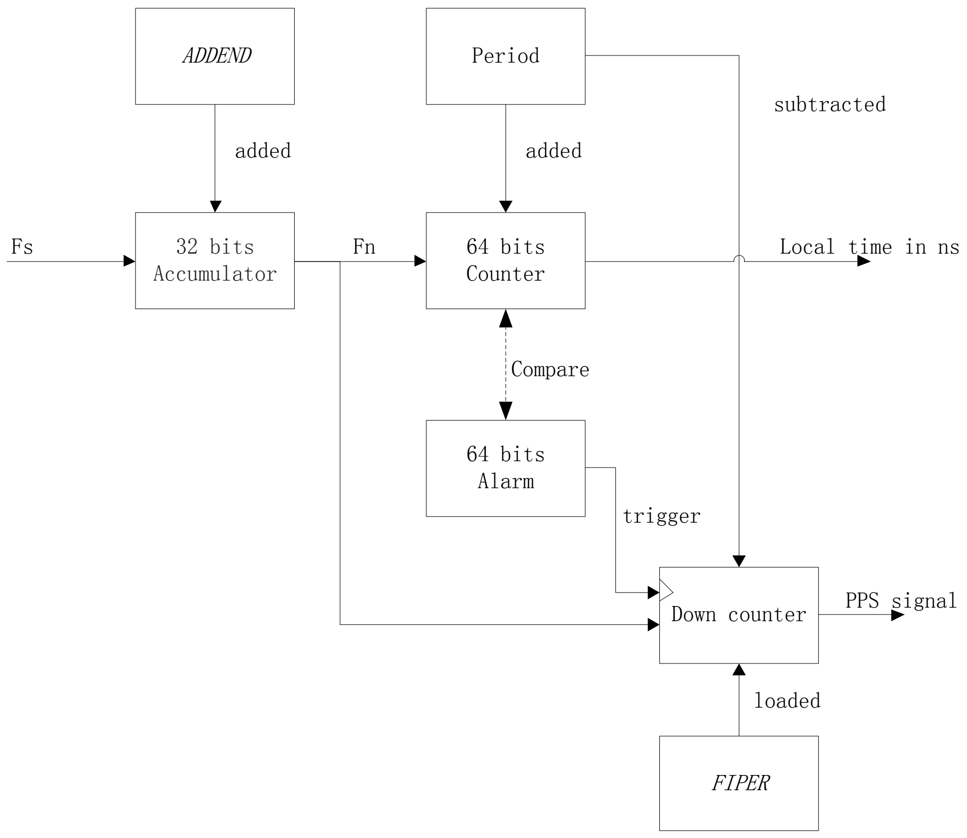


Figure 3.1. *P2020 HW clock structure.*

F_s is the frequency of eTSEC system clock (i.e., system frequency). Other than this, eTSEC1 transmit clock, external high precision timer reference clock (*TSEC_1588_CLK_IN*, a signal pin on P2020) and Real Time Clock (RTC) function can also be used, configured by *TMR_CTRL[CKSEL]*. This thesis was implemented while only the eTSEC system clock and eTSEC1 transmit clock are available. The system clock was used, because its frequency is higher (eTSEC1 transmit clock frequency is 25 MHz). And a higher F_s offers better clock resolution. F_n is the nominal frequency, both in Hz, and their relation is:

$$F_n = F_s \cdot \frac{A}{2^{32}} \tag{3.1}$$

A – the value in *TMR_ADD* register, hereinafter, the addend.

The system clock is an oscillator that offers the frequency used by the processor. The same clock can also be used for other chips on the same board. In this implementation, its ideal frequency is 200 MHz. For every pulse from system clock, *A* is added into the timer accumulator register (*TMR_ACC*). Once the accumulator overflows, it gives a pulse as output. For example, if *A* is 2^{31}, meaning for every two pulses received by the down counter, one pulse would be generated, so the nominal frequency would be half of the system frequency. The *TMR_CTRL[TCLK_PERIOD]* holds the period value, which is added into the timer counter register (*TMR_CNT_H/L*) at every incoming pulse. If the nominal frequency is 100 MHz, the period value would be 10, to make *TMR_CNT_H/L* have 1000 MHz (nominal frequency times the period value) frequency, having the same meaning of nanoseconds (recall that one second is 1000 million nanoseconds). Because of this, the period value is also the resolution – the minimum accuracy a clock can offer. The counter register (*TMR_CNT_H/L*) therefore provides the time information, which is compared with that in the master clock. The goal is to make the value in this counter as accurate as possible to the master clock. Counter register is a 64 bit register (a pair of 32 bit ones).

Unlike these accumulators, the way to generate PPS signal in this processor is achieved by down counting. Every pulse would cause one period being subtracted. When the value in down counter register is no bigger than one period, at the next pulse, it would generate a PPS signal. And one pulse after that, the value in the *TMR_FIPER1* register is loaded again, so the *TMR_FIPER1* register holds the initial minuend. A tidy equation to sum these descriptions and what value FIPER should have is

$$F = F_n \cdot P - P \tag{3.2}$$

F_n – nominal frequency,
F – value in *TMR_FIPER1* register,
P – value in *TMR_CTRL[TCLK_PERIOD]*.
For example, assuming the system frequency is 200 MHz, if one wants to use 100 MHz as the nominal frequency (can be any value less than 200 MHz), *TMR_FIPER1* and *TMR_CTRL[TCLK_PERIOD]* should be 999999990 and 10 respectively.

The width of this PPS signal is twice the period value. This is because when value in the down counter is no bigger than the period, PPS signal raises to high level. If the period is in nanoseconds, the duration of PPS high level is $2 \cdot P$ ns. One needs to ensure that the width is not too small for the following peripheral to catch it.

In order to get a phase aligned PPS signal, *TMR_ALARM1* register is used. Once the value in *TMR_CNT_H/L* reaches that in *TMR_ALARM1*, a down counter for PPS starts to work. If *TMR_CNT_H/L* offers accurate enough time information, the PPS signal can be phase aligned to the reference PPS with nanosecond accuracy.

Timer offset register (*TMROFF_H/L*) is a register that plays an important role after the counter (*TMR_CNT_H/L*) overflows. If the 64 bit *TMR_CNT_H/L* register is not enough to present current time, offset register is used to simplify the timestamp operation in SW. Offset can also be used to form a time scale which has its own special epoch – let offset register contain value that is the difference between *origTime* in PTP message, and local clock time. When the SW compares the local clock time and the *origTime* in PTP message, offset should be added to the local time. The benefit of doing this is multiple SW processes that need ToD (Time of day) can visit the register, which is easier than using Inter-Process Communication (IPC) to get this value.

Without the support of clock driver or kernel, directly taking timestamp in *TMR_TXTS* (transmitted timestamp) or *TMR_RXTS* (received timestamp) registers is easier and efficient. *TMR_TXTS-ID* registers hold the value that can be used to distinguish timestamped packets, while *TMR_TXTS* contains the timestamp for that packet, so are the *TMR_RXTS-ID* and *TMR_RXTS*. The shortcoming by doing this is the PTP SW must have a mechanism to handle the possible HW latency. No matter in which way the timestamp of the packets is gained (by polling or interrupt), the HW doesn't touch anything in the PTP message, i.e., no HW "pen" would fill in the timestamp in packets for PTP SW – it is PTP SW's task to fill it in.

To enable the above registers, registers that control the PTP HW function are also necessary to be correctly configured. For example, if the following parameters are used: 10 ns period, external storage for transmitted packet timestamps and using system clock, the configuration of *TMR_CTRL* register would be 0xA8005. And the *RCTRL[TS]* in receive control register (*RCTRL*) needs to be set. This processor offers other configurable features as well, but since they are not so helpful for the purpose in this thesis implementation, they are not listed here. One can find more information about P2020 chip in [14] and especially its PTP specific issues in [15].

3.3. SW support for timestamp information

Since the HW support is achieved at the PHY layer or MAC layer (the processor used in this thesis is at MAC layer), the kernel or network driver should offer solutions for:

1. How to inform the HW to timestamp the PTP message when it passes the MAC layer.
2. How to deliver the timestamp to the PTP SW.

The first issue also requires to distinguish what packets need to be timestamped, i.e., to recognize the PTP messages.

The Linux kernel used in this implementation has Version 2.6.34.x. And the driver is called *Gianfar*, made by P2020. Its code can be found in that version kernel tar ball, at http://www.kernel.org/. *Gianfar* driver in newer kernel have functions to deal with the above issues, but not the one in 2.6.34, so the driver needs to be modified.

Conventionally, data at PTP layer is named as message (as called in IEEE 1588v2 standard), UDP is message, IP is packet and MAC is frame.

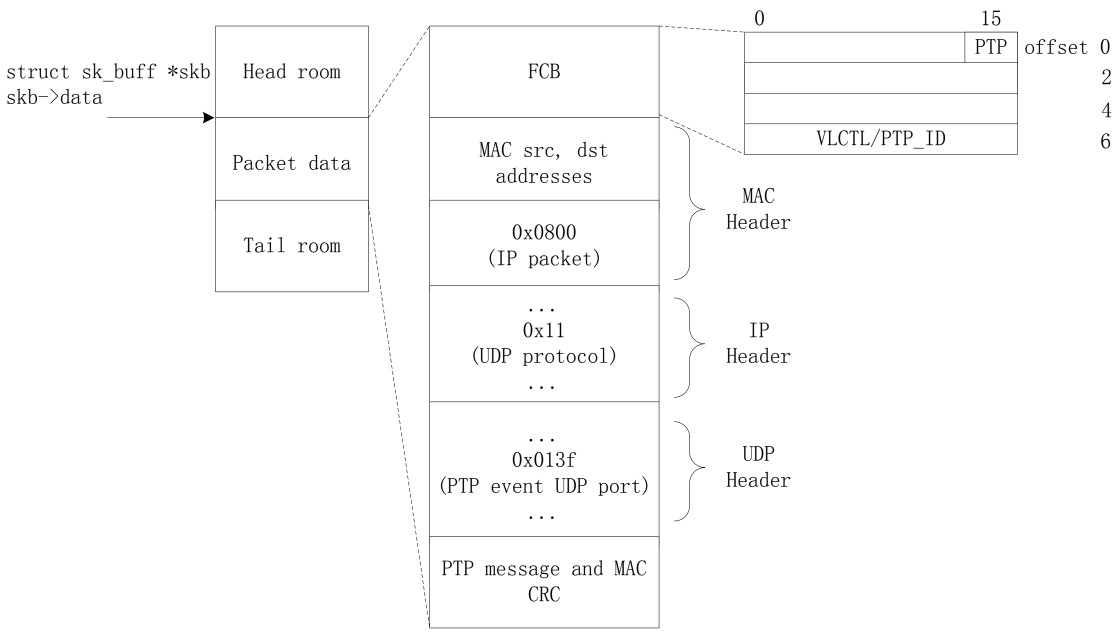


Figure 3.2. *The PTP sensitive items in the data field of sk_buff structure.*

In order to tag and store timestamps in transmitted event packets (and also in the *Follow_up* message), *sk_buff* plays an important role. *sk_buff* is such a structure that links the kernel protocol stack (TCP, UDP and IP, etc) and the network driver no matter in which direction data flows to (transmitting or receiving). It has a field called *data* that is actually the MAC layer frame. PTP protocol uses reserved UDP port, 319, for its event messages. The MAC frame at this moment already excludes the Start of Frame (SoF, a special bit sequence defined in MAC). Checking the UDP port value, if not 319, this is not a PTP message. The same check is done to the other sensitive bytes too. For example, in our implementation, the slave SW uses, from top down, PTP, UDP, IP and Ethernet, so checking the sensitive items is to see if it is UDP, and IP. *sk_buff* contains Frame Control Block (FCB) and MAC frame. This structure and PTP sensitive items that need to be checked are shown in Figure 3.2.

After a packet passes all these inspections, it is considered as a PTP event packet. The network driver will then inform HW to give a timestamp on it. In P2020, this is done by adding a FCB structure at the beginning of normal frame. And this “add” operation is done by writing *sk_buff*. FCB contains several items, when it comes to the PTP packet, FCB[PTP] and FCB[VLCTL] are special fields compared with the other packets. Setting FCB[PTP] to 1 is to turn on HW supported timestamp function for this packet (a more correct way to say is “this *sk_buff”*). The *TMR_TXTS* would contain the value in *TMR_CNT_H/L* exactly at this moment, and *TMR_TXTS_ID* would be filled in with FCB[VLCTL]. FCB[VLCTL] is shared between Virtual Local Area Network (VLAN) control word and PTP. FCB[VLCTL] can be assigned as a combination of subdomain number, message type (for transmission, either *Sync* or *Delay_Req*) and sequence number to uniquely identify the packet. Because there are 2 pairs of *TMR_TXTS*, (recall that *TMR_TXTSn* records timestamp for *TMR_TXTS_IDn*), and they are in turn filled in. We can keep polling the *TMR_TXTS_ID1/2* register until one matches, and its corresponding *TMR_TXTS_H/L1/2* register would have the expected timestamp. However, we cannot be fully confident about the reliability of HW, so a counter in the SW is added to avoid an endless loop when none of the *TMR_TXTS_ID* matches.

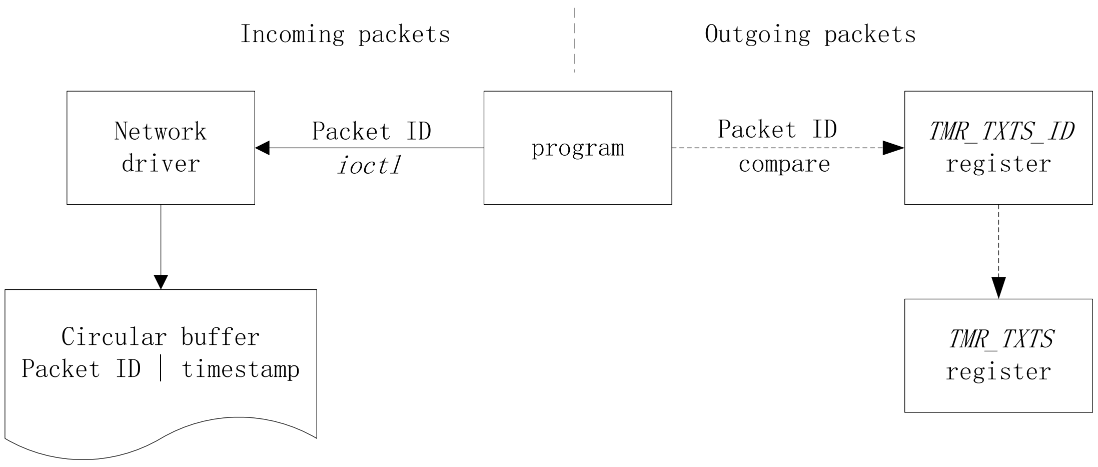


Figure 3.3. *Method to get local timestamp for outgoing and incoming packets.*

The PTP frame recognition in reception also relies on the *sk_buff*. However, unlike the timestamp operation for transmitted packets (recall that the transmitted packets are only given timestamp feature after the packet's *sk_buff* FCB[PTP] is set to 1), the processor gives timestamp to all the incoming packets. Although the processor also has a *TMR_RXTS* register to store the timestamp generated by HW, it is too harsh for a SW to catch it up. Even with interrupt handler, the flood on incoming packets would make this task tough.

Instead of polling this register, in order to pick out the PTP packets among all the new comers, the driver parses the sensitive fields that uniquely point to PTP packets. Once a PTP packet passes this check, the driver stores the *TMR_RXTS* register value into a circular buffer along with an ID. The same as that in the transmission, ID is a combined word with subdomain number, packets type (either *Sync* or *Delay_Req*) and sequence number. If the *RCTRL[TS]* is set to 1, when the user SW needs to know this information, it sends a request to the OS by using the system call function *ioctl*. The driver chooses a unique constant that is required for this function from user's own defined constant, e.g., *SIOCDEVPRIVAT*. The driver would then take the responsibility to look up the list, and return the timestamp. The operation and maintenance on such a circular buffer following normal steps – check if the buffer is empty, insert new item and look up. The size of this circular buffer should be large enough so that even the PTP engine runs at the maximum throughput, the fresh data would not be flushed. And in order to make this lookup faster, *Sync* and *Delay_Req* could be stored separately (i.e., two circular buffers). Figure 3.4 shows the structure of stored ID, FCB[VLCTL] is the ID record only for the transmitted packets, while the circular buffer is for received packets.

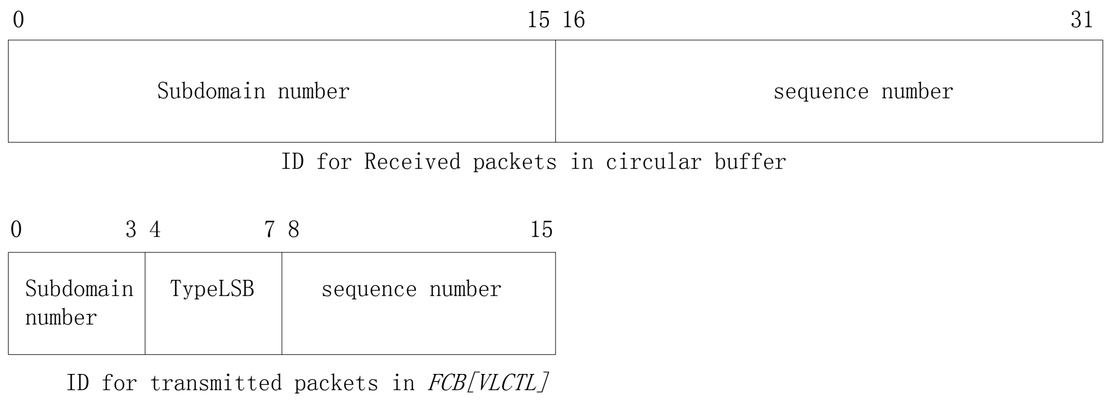


Figure 3.4. *ID structure for transmitted and received packets.*

TMR_CTRL and *RCTRL registers* are special in that they need to be initialized after all the others. And writing to these two registers would reset the other related registers to the default values.

Although the kernel used in this implementation does not support the PTP, versions newer than 2.6.35 have PTP enabled *Gianfar* network driver. Since Version 3.0 it has a new *struct ptp_clock_info* by which clock drivers can register themselves to the class driver. Then the kernel treats the external clock (i.e., *TMR_CNT_H/L* registers) as a character device. After the device is opened as a normal file descriptor, the clock control operation in user space program can be done by system calls, such as *clock_gettime and clock_settime*, are available since kernel Version 2.6. And the reading HW assisted timestamp is done by reading functions, e.g. *read*. New features added to Version 3.0 and corresponding operations are shown in Figure 3.5. More detailed information can be found at http://www.kernel.org/doc/Documentation/ptp/ptp.txt.

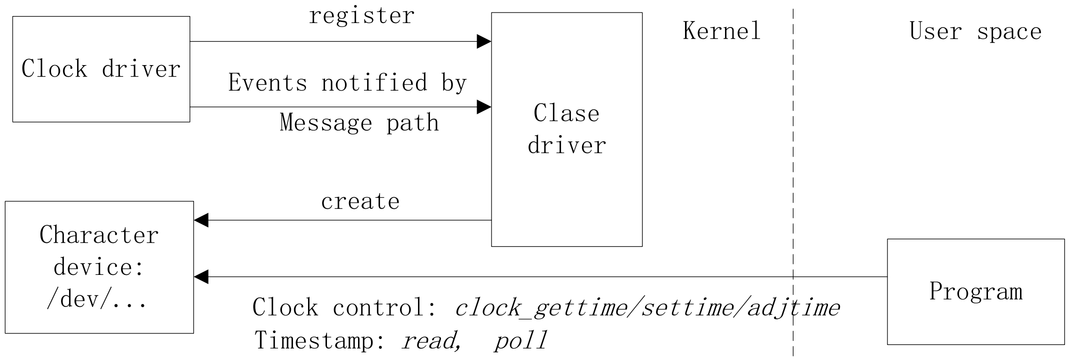

register
Clock driver
Events notified by Message path
Clase driver
Kernel
User space
create
Character device: /dev/...
Program
Clock control: clock_gettime/settime/adjtime
Timestamp: read, poll

Figure 3.5. *New operation in kernel Version 3.0.*

3.4. Phase aligned operation

As shown in Figure 3.1, without *TMR_ALARM1*, *TMR_FIPER1* is enough to generate the pulse signal. If the rising edge of pulse needs to be phase aligned to a reference pulse, *TMR_ALARM1* register is supposed to be configured. Although tuning OCXO frequency is not based on the pulse rising edge, this operation helps tests. In order to compare P2020 PPS signal against the GPS reference signal, the oscilloscope displays both PPS in one screen. The way to achieve phase aligned output PPS signal is to follow the instruction in [14. p, 787]. What deserves special attention is that the phase alignment enabling should be turned on before the timer starts, or the phase shift would behave abnormally.

3.5. Summary

At the beginning of the chapter, the PTP HW structure in the P2020 processor is introduced, including how to calculate the proper value in registers and the way to generate PPS signal. The second section explained how to adjust the network driver – *Gianfar* in Linux kernel 2.6.34.x to adapt the PTP ancillary HW clock features, including:
1. Initialize registers;
2. Recognize PTP packets;
3. Notify HW to give timestamps on packets;
4. Store timestamps;
5. Deliver them to user space SW when requested.

Additionally, at the end of this section, it was explained shortly how the higher version kernel manipulates these features.

After both the HW and the OS are ready to use, the user space SW that implements the PTP slave clock is going to be explained next.

4. PTP SLAVE SW

The goal of this thesis is to see how PTP can be employed in mobile communication systems, given a GPS connected, product level grandmaster acting as the most accurate timing source. It is also important to understand, how harsh the network environment can be until it heavily affects the synchronization of the slave. Because the slave SW exchanges packets with the master, the first step is to implement the network handler.

4.1. Network handler

IEEE 1588v2 complies with both multicast and unicast. Not all routers support the Internet Group Management Protocol (IGMP), which is necessary for multicast. The built up instructions for general unicast communications is given in the standard Section 16.1. The standard also offers several transport layer and network layer options. In this thesis, unicast over UDP/IP is used.

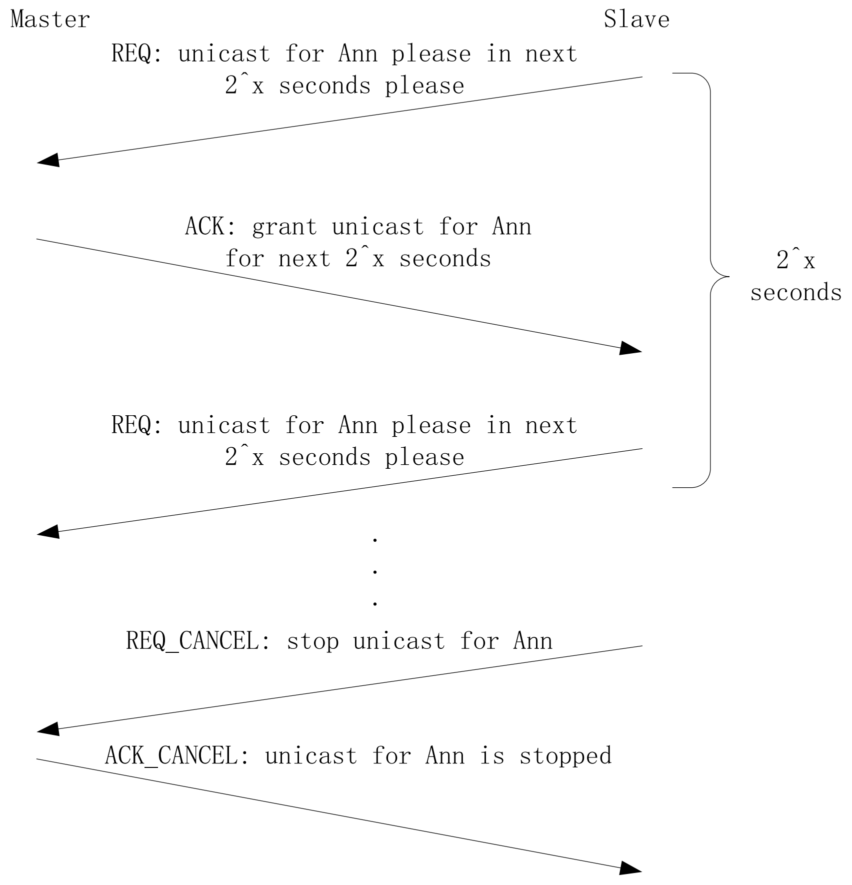
As default, the master clock is not in the unicast mode, and the unicast mode needs to be invoked by the slave. In other words, the master clock is unicast passive. The unicast requests sent by slaves are suitable for three types of PTP messages. Figure 4.1 shows the unicast negotiation for *Announce* messages in normal situation, as well as the *Delay_Resp* and *Sync* messages.

The unicast request contains key information – the unicast packet type (*Announce*, *Sync* or *Delay_Resp*), the duration in which the request is valid and the log interval between successive corresponding messages. If a slave expects to have only unicast PTP, it has to resend the same request before the former one expires. Otherwise, the master clock usually stops unicasting that type of message after the request expires. Although the slave invokes the unicast request with a duration it expects, masters decide if a different duration should be granted according to both its capacity and the regular routines. The unicast response from master therefore has three possibilities:

- Grant is denied – duration field is 0.
- Granted, but duration is different from that in the request.
- Granted for the duration in request.

Upon reception of a unicast acknowledgement packet, slaves check if the duration is different from its expectation, and set the correct timer for transmitting the next unicast request.

Similarly, the item of log interval also needs to be checked. On the one hand, many PTP studies claimed that the interval between *Sync* packets would be meaningless if it is bigger than 4 seconds. It is commonly considered that too slow event packets fail to feature the complex network environment. On the other hand, the master clock has limitation on how many, e.g., *Sync* packets, can be transmitted in one second. According to the standard, the log interval items in the headers of *Sync* and *Delay_Resp* messages are not used in the unicast mode.



Master
Slave
REQ: unicast for Ann please in next 2^x seconds please
ACK: grant unicast for Ann for next 2^x seconds
2^x seconds
REQ: unicast for Ann please in next 2^x seconds please
REQ_CANCEL: stop unicast for Ann
ACK_CANCEL: unicast for Ann is stopped

Figure 4.1. *Unicast Announce message negotiation.*

The standard declares that a slave could send a “unicast cancel” *Signaling* message to inform the master that unicast is no longer needed. This design is mainly for decreasing the network traffic amount, but since accurate timing protocol is usually crucial to ensure that machines function normally, it is always turned on. Therefore, in this thesis, the network handler does not send such *Signaling* messages. Moreover, in the standard there is another unicast negotiation service called "unicast enable" by exchanging *Signaling* packets. However, the OSCILLOQUARTZ 5331 master used in this thesis can be manually configured as unicast through web interface, so that the “unicast enable” service was not built in the slave network handler.

It is worth mentioning that the unicast valid duration in request does not follow ‘the longer the better’ rule, although longer duration might save some trouble from the slave, such as processing speed. For example, in a dynamic network environment, with cascaded PTP functions, slaves need to decide the time source to synchronize to, which is part of the BMC. The selection is completed by comparing clock quality information in *Announce* message. If the duration is very long, and there are many clock sources to choose from, one result would be that masters that are not chosen to be the best master keep sending *Announce* messages until *Announce* unicast requests expire. This slows the slave processing speed as it needs to look up master list, and drop the packets from these masters. A better choice might be: before the best master clock is chosen, a shorter duration is used for the *Announce* message. This is easier than sending “unicast cancel” *Signaling* messages.

Because UDP is not reliable, it is the application layer's responsibility to ensure that the packets are correctly delivered. For *Signaling* messages as introduced above, it is important to make sure that the master receives them. The slave SW could turn on a timer that has an assumed Round Trip Time (RTT). If the slave fails in receiving any response from the master before the timer expires, it retransmits that *Signaling* message and restarts the timer. The method to estimate RTT could be based on the Linux kernel code of TCP. Alternatively, a method to calculate RTT from the network jitter is given in [25]. On the contrary, it is meaningless to retransmit PTP event packets in case of packet loss. Because event packets are sensitive to path delays, the "late" packets cannot be used to adjust the local clock. They introduce big accidental errors. Arriving in a wrong order can be detected by the sequence values in PTP headers. They are also indicators for the network environment. Based on the above explanations, in the PTP slave SW, losses of normal packets deserve retransmission, while reordered event packets can be silently discarded.

Figure 4.2 is an example of unicast negotiation for an *Announce* message, but the same mechanism also applies for the other two types. The first time a slave sends its unicast request, it would blocked at waiting for the master acknowledgement as the communication link is not built up yet. After that, the slave would focus on handling the other incoming PTP packets, until *Timer2* triggers a signal to force the slave to retransmit request packets. Moreover, in an implementation, one can replace multiple timers with a timer event sequence.

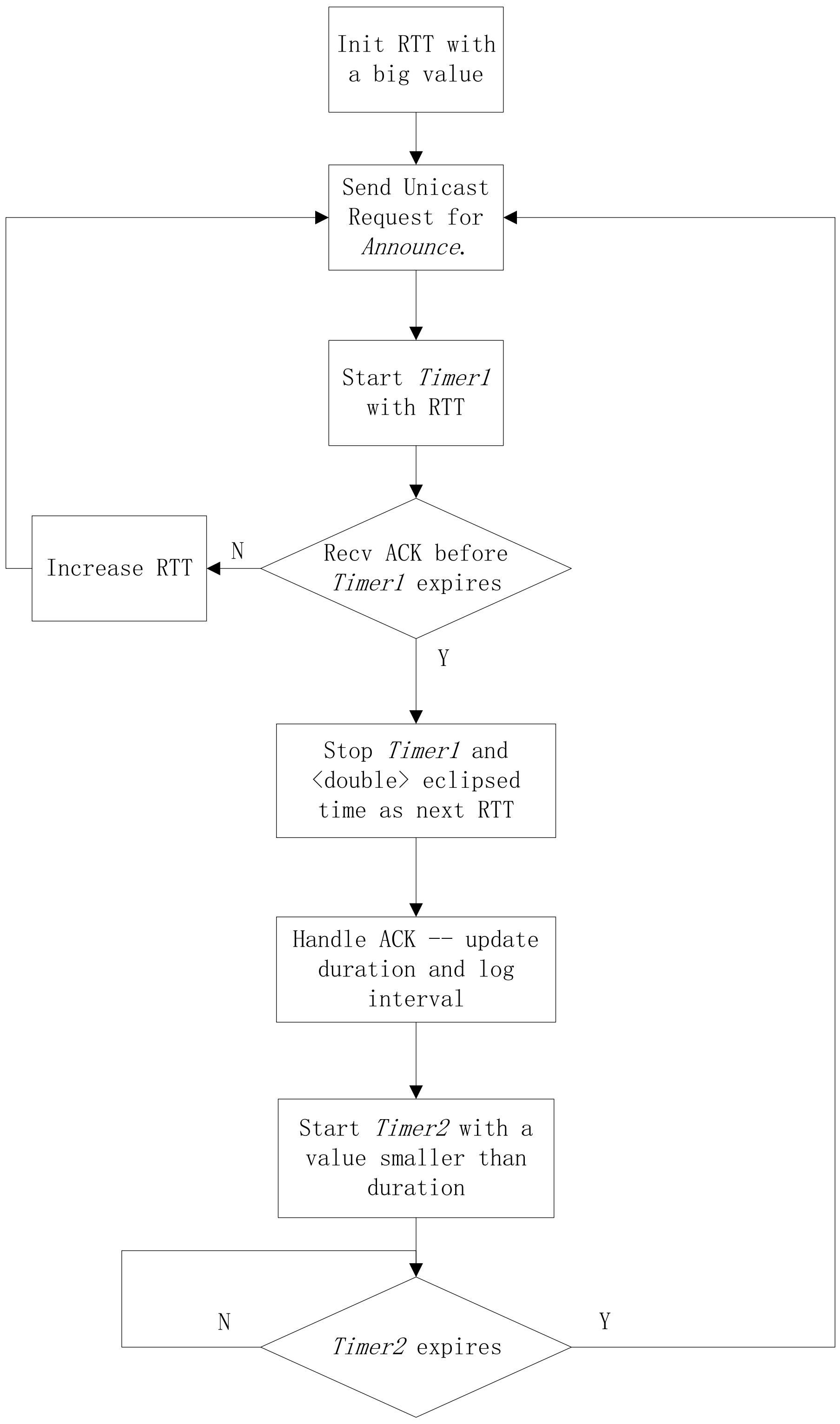

Init RTT with a big value
Send Unicast Request for *Announce*.
Start *Timer1* with RTT
Recv ACK before *Timer1* expires
N
Increase RTT
Y
Stop *Timer1* and <double> eclipsed time as next RTT
Handle ACK -- update duration and log interval
Start *Timer2* with a value smaller than duration
Timer2 expires
N
Y

Figure 4.2. *A flow chart of the Announce message handler.*

4.2. Tuple list

As explained in Section 4.1, PTP over UDP is not reliable. PTP event packets, *Sync, Delay_Req* and *Delay_Resp*, are collected for they contain time information that is used as the local clock tuning basis. However, one or more packets' loss would break this information chain, which requires special treatments.

...

TUPLE[n]	TUPLE[n+1]	TUPLE[n+2]
Sync Seq	*Sync Seq*	*Sync Seq*
t1	*t1*	*t1*
accurate *t1*		accurate *t1*
t2	*t2*	*t2*
Delay_Req Seq	*Delay_Req Seq*	*Delay_Req Seq*
t3	*t3*	*t3*
t4	*t4*	
Flag	Flag	Flag

...

Figure 4.3. *Tuple structure.*

The term "tuple" represents a set of collected information that is necessary to form an adjustment basis, including timestamps and the calculated delays and offsets. Figure 4.3 shows three main cases how a tuple could be affected by the network: no packet loss in TUPLE[n], loss of *Follow_Up* in TUPLE[n+1] and loss of *Delay_Resp* in TUPLE[n+2], respectively. The former two cases do not affect the offset and delay calculation, but the last situation fails to offer enough data. Every time a slave receives a valid event packet, it turns on the corresponding bit in the *Flag* field, and fills in the corresponding timestamp. *t1* and *t2* are filled in together like a timestamp pair at the reception of each *Sync* message. Upon receiving a valid *Sync* message, the slave transmits a *Delay_Resp* message and fills in *t3* and the *Delay_Req* sequence ID (*Delay_Req Seq*). If a following received *Delay_Resp Seq* matches the stored *Delay_Req Seq*, the received *Delay_Resp* is considered as a valid *Delay_Resp*. *t4* and *Flag* would be filled in and fixed respectively. Because the master may never receive a *Delay_Req* sent by slave, it is not always true that the received *Delay_Resp* is for the *Delay_Req*. *Seq* field plays such a role that it checks if the received *Delay_Resp* belongs to the same tuple, i.e., *t4* is a pair of *t3*.

After the slave fills in *t4*, it checks the *Flag* item to decide if all the necessary event packets are received, and then calculates the offset and delay. If the type of *Flag* is an 8-bit char, Figure 4.4 shows the used *Flag* structure.

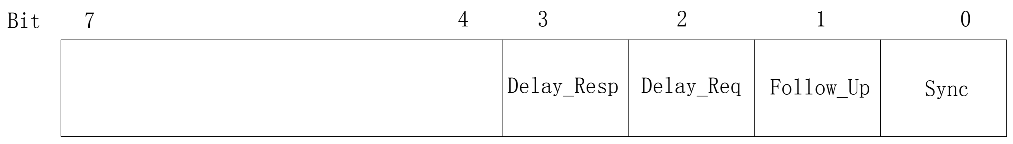


Figure 4.4. *Flag.*

If all packets in Figure 2.3 are sent and received correctly, *Flag* is 0xF, offset and delay calculation would be,

$$\begin{cases} \text{offset} = t2\text{-accurate\_}t1\text{-delay} \\ \text{delay} = \dfrac{t2\text{-accurate\_}t1}{2} + \dfrac{t4\text{-}t3}{2} \end{cases} \tag{4.1}$$

If *Follow_Up* is missing, *Flag* is 0xD, and accurate_*t1* is replaced by *t1* in (4.1). After delay and offset are obtained, the head of tuple is moved onto the next one, ready for constructing new tuples; meanwhile the slave enters the servo phase.

In case of *Delay_Resp* losses or it arrives later than the *Sync* message (detected by checking if the *t1* in the head of tuple is already written), the head of tuple would directly move to the next one. However, this deserves a second thought, because when the interval of *Sync* message is smaller than the slave-to-master delay, or at least these two quantities are of the same order, a competition of sending *Sync* and sending *Delay_Resp* may happen. It is a dilemma to set timing threshold to distinguish if a *Delay_Resp* is still usable or it suffers too big network delay. Because of this reason, the length of tuple list depends on the *Sync* interval. During the test of this implementation, when the frequency of *Sync* is set to 64 pkt/s, the incomplete tuples that have lost *t4* show up more often than what has been observed when the frequency is 1 pkt/s.

As mentioned in Section 4.1, only the newest event packet is accepted – *OrigTimestamp* field in a newly received *Sync* and *Delay_Resp* would be checked, if it is not newer than the corresponding type message in tuple, it would be discarded. In this way, both duplicated packet (same timestamps) and out-of-order packets would be discarded. The handler for receiving event packets is shown in Figure 4.5.

Although Figure 4.5 only shows the *Delay_Resp* and *Sync* messages, in a tuple structure as Figure 4.3, the *Follow_Up* message is handled in a similar way as *Delay_Resp* packet. First, look up a matching *Sync Seq*, if such a tuple exists and *Follow_Up* bit is not set in *Flag*, then write the accurate *t1* and set the bit in *Flag*. In this implementation, the delay and offset is calculated just after *t4* is obtained, so the accurate *t1* is only filled in for the newest tuples.

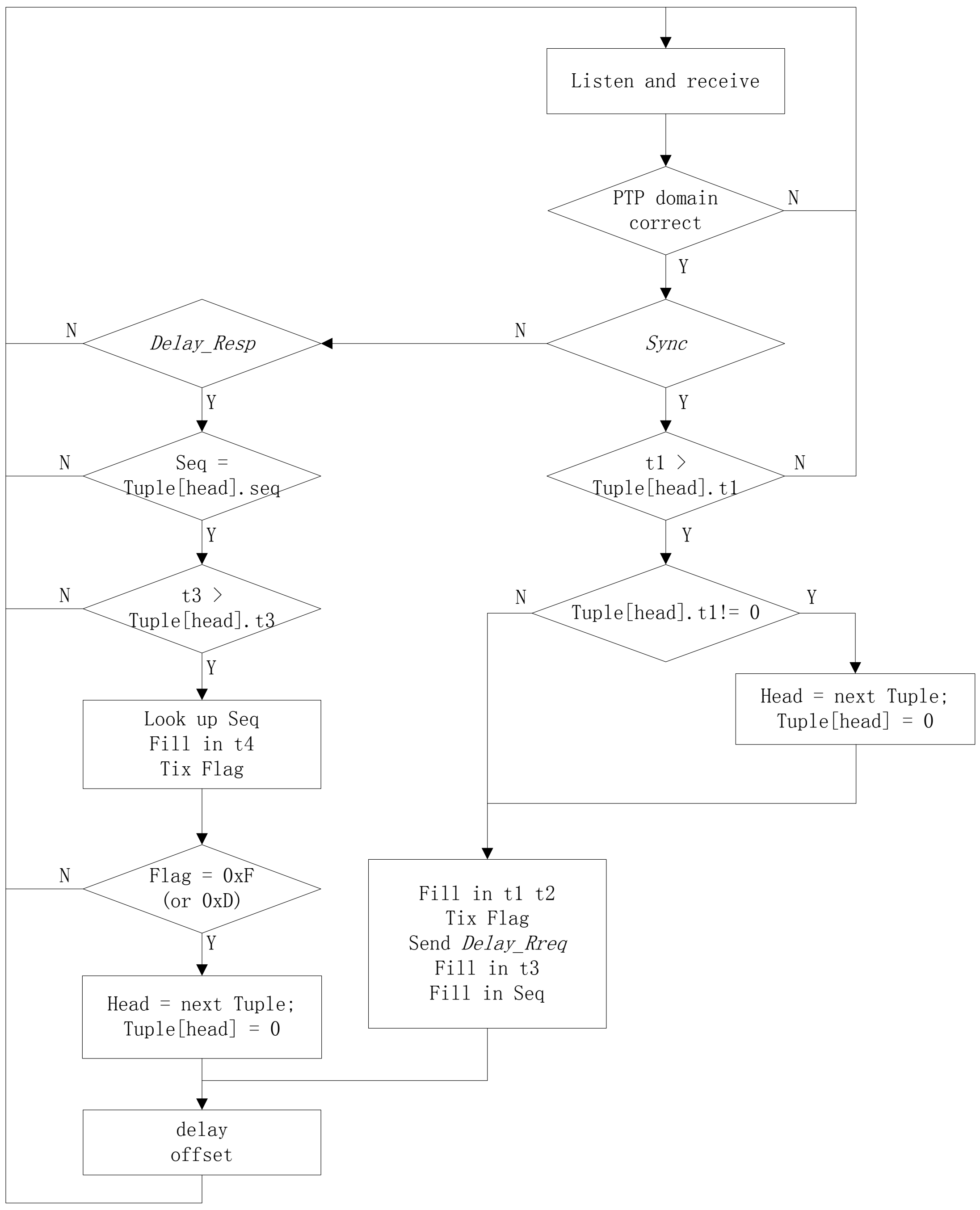


Figure 4.5. *Event packets organizer – tuple list operation.*

Whether a two-step PTP should be employed mostly depends on the structure of processor. For example, the processor of our slave has timestamps tagged at the MAC layer, but the *OrigTimestamp* field in *Sync* message needs to be handled at application layer. If a same processor is also used in a master, the *Follow_Up* message must be used to carry the accurate timestamp. Experimentally, it is observed that the accuracy is improved by around 60 μs.

4.3. Filters

The goal of this subsection is to introduce methods that are tested in the thesis implementation for suppressing the effect of PDV. Large PDV of the network degrades the performance of PTP. Lowpass filters (LPF) can be used to reduce the PDV. On the one hand, it observed that the effect of PDV is much higher than the variation of clock wander. On the other hand, HW faults would also cause the calculated offset to be much larger than it actually is.

4.3.1. Offset thresholds

Step controllers and popcorn spike suppressors are used to filter out the samples over thresholds. The former ones are used for samples which are likely to have suffered from HW faults and the latter ones are used for samples with exceptionally large PDVs.

A step controller is to discard the tuples whose offset is more than 128 ms unless this situation lasts for 15 minutes [4]. A single burst that stepped over the threshold was observed during this implementation at the very beginning. First, this phenomenon was caused by the HW structure. If the *Delay_Req* sequence ID is initialized to be the same as the value already stored in the *TMR_TXTS_ID1/2* registers (left by the previous run of the SW), a race between HW updating and SW reading appears, and usually the latter wins. This HW fault results in not only a huge offset but also a huge delay. It can be recognized and thereafter discarded by setting two thresholds on offset and delay individually. In order to avoid this ambiguous initialization, the SW may need to keep a log where the status is read/written at the beginning/end of its execution. Secondly, if the SW is turned on after a long silence, the clock bias easily steps over 128 ms. The solution is to step the *TMR_CNT_H/L* value forcibly, because writing to this register is to directly change the time offset. Compared with the huge offset in the HW fault situation, there is no accompanied abnormal delay in the second scenario. Therefore, delay threshold can be used to distinguish two cases, which are handled differently.

Unlike the step controller, the threshold (jitter) of the popcorn suppressor is dynamic and determined by the calculated clock offsets. Popcorn spike suppressor is described in NTPv4 as "Compare the difference between the last and current offsets to the current jitter. If greater than SGATE (3) and if the interval since the last offset is less than twice the system poll interval, dump the spike."[4]. Once a popcorn spike is detected, it is dropped. The popcorn spike detector is characterized as:

$$J^2 = \frac{1}{L} \cdot \sum_{i=1}^{L} {d_i}^2 \tag{4.2}$$

$$d_i = s_i - s_{i-1} \tag{4.3}$$

$$|d_L| > 3 \cdot J \text{ and } Sync\_seq_L = Sync\_seq_{L-1} + 1 \tag{4.4}$$

L – The length of widow,
J – Jitter,
s_i – Offset, s_L is the current offset,
d_i – Difference between offset and its former offset.

For every newly calculated offset, its difference from the former one is calculated as (4.3). The result is applied to equation (4.2) to get an updated jitter value. If an offset is changed more than three times of the current jitter value, it is considered as a popcorn spike. As what has been explained in Section 4.2, it is possible that there is one missing tuple. Therefore, the popcorn spike suppressor acts only if the previous two tuples have been received in sequence, i.e., if $Sync\_seq$ is increased only by 1.

The popcorn spike suppressor is very useful in practice. In our tests, a single offset spike was randomly observed and thereafter dropped.

4.3.2. Packet selection

In NTPv4, before the estimated offset is finally used to discipline the clock, it is passed through *clock filter*, *selection*, *cluster*, and *combine* algorithms. They act like LPFs by dropping the aberrant samples that have large delays.

The clocks *A*, *B*, *C*... are candidates for a slave to synchronize to. For each of them, the slave maintains a tuple list and every time a new tuple arrives, the smallest delayed one within the processing window survives, and is passed to the *selection* stage. In the *selection* algorithm, the offset and its variation would be used to calculate the upper and lower limits. It compares the other offsets and counts how many of them fall within these two boundaries. The clock that has most other clocks in its boundaries is considered as the best, and clocks within the intersection zones survive. The selection of the intersection clocks is completed in the *cluster* algorithm. The offsets from these survived clocks are merged in the *combine* algorithm as a weighted average:

$$\frac{\sum_{i=1}^{i=N} \frac{f_i}{d_i}}{\sum_{i=1}^{i=N} \frac{1}{d_i}} \tag{4.5}$$

where *N* is the number of clock survivors, *d* is the root distance and *f* is the offset. The root distances are calculated based on both the physical distance and the jitter of each clock obtained by (4.2).

The idea of NTPv4 filters are shown in Figure 4.6, with details omitted. For example, in NTPv4, in order to simplify the detection of the smallest and newest delay, the tuple list is sorted first. *f.A* in Figure 4.5 means offset from clock *A*, *v.A* represents variation of *A*.

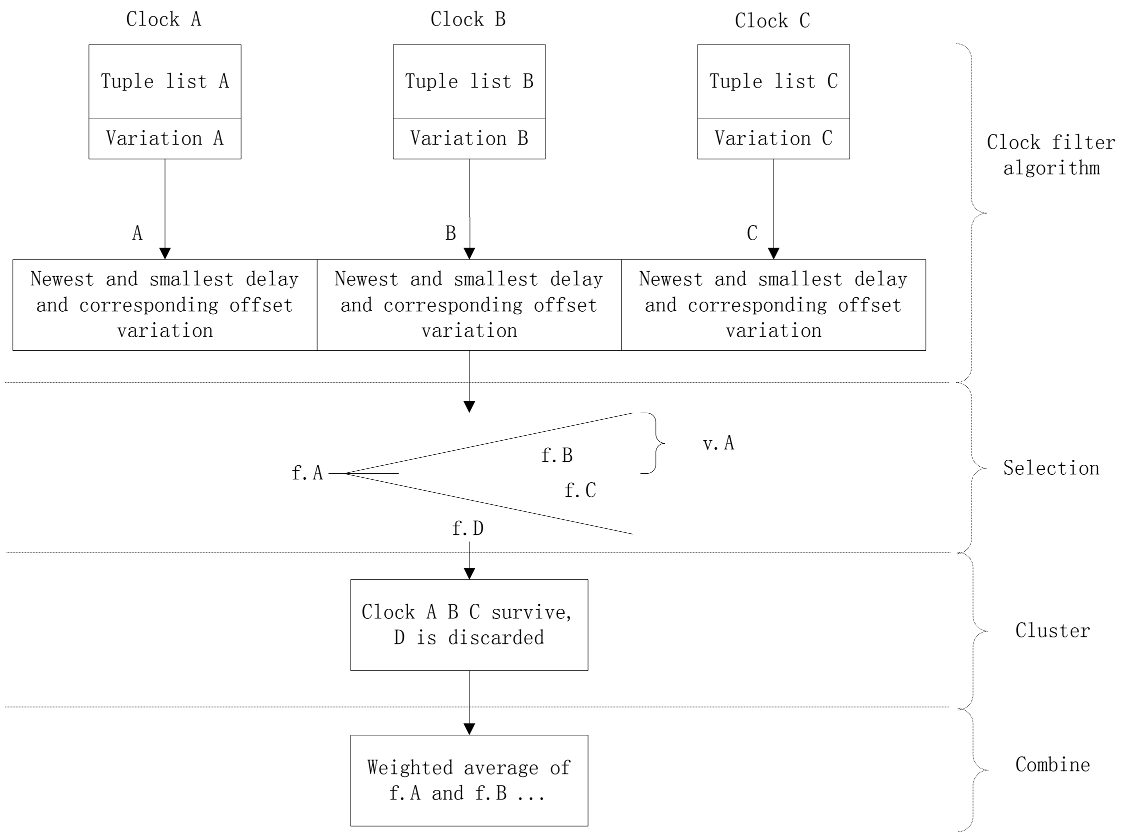

Clock A
Clock B
Clock C
Tuple list A
Tuple list B
Tuple list C
Variation A
Variation B
Variation C
Clock filter algorithm
A
B
C
Newest and smallest delay and corresponding offset variation
Newest and smallest delay and corresponding offset variation
Newest and smallest delay and corresponding offset variation
v.A
f.B
f.A
f.C
Selection
f.D
Clock A B C survive, D is discarded
Cluster
Weighted average of f.A and f.B ...
Combine

Figure 4.6. *Filters in NTPv4.*

Because in this implementation only one master clock was used, *selection* and *combine* algorithms were not used. However, in a robust telecommunication system, there would be for sure more than one master clock for slaves to synchronize to. Besides, the BMC algorithm stated in PTP gives another method to choose the master clock, based on the masters' "self-confidence". If every master clock describes itself as the best one (carried in the *Announce* message), methods introduced in NTP could be used.

The modified *clock filter* algorithm is shown in Figure 4.7. Every time a tuple is generated, it is inserted into a sorted array (according to its generation time). Then the tuple whose delay is the smallest is passed to the next stage. The other tuples are not used but still remained in the array. During our experiments, the update took place typically every four tuples (when the length of array is 10). In NTPv4, the array length *L* is 8.

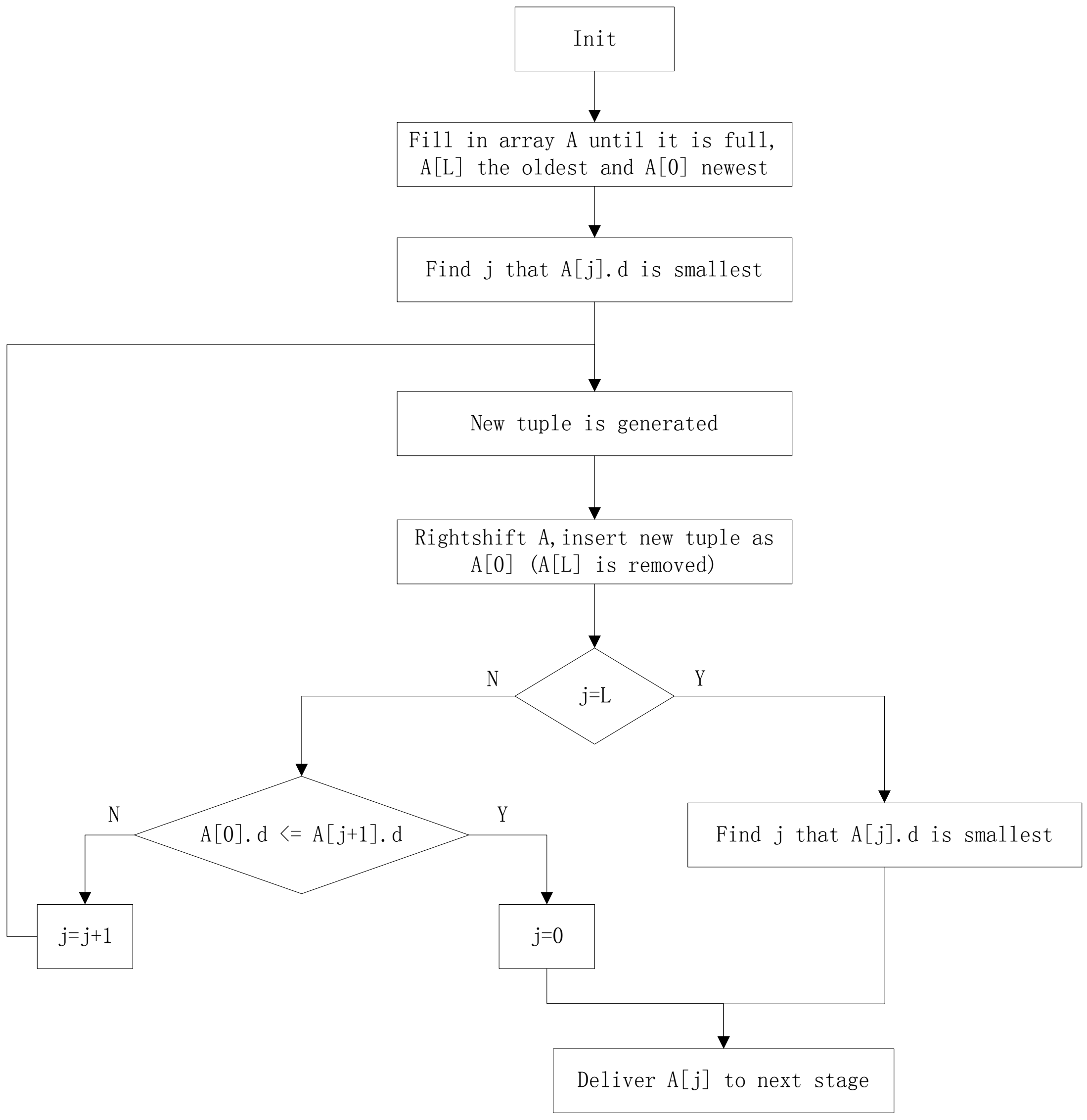

Init
Fill in array A until it is full, A[L] the oldest and A[0] newest
Find j that A[j].d is smallest
New tuple is generated
Rightshift A, insert new tuple as A[0] (A[L] is removed)
j=L
N
Y
A[0].d <= A[j+1].d
N
Y
Find j that A[j].d is smallest
j=j+1
j=0
Deliver A[j] to next stage

Figure 4.7. *Clock filter algorithm.*

In NTPv4, the array length L = 8. The real clock filter in NTPv4 is a little different from the one shown in Figure 4.7. Especially, every time a new tuple arrives, it is filled in the array sorted by the arriving time, and then array is sorted again according to the elements’ delays.

4.3.3. Digital filters

The huff-n'-puff filter was claimed to be the only method that helps to reduce the asymmetric delay. It has three prerequisites – severe asymmetry, multiple links available but only one of them is relatively slow, and the slave clock offset is very small [29, p.54]. Figure 4.8 shows the concepts of Huff-n'-puff filter. Among all the offset-delay pairs, the filter first selects the one that has smallest delay, i.e., the vertices of the scattergram. The other pairs are compared with the selected vertices. If the offset is bigger than that of the vertices, half of the delay difference is subtracted from the calculated offset, and vice versa. However, the author did not quantify the used descriptions, e.g., "small offset" and "severe asymmetry". The author also claimed that “the wedge scattergram plots sample points (*x*, *y*) corresponding to the measured delay and offset” [29, p.44 Figure 3.4], which serves as the filter basis. However, by analysing the test result of the slave SW, the phenomena were not like that. *Offset-delay* graph is not a wedge, *offsetChange-delayChange* is instead. Also by examining Figure 2.4, delay and offset do not have a direct relation. For example, if all delays in Figure 2.4 remain without change, which means the "Mean Delay" will not change, *t2-t1* becomes bigger, and the biased offset would be bigger – same delay but bigger offset. Besides, implementations with and without Huff-n'-puff filter did not show obvious performance difference. It is highly possible that the test of this thesis implementation did not satisfy the entire prerequisite. But based on the description and the Huff-n'-puff filter’s place in the filter chain (after *clock filter* algorithm), I venture to guess that this filter would be helpful if the heavy asymmetric path delay has lasted for up to tens of minutes or even hours (*clock filter* gives the apex of wedge with a window of several seconds). Because of the above reasons, Huff-n'-puff filter is not used in the thesis implementation.

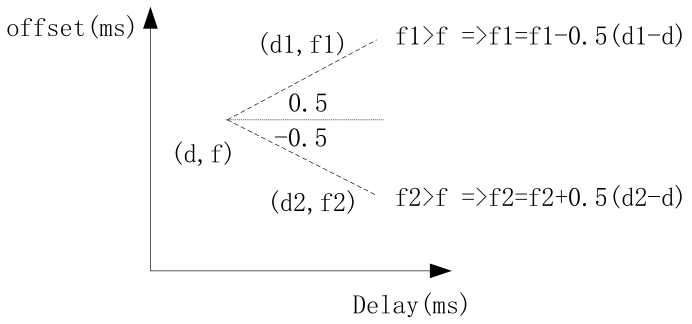


Figure 4.8. *Huff-n'-puff offset processing.*

The PTP daemon (PTPd) is an open source project of PTP [24]. In PTPd, the delay and offset are filtered separately. After delays are calculated according to (2.4), they are processed by an Infinite Impulse Response (IIR) filter. Substituting the processed delay into the second equation in (2.4), the derived offset will be passed to an FIR filter. Figure 4.9 presents this procedure.

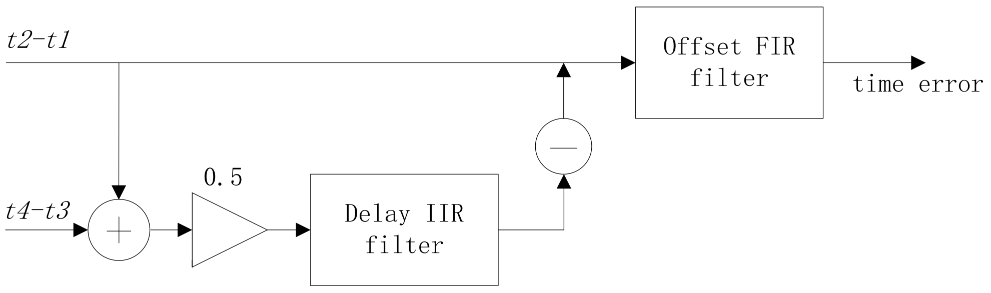


Figure 4.9. *PTPd offset and delay filtering structure.*

The difference equation of an IIR delay filter is

$$\mathrm{y}[n] = \frac{(s-1) * y[n-1] + \frac{x[n] + x[n-1]}{2}}{s} \tag{4.6}$$

in which $x[n]$ and $x[n-1]$ are current and former delay samples, $\mathrm{y}[n-1]$ is the former IIR filter output and s is a positive integer which stands for the stiffness.

Its transfer function is

$$H[z] = \frac{Y[z]}{X[z]} = \frac{\frac{1}{2s} + \frac{1}{2s} z^{-1}}{1 - \frac{s-1}{s} z^{-1}} \tag{4.7}$$

This is a first-order IIR filter and it has one single zero at -1, and only a single pole

$$z = \frac{\mathrm{s}-1}{\mathrm{s}} \tag{4.9}$$

Based on (4.9), because *s* is positive, the pole is always within the unit circle, meaning this filter is always stable. Moreover, the amplitude response of the filter would have a peak at frequency 0. The bigger *s* is the closer *z* is to the unit circle, resulting in a sharper amplitude response. Additionally, because -1 is the zero point, the frequency π has zero amplitude response. This is shown in Figure 4.10.

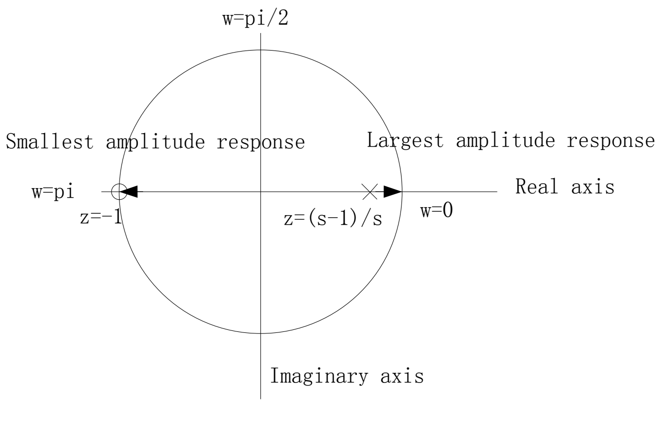


Figure 4.10. *Zero and pole of the first-order IIR filter.*

With the following Matlab code, simulated amplitude response is shown in Figure 4.11

```
x = [1 1];
s = 4;
y = [2*s -2*(s-1)];
[X,w] = freqz(x,y);
plot(w/pi, abs(X));
grid;
```

Running the simulation with different *s*, the result that is shown in Figure 4.11 confirms the above assertion – the bigger *s*, the better frequency selectivity.

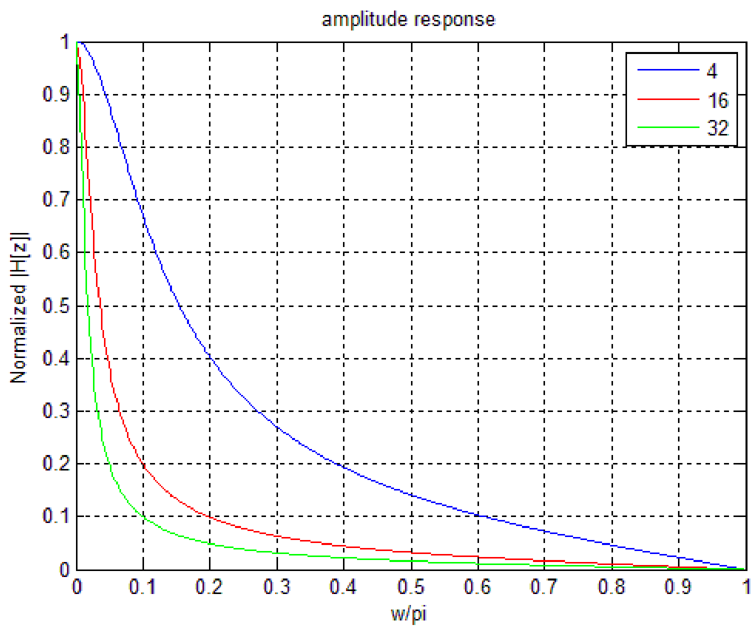


Figure 4.11. *Amplitude response with different values of coefficient s.*

The output of the offset filter is simply an average of two consecutive inputs. Taking the mid value of two inputs would decrease the difference between two samples, making it a LPF. Its time domain expression is

$$y[n] = \frac{x[n-1] + x[n]}{2} \tag{4.10}$$

Its transfer function is

$$H[z] = \frac{Y[z]}{X[z]} = \frac{\sum_{k=0}^{M} a_k z^{-k}}{1 - \sum_{k=1}^{N} b_k z^{-k}} = \frac{\frac{1}{2} + \frac{1}{2} z^{-1}}{1} = \frac{\frac{1}{2} z + \frac{1}{2}}{z} \tag{4.11}$$

Based on (4.11), its amplitude response is

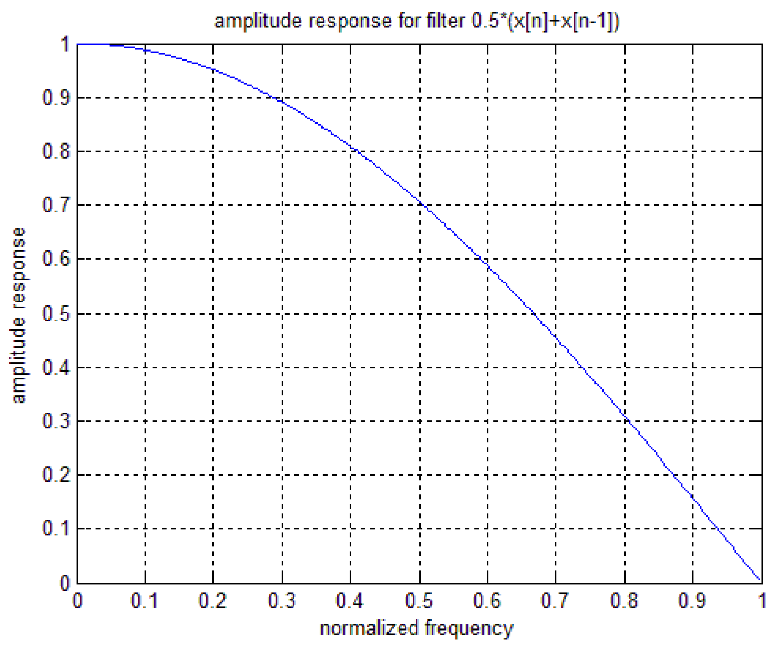


Figure 4.12. *Amplitude responses of the FIR filter (4.10).*

4.4. PI controller

The filtered offset is then delivered to the PI controller. The integrator was not considered when FLL was implemented in this thesis, which could be one reason that results in an unstable clock. One over adjustment followed by another overshoot (opposite direction) was often observed. The PLL method contains integrations. Overshoot still appeared, but a lot less. This comparison shows that the integration in PI controller helps to reduce residual error. The time domain expression of a discrete PI controller is

$$y[n] = K_p \cdot x[n] + K_I \cdot \sum_{N=0}^{n} x[n-N] \tag{4.12}$$

$x[n]$ – Filtered offset,
$y[n]$ – Adjustment,
K_p – Proportional gain,
K_I – Integral gain.

Its transfer function is

$$H[z] = \frac{Y[z]}{X[z]} = K_p + K_I \frac{z}{z-1} \tag{4.13}$$

Because the pole is on the unit circle, the PI controller is conditionally stable. A thorough analysis of the PI controller stability can be found in [10, p. 152].

4.5. Clock discipline methods

Similar to that in NTP, clock discipline algorithm is a method used to adjust the local clock, based on the processed error (offset).

In NTP, the discipline algorithm adopts FLL and PLL [27]. PLL is used to directly adjust the clock phase and is shown in Figure 4.13.

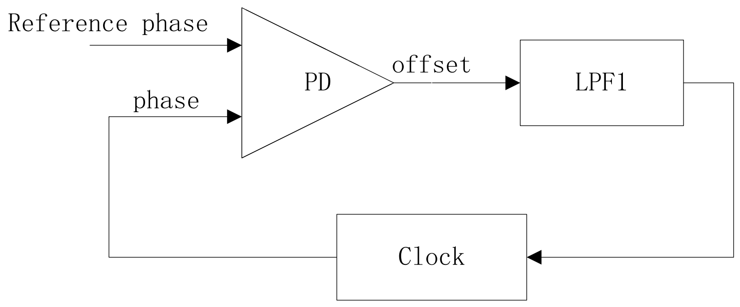


Figure 4.13. *PLL.*

According to [26, p. 12], in the simplest occasion, a PLL is formed by a Voltage Controlled Oscillator (VCO), i.e., clock, and a Phase Detector (PD). In order to suppress the jitter and fluctuation, an LPF can be added into the PLL. This is type I PLL. And in order to minimize the residual error (similar to the residual error in a filter stop band), an integral component is added, which makes it a type II PLL. A type II PLL is implemented in NTP.

The FLL method is described in [27, p. 2, 3]. Basically, as shown in Figure 4.14, the FLL method is to gain the clock frequency drift by observing the relative offset within a period (up to several minutes). This method is only helpful when the clock frequency wander dominates. Longer update interval makes the FLL more stable.

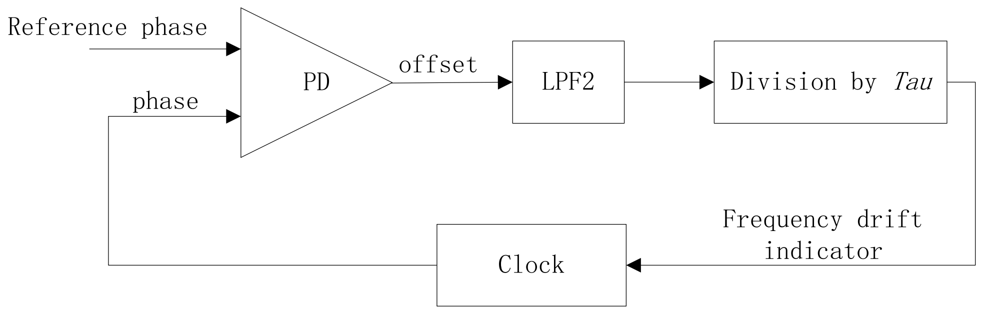


Figure 4.14. *FLL.*

The difference between FLL and PLL is that in FLL, instead of fixing the phase error directly, it adjusts the frequency of the clock by counting how much the phase wanders during the interval τ (*Tau* in figure). For example, at the moment of t_A/t_B, the time offset is s_A/s_B. The frequency shift can be calculated from the time offset during τ as

$$\begin{cases} f_w = \dfrac{s_B - s_A}{\tau} \\ \tau = t_B - t_A \end{cases} \tag{4.14}$$

It is claimed in [14] that the NTP clock discipline algorithm was designed independently from the clock structure, which can be a VCO or a SW clock. A precise time keeping method for SW clocks is given in [26] as a kernel model. And based on the demo code, Linux has a fixed tick rate (the macro HZ defined in the kernel). Then tuning the OS SW clock becomes changing the tick length, which is derived based on the above discipline methods. This is also the principle for *adjtimex* function.

4.6. Summary

In this chapter, according to their places in the slave SW, three main aspects were introduced. They are network handlers, data organizations and the data processing.

PTP was built over Ethernet, so the network handler was firstly introduced, this step contains the network connection control (i.e., handshake on *Signaling* message for unicast negotiation), and the reliability analysis of each PTP message type.

The tuple list section gives a flow graph on how to handle/store each received PTP messages using unreliable UDP. The goal of this tuple list is to ensure that enough information is gained to calculate the offset and delay.

The offset and delay offer the inputs for filters. Several interesting filters were discussed in a sequence of their position in the slave clock servo. They are:

- Filters that discard input according to thresholds,
- NTP algorithms that pick offset with less PDV,
- PTPd IIR delay filter and FIR offset filter,
- PI controller that finally generates the control signal.

If the tuning indicator of the system is the time offset, it is a PLL. For the FLL method, the indicator is the variation of time offsets.

This chapter, in combination of the P2020 HW registers introduced in Section 3.2, completes the introduction on how to build a slave servo.

5. PLL METHOD SIMULATION

In order to study the system behavior, and find the effect of parameters of the PI controller, the PLL method is simulated in Matlab. Besides, simulations could also evaluate the asymmetric path delay effect which is seen as an impossible task by the experimental evaluation.

The simulations were conducted in two cases:

- The polluted case where the delay effect on time offsets is simulated,
- The ideal case in which the delay does not exist and the system frequency does not vary.

The polluted case simulation is to inspect how PDV affects the synchronization accuracy. The one for the ideal case is to study how the system characteristics affect the slave clock performance.

5.1. Polluted case

The Matlab simulation program is given in the Appendix 1 .

The main.m file simulates the network handler and tuple list, and it offers the interfaces for the other files – decision.m and *updateofs.m*. In order to evaluate the effect of PDV or asymmetric path delay on the slave clock offsets, the first step is to generate path delays in two directions – *dms* and *dsm.* The statistical features of the simulated delays are drawn based on the real network delays, collected in one experimental test. Delays in two directions are generated separately (i.e., two random processes), their sum satisfies the derived expectations and variance (according to the Central Limit Theorem and variance properties). Moreover, considering the delay spikes in the experimental tests, a uniform distribution ($0 \leq$ range ≤ 1) array (*peak_on_delay*) is used. Once its element exceeds 0.99, the corresponding delay is amplified by 5, so the delay spike rate is 1%. More detailed considerations, e.g., the calculations of statistical features (variance and expectation) are explained in the comments in the Appendix 1. The second half of this script simulates the time offsets. The unbiased offset is equal to the time in the slave minus that in the master, and the biased offset differs from the unbiased offset in that it takes the asymmetric path delays into account. The unbiased offsets are then passed to the updateofs.m and the decision.m files. The second half of the main.m script, i.e., the servo process, is encapsulated in an iterator with its times stored in *loopcnt.* The *loopcnt* is also the number of simulation samples. At last, the script also initializes the state and parameters (*Sync* rate, PI controller parameters and the initial unbiased time offset).

The script updateofs.m simulates the IIR filter on delays and the FIR filter on offsets. The script takes the master-to-slave delay, the average delay and the unbiased offset from the main.m file as inputs, and returns the filtered delay and offset. Firstly, the variable s in this file is the stiffness in (4.6). The script adjusts the stiffness based on the estimated delays: the biggest exponent that makes 2^s just less than the delay. This is because the stiffness s not only sharpens the LPF amplitude response, but also helps to avoid the overflow in the multiplication operation. Secondly, according to the Figure 4.9, the input of the FIR offset filter can be characterized as

$$unbiased\_ofs + dms - flt\_delay \tag{5.1}$$

$unbiased\_ofs$ – unbiased offset,
dms – delay from master to slave,
flt_delay – filtered delay.

The decision.m file simulates the PI controller and the plant. The function receives the unbiased offset and filtered offset from the main.m. The filtered offset is the base to derive the addend adjustment, while the unbiased offset is used to calculate the time difference between the slave and the master. In order to achieve this, a PI controller is implemented, with *ai* and *ap* as its integrator and proportional coefficients. At the end of this file, the new unbiased offset for the moment when next "*Sync* message" arrives is calculated. The simulation assumes the "UDP connection" to be perfect with neither packet loss nor misordering.

In order to compare the effects of the asymmetric delay location, the simulation firstly generates delays in two directions (i.e., *dms* and *dsm*)separately, and stored them in files. And the random delay spikes are handled as

- for the case of *dms* heavier than the *dsm*, the *dsm* is loaded from the file, but the *dms* is equal to the difference between the stored average delay and *dsm*. (variance of dms is 4.1660e+5, dsm is 6.775e+5)
- in the case of *dsm* heavier than the *dms*, the above procedure is repeated with *dms* and *dsm* exchanged. (variance of dms is 7.090e+3, dsm is 4.17320e+5)
- for the case that delays are uniformly distributed in the two directions, the random delay spikes in the average delay are added to *dms* and *dsm* evenly. (variance of dms is 1.0613e+5, dsm is 1.0633e+5)

Figure 5.1 shows the biased offset and the unbiased offset as simulation results, and they conflict the experimental results by having spikes that are against the filtered delay. The proportional and integral parameters for the PI controller are 0.01 and 0.05. The biased offsets are heavily affected by the asymmetric path delays. Moreover, if the delay spikes are contained in the master-to-slave direction, the unbiased offset is more affected compared with the other two cases. This is because in equation (5.1), the timestamp *t2* is equal to the sum of the unbiased offset and the master-to-slave delay. When the master-to-slave delay is not the main issue in the link, as in both $dsm = dms$ and $dsm \gg dms$ cases in Figure 5.1, the unbiased offsets show equal fluctuations.

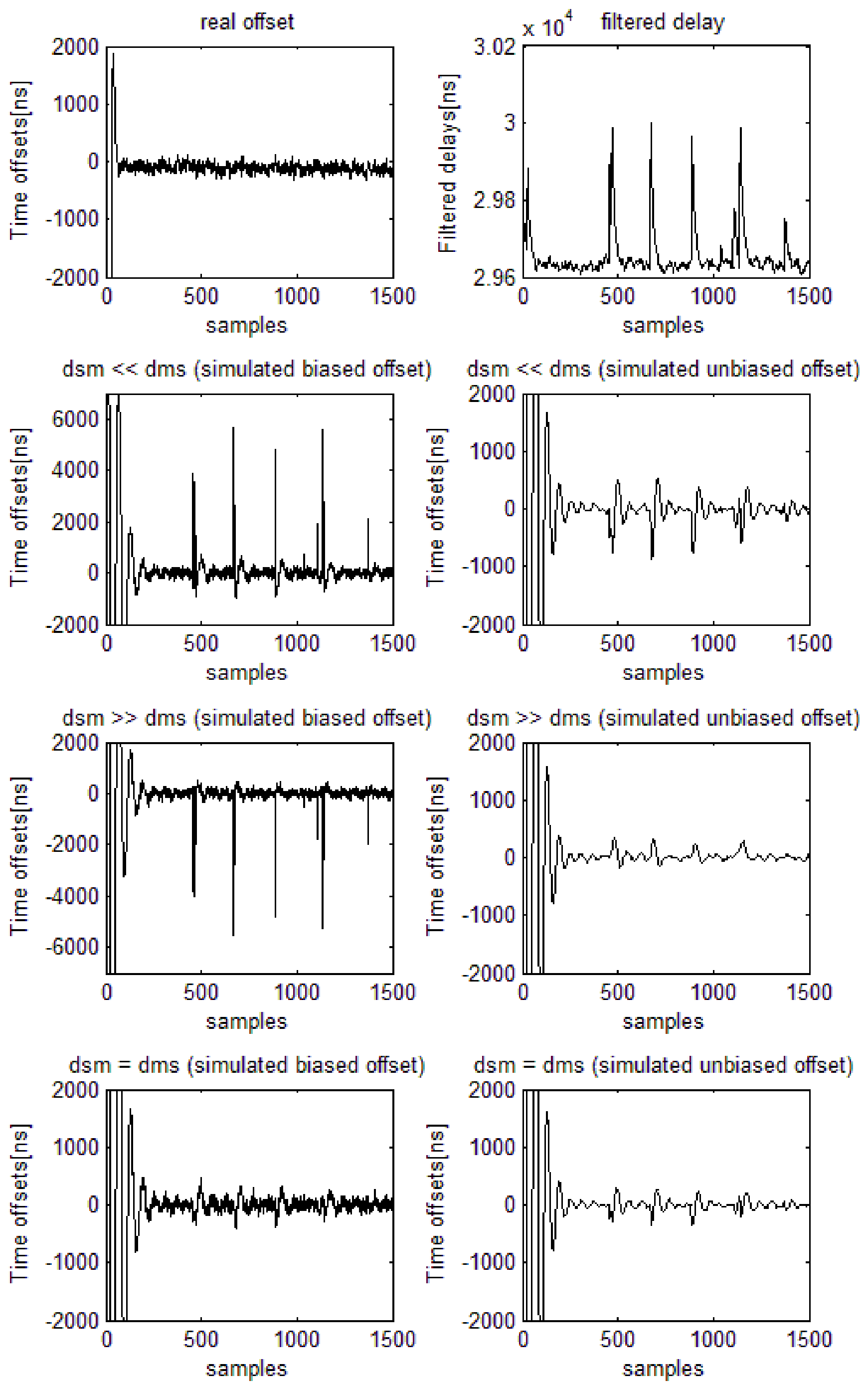


Figure 5.1. *Simulation results for three cases, offsets collected from an experiment (uppermost left), simulated filtered delays (uppermost right) followed by biased & unbiased offsets for each case.*

The absolute frequency drift corresponding to the simulations in Figure 5.1 are shown in Figure 5.2. The frequency stability depends on the networking PDV in the direction of master to slave. The large PDV in the master-to-slave link degrades the frequency stability in the slave. It is also observed during the experimental evaluation that the PPS signal from the slave clock wanders in steps, and the damping oscillation in Figure 5.2 testifies this.

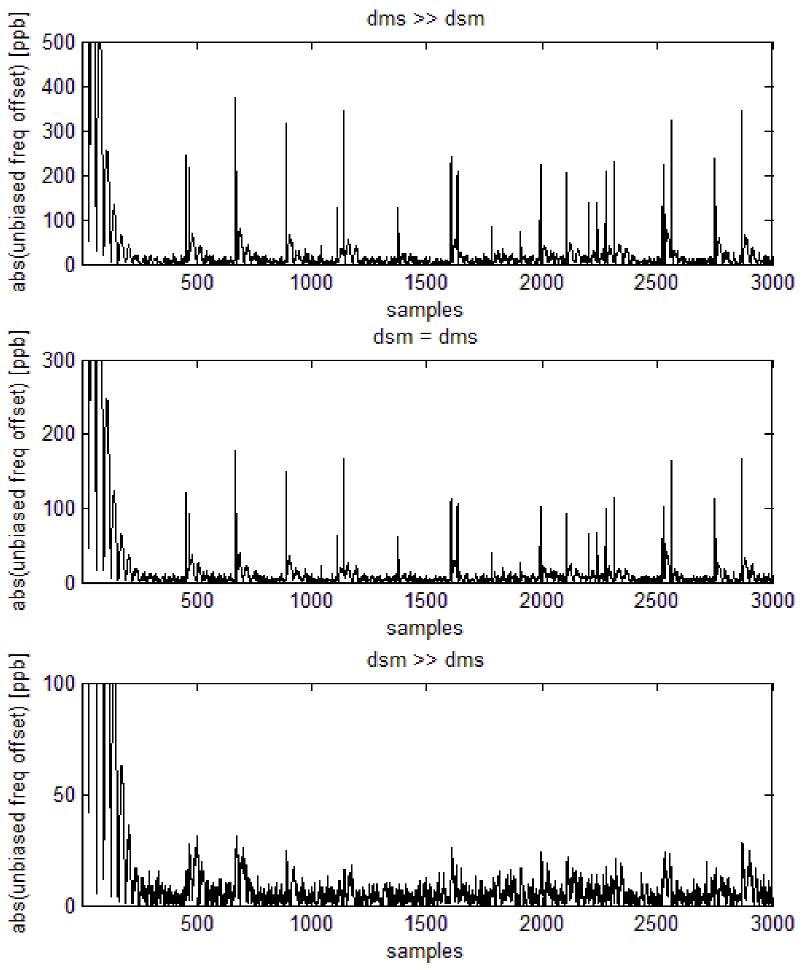


Figure 5.2. *Observed frequency offset in simulations.*

Figure 5.1 - 5.2 show that when the large PDV takes place in the master to slave direction, both the unbiased time offset and frequency drift exhibit serious fluctuation. Moreover, when the link in one direction is more delayed than the other one, the biased offset shows larger fluctuation is because of the asymmetric delay, namely the *dms* is incomparable with the filtered delay. However in the case of equal delay, the fluctuation in the offset related with delay spikes is due to the delay filter. Moreover, with parameters 0.01 and 0.05, the system is over damped in terms of time offset and frequency drift.

In another aspect, the simulated delay and filtered delay relation are different from what was observed during the experimental evaluation by having spikes. In the experiment, it was observed that as the observing duration increased, the delay spikes were more suppressed by the FIR delay filter.

Another point that is also valuable to study is the IIR delay filter effect. Figure 5.3 shows a comparison between the raw delays and filtered ones in simulations. Even the filtered delays exhibit spike effects. However, in the experiments, after the IIR filter ran for a while, spikes were suppressed, only those encountered at the beginning will appear after being filtered.

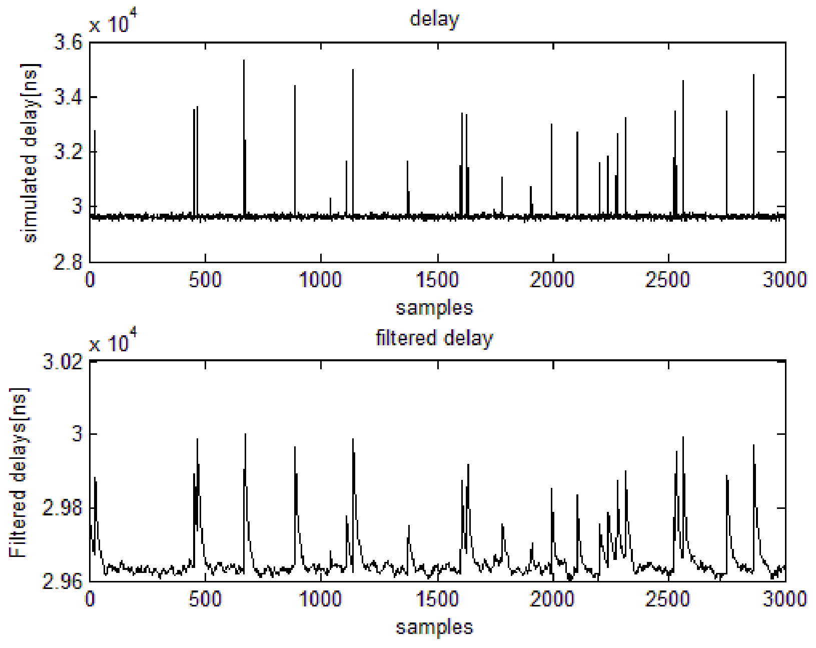


Figure 5.3. *The Simulated delay (upper) and filtered delay (lower).*

5.2. Ideal case

To test how the PI controller parameters affect the system, the system was simulated without any delays. And the system frequency is exactly 200 MHz. Given these conditions, in terms of time offset, the mathematical expression of the system is

$$\begin{cases} d[n+1] = d[n] + \frac{A}{p} - \frac{1}{p} \cdot a[n] \\ a[n] = k_p \cdot d[n] + k_i \sum_{i=0}^{n} d[i] \end{cases} \tag{5.2}$$

where $d[n]$ is the current time offset in ns, p is the *Sync* rate in pkt/s, $a[n]$ is the output of the PI controller, and k_p, k_i are the proportional and integral parameters of the PI controller. Rewriting (5.2) into one equation gives

$$d[n+1] = d[n] + \frac{A}{p} - \frac{k_p}{p} \cdot d[n] - \frac{k_i}{p} \sum_{i=0}^{n} d[i] \tag{5.3}$$

The constant in (5.3) decides the oscillation phase and amplitude. It also makes the system oscillate when the initial condition $d[0]$ is 0. The coefficients of $d[n]$ affect both the envelope and the period. And the initial condition $d[0]$ only affects the oscillation initial phase. The Sync rate expressed by p affects the oscillation by changing the coefficients of $d[n]$. Based on these, if the constant is not considered, and use

$$a_p = \frac{k_p}{p}, \qquad a_i = \frac{k_i}{p}$$

equation (5.3) becomes

$$d[n+1] = d[n] - a_p \cdot d[n] - a_i \sum_{i=0}^{n} d[i] \tag{5.4}$$

As can be seen from (5.4), in terms of time offset, the PLL system could be simplified into a system that only contains proportional and integral blocks.

With the help of Matlab, the above equations are drawn in form of 3D images with different a_p and a_i. The parameters are configured as such: $d[0]$ is 0, the constant is 10 and the loop size (n) is 500. The ranges of a_p and a_i are identical, i.e., 0.01 to 1 in steps of 0.01. Figure 5.4 shows the number of harmonic peaks, the fewer peaks the bigger damping ratio.

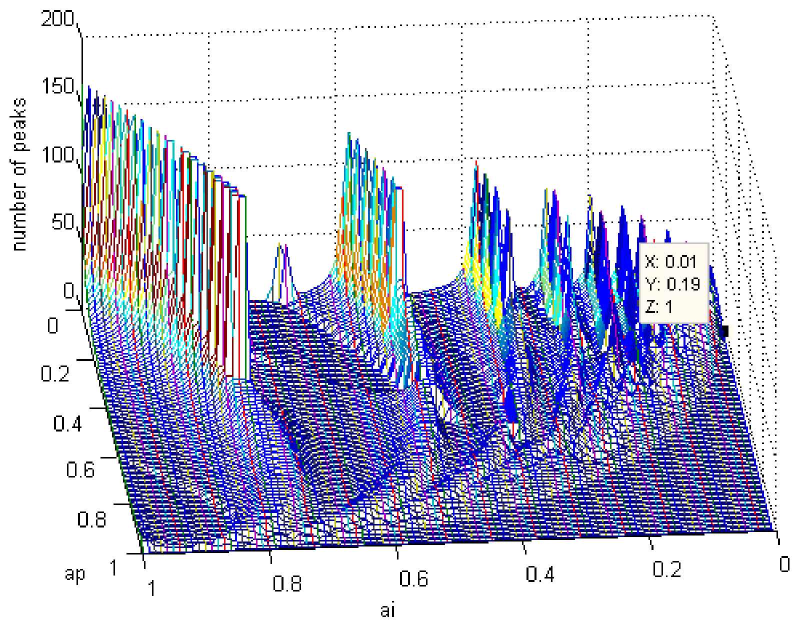


Figure 5.4. *The number of harmonic motion peaks. ai and ap are in the range of 0.01 to 1, in 0.01 increments.*

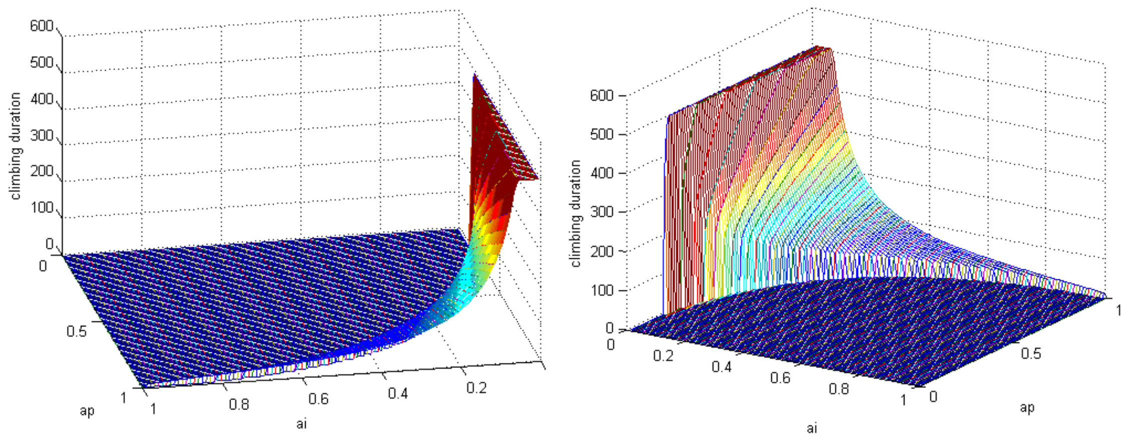


Figure 5.5. *The climbing durations from two different views.*

When the number of peaks is 1, it is either over damped or critical damped. Figure 5.5 shows the climbing duration with the same simulation parameters. The more samples it takes to reach the stable condition, the bigger damping ratio it has. And in the simulation, the under damped condition is described as 0 climbing duration. Comparing Figure 5.5 and the left graph in Figure 5.6, the parameters that make system under damped are identical, i.e., they are located within a quadrant (approximately).

We randomly choose the a_i and a_p pairs that are outside the quadrant (either overdamped or critical damped), and run the polluted simulation program. The result is that the clock does not behave underdamped any more, and when the slave to master PDV is heavier, the synchronization performance is improved. However, if the large PDV resides in the master to slave direction, the PI controller sharps the time offset peaks. This is shown in Figure 5.7 (the parameters a_i and a_p are 0.01 and 0.19). Besides, by examining simulations with bigger climbing period a_i and a_p pairs, we observed that a higher damping ratio is accompanied with a larger frequency variation. As shown in -Figure 5.7, for the case that the PDV in the slave to master is heavier, 0.4 a_p has bigger damping ratio, but larger frequency and time variations. The larger *Sync* rate p improves the frequency and time stability, with a prerequisite that the PI controller parameters are fixed. However, this is more likely because the larger *Sync* rate decreases the coefficients of the time offset as in (5.3) and thereafter suppresses the time offset variations. Moreover, in some cases (depending on the PI controller parameters), a certain *Sync* rate changes the harmonic conditions, e.g., from over damped to under damped.

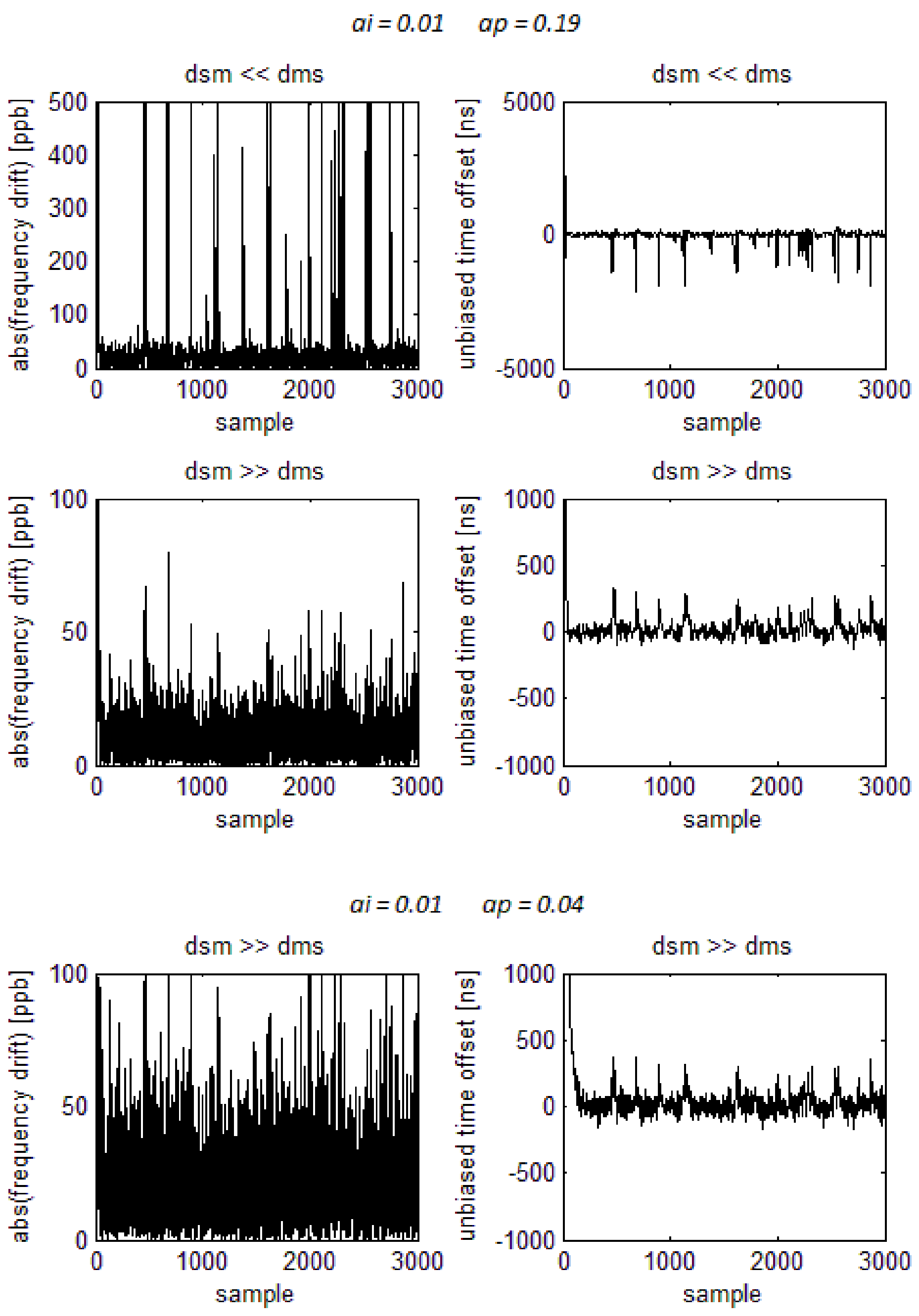


Figure 5.6. *The absolute value of frequency drifts for two scenarios: large PDV resides in the slave to master direction (top) and in the master to slave direction (bottom).*

6. RESULTS AND DISCUSSIONS

This chapter will show how the issues discussed in former chapters are combined and the corresponding results and analysis. Section 6.1 and 6.2 introduce the experimental evaluations of the FLL and PLL methods respectively. The evaluation for the PLL was conducted in two different ways, i.e., using a frequency counter or an oscilloscope individually.

6.1. FLL

Figure 6.1 shows these procedures in a flow chart. *r* is short for ratio, and *d* is the offset difference in duration τ. The right side of the flow chart gives a zoom out of the used filters.

First, the clock data set is initialized in the *Init* phase. Some information is assigned manually by the SW user, such as the message exchanging rate. This will be negotiated between the slave and master through *Signaling* messages. Some other configuration can be read from files, e.g., the last sequence number sent by the slave clock.

Secondly, a tuple contains four timestamps, the flag and sequence ID, which serves as the basis to indicator whether a newly received PTP message (*Sync* and *Delay_Resp* messages) is fresh enough to use. This step ensures that the timestamps are valid and reliable. PTP event packets heavily delayed are discarded.

Thirdly, the derived offset and delay are inputs of filters. FLL was also implemented in NTP, so filters in NTP, in terms of a single master clock, are used. Step controllers filter out the offset that exceeds a predefined threshold, unless this aberrant situation lasts more than 15 minutes. The popcorn suppressor removes the suspicious offsets if they exceed three times of a weighted average of the last *n* samples (such as 10). The *clock filter* algorithm selects the newest tuple whose delay is also the smallest. If the least delayed tuple is not newer than the last one passed through filters, the process returns to the network handler.

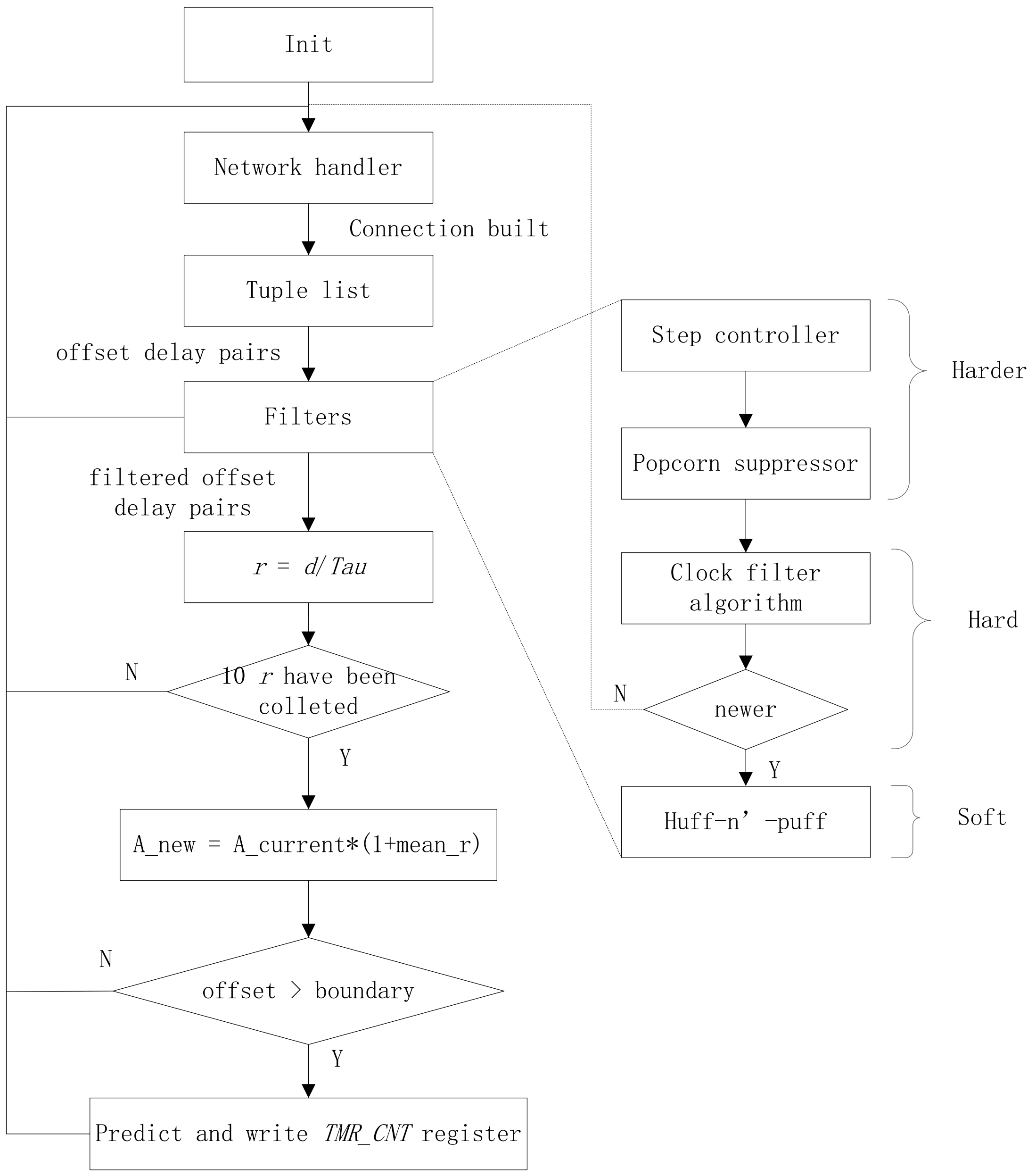


Figure 6.1. *Flow chart of an FLL based slave SW.*

The frequency error is calculated by dividing the offset difference and the corresponding duration. For example, if the two consecutive slave-to-master time offsets are 100 ns and -200 ns, and the duration is 10^9 ns, the oscillator frequency drift is

$$\frac{300}{10 \times 10^9} \times 10^9 = 30 \text{ ppb}$$

Recall that the frequency could be changed by *TMR_ADD* register, so the new factor is (1+30 ppb).

The FLL only fixes the clock frequency drift, it does not directly affect the phase (e.g., if the clock is ahead of the master clock by several hours). The solution is to set an upper limit. If the time offset steps over the limit, the value in *TMR_CNT_H/L* register would be directly changed. In practice, it is noticed that the adjustment of frequency drift gains a better stability if the average drift is used. Another point is that in NTP the period to tune clock frequency could be as long as 4.5 hours. It is observed in this thesis that, when the oscillator ran freely, the frequency performed a certain trend in duration *T*, and then wandered to another direction in the next *T'* duration, like a 'V' shape. This means that the tuning period should not be too long, or the FLL will be too slow to compensate the frequency drift. This is consistent with what has been explained in NTP-related materials, where the Allan Deviation (ADEV) was used as a parameter of FLL. ADEV indicates whether the clock frequency drift, other than the large PDV, is the main cause for the frequency drift. ADEV is a constant defined in NTP and in the Linux kernel, although the value is changed in newer kernel versions. A step-by-step guide on how to calculate ADEV is given in [22, p. 20].

If the *Sync* message rate is 1 pkt/s and the *clock filter* algorithm has length 10, the addend value would be adjusted based on its currently stored value, every 50 seconds.

6.1.1. Experimental evaluation

The devices used in the test are:

- Grandmaster clock – OSCILLOQUARTZ 5331 (includes GPS accessories)
- CISCO 100M switch.
- Slave clock – a computer with processor P2020.

The grandmaster and the slave are directly connected by a switch, which is part of a LAN including around 100 devices. Figure 6.2 shows the network connections.

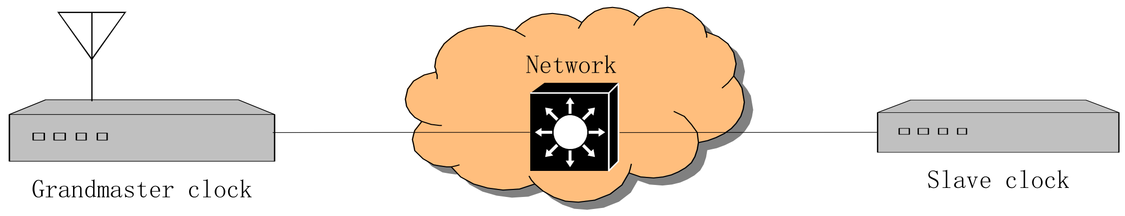


Figure 6.2. *Network connection used in the test of FLL method.*

The slave-to-master time offsets were collected from the output of the tuple list, so they are what the slave believes it has. It does not reflect the asymmetric path delay if there is any. The *Sync* message rate is 2 pkt/s and the whole test lasted about 2 hours. The testing results are drawn in Figure 6.3, in which the corresponding packet delays are presented as references (i.e., the ‘offset delay pairs’ in Figure 6.1). The figure shows that the offset is heavily affected by the delay, i.e., a peak on the delay is accompanied with a peak on the offset in the opposite direction.

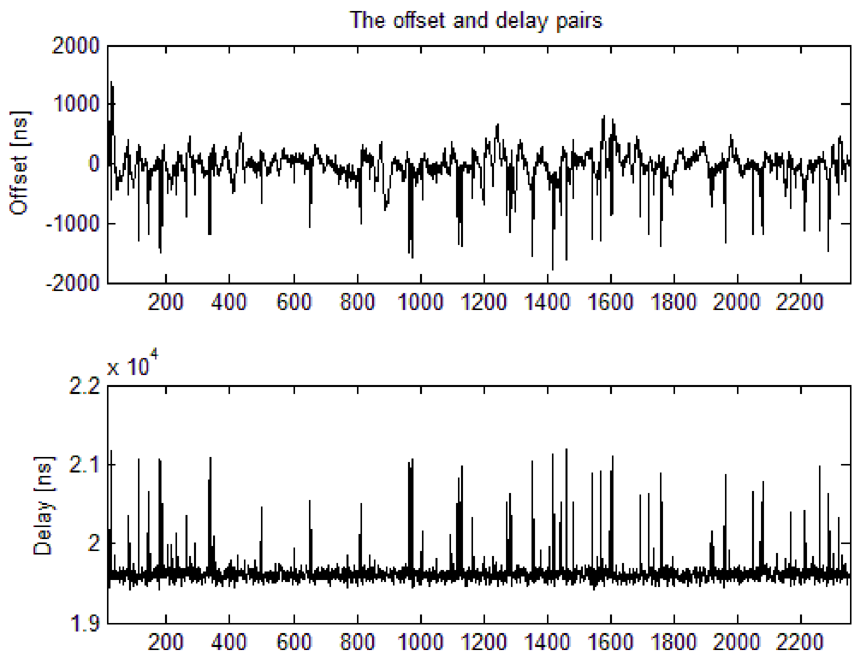


Figure 6.3. *Offset and delay observations in a two-hour test using the FLL method.*

Figure 6.4 shows the observed frequency drift, which is equal to the relative offsets divided by the observing duration. The obvious symmetry represents that the adjustment of oscillator frequency would cause a predictable adjustment in the near future, but in the other direction.

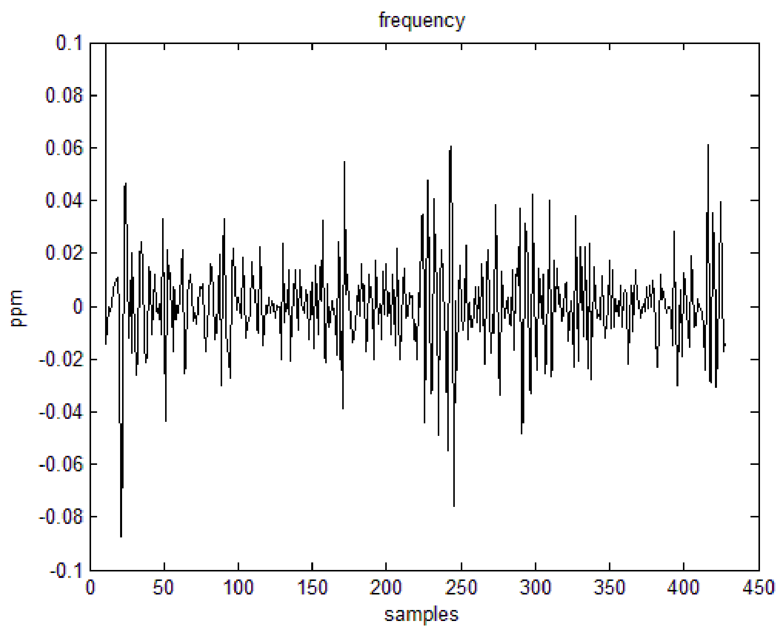


Figure 6.4. *Observed frequency drift – the difference of offsets divided by the time interval.*

Figure 6.5 shows values in the *TMR_ADD* register. The samples form a down slope but still can be predicted. When the oscillator frequency increases (mainly due to the environment), the addend value decreases to compensate for the frequency drift.

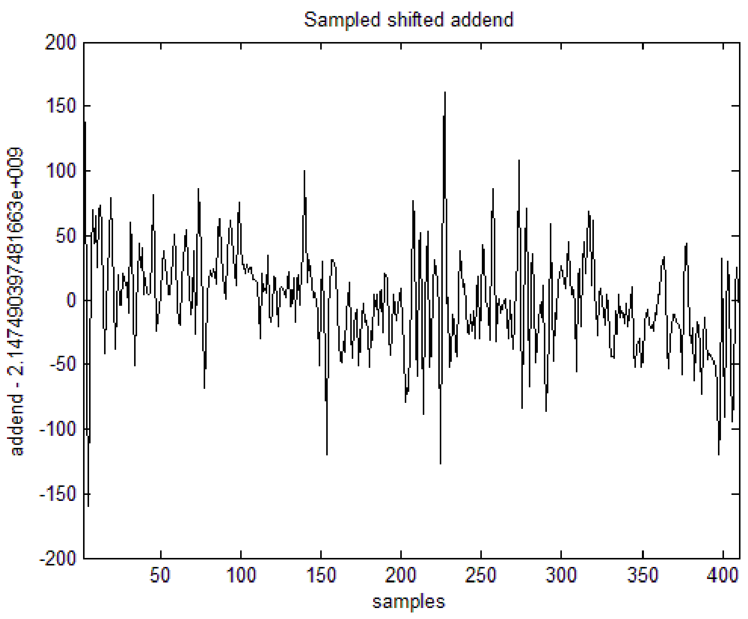


Figure 6.5. *Observed values in the TMR_ADD register.*

The number of collected addend (and frequency) values is 409, which is almost one fifth of the collected offset values (i.e., 2351).

As mentioned before, in NTP, the Huff-n'-puff filter is claimed to have a prerequisite that the delay and offset have wedge edges. However, as seen in Figure 6.6, the relative offset (difference between two consecutive offsets) has a certain relation with the relative delay (difference between two successive delays), while the absolute offset and delay do not present such a relation. In one test, the correlation between relative delay and relative offset is -0.867 while it is -0.596 between absolute delay and offset.

In Figure 6.6, the relative delay and offset exhibits a negative correlation, which satisfies their inverse relation in equation (2.4).

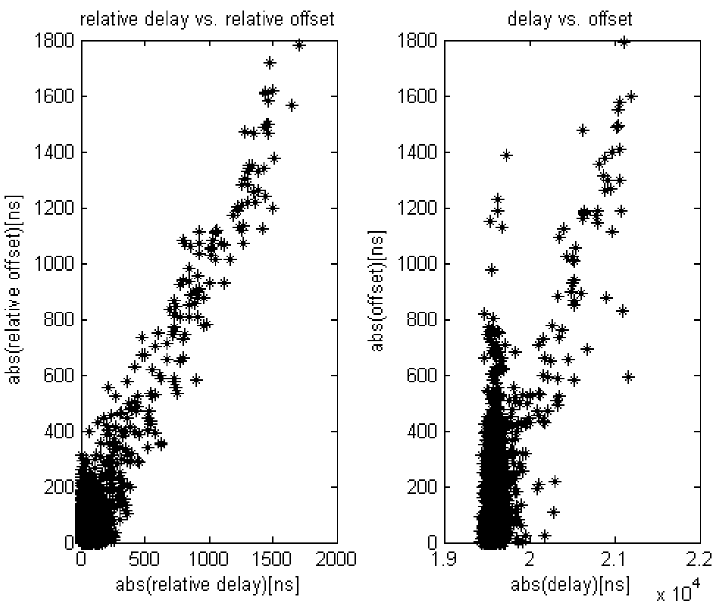


Figure 6.6. *Comparison of the relations between relative/absolute delay and offset.*

6.2. PLL

The PLL method resembles the FLL method in the first three steps.

In PLL, the *clock filter* algorithm is replaced by an IIR delay filter and a FIR offset filter in PTPd. K_I and K_P in the PI controller are equal to 0.01 and 0.05 respectively. Tests with some other values did not show an obvious difference. Because it takes about 10 adjustments for the controller to go back to its normal status for every large input fluctuation, the input of the PI controller is clamped at a threshold *adj_max*. Compared with the FLL method, the tuning target of PLL is the initial addend value.

The frequency is not calculated in the PLL method, so the time interval between two sampled offsets is not involved in obtaining the addend adjustment. The effect, if any, that laying on the PI controller deserves a future study.

Other than this, the experiments show that the PLL is better than the FLL in terms of oscillator stability. Figure 6.7 gives a flow chart of the PLL implementation.

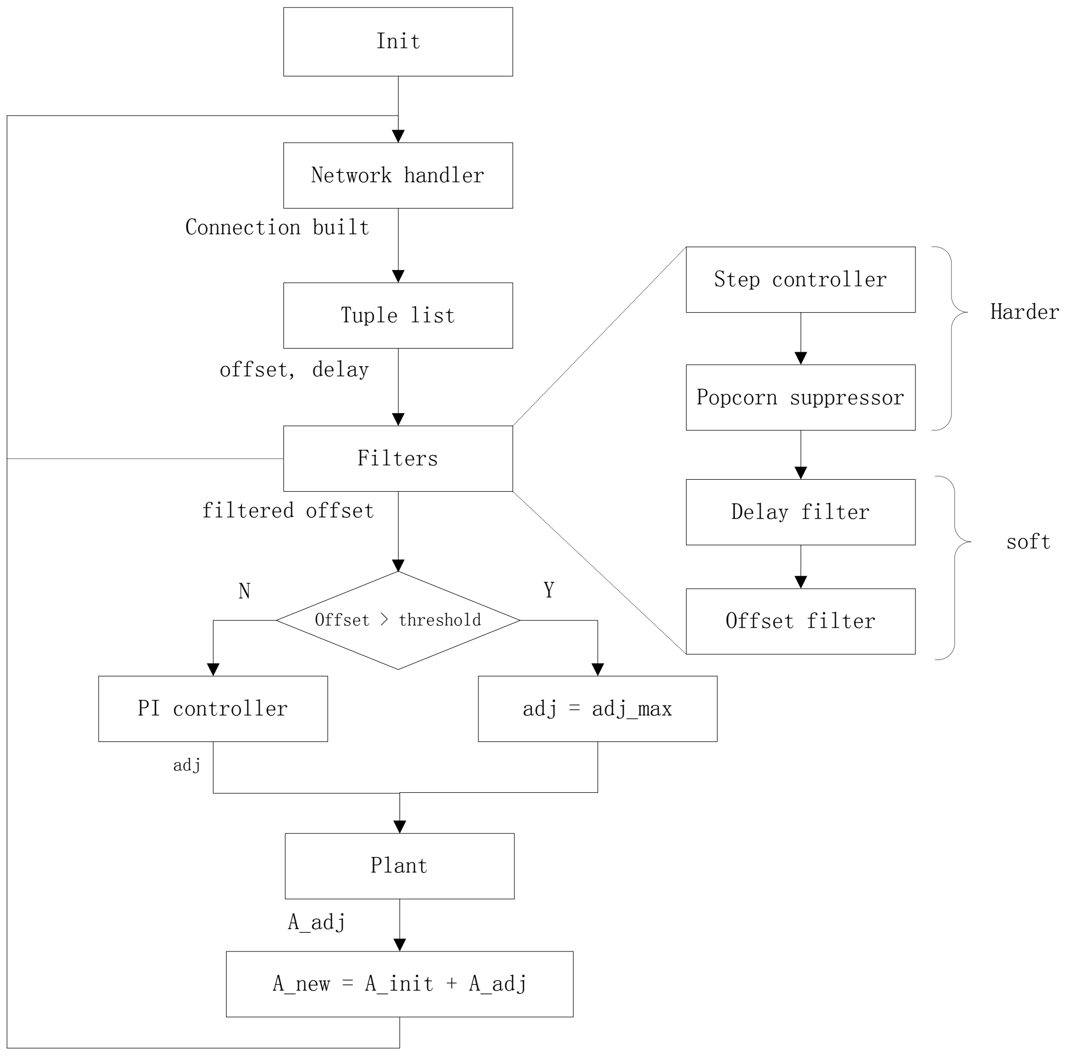


Figure 6.7. *Flow chart of the slave using PLL.*

6.2.1. Experimental evaluation

The connection is shown in Figure 6.8.

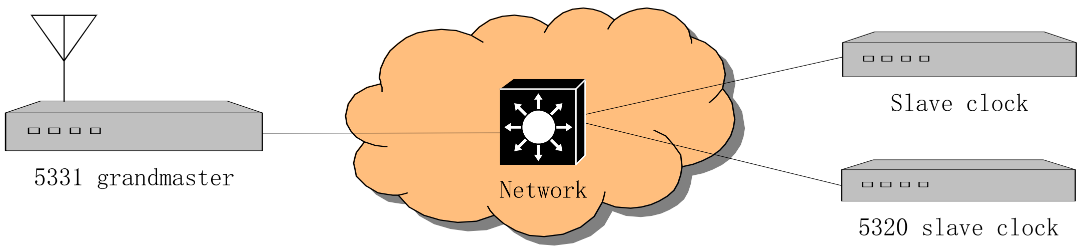


Figure 6.8. *The network set up for the PLL method.*

The devices used in the test are:

- Grandmaster clock – OSCILLOQUARTZ 5331 (includes GPS accessories).
- CISCO 100M switch – big capacity
- switch2 100M – small capacity
- Slave clock – computer with processor P2020.
- Slave clock – OSCILLOQUARTZ 5320.
- TEKTRONIX oscilloscope.
- Frequency counter.

To test the PLL method, not only the offsets and delay from the slave were collected, but also the phases of the PPS signals were compared. Both the grandmaster and slave clock can output phase aligned PPS signals. The phase of grandmaster 5331 is aligned with the UTC time carried in the GPS message. The 5320 slave clock helps to monitor the network. If both 5320 and the designed PTP slave exhibit a large offset fluctuation simultaneously, the PDV could be recognized as the reason.

6.2.1.1 Frequency counter

To test the slave PPS against the grandmaster PPS, a frequency counter was used. Although it does not give any phase offset information, the frequency counter senses the input signal frequency with a very high precision. The experiments used the internal oscillator of the frequency counter as the frequency generator. The other choices would be using the standard 10 MHz signal or the PPS signal from the grandmaster as the frequency counter reference. Because the frequency counter was calibrated recently, they did not show a difference during tests. The set up for 3 methods is shown in Figure 6.9.

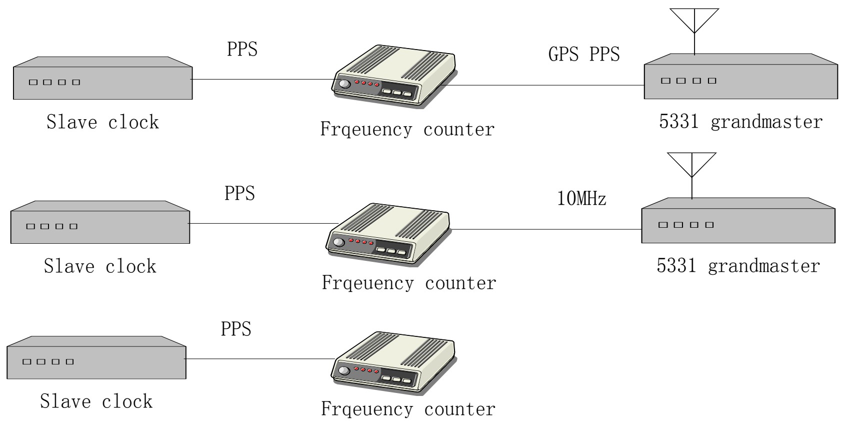


Figure 6.9. *GPS PPS reference signal (upmost), 10 MHz calibration signal (middle), internal oscillator freely running (lowest).*

Because the width of the slave PPS is 20 ns (2 times *TMR_CTRL[TCLK_PERIOD]*), the trigger mode of frequency counter was turned on and the sensing sensitivity was set to a proper threshold. Thereafter, the figure displayed on the front panel was not affected by the wide band noise.

Because the frequency counter was too old to support modern storage media, in order to evaluate the slave PPS frequency, we randomly observed the figure on the screen. The longest period was 1 s and 23 ns (-23 ppb), while the shortest was 25 ns to 1 s (+25 ppb).

Using a frequency counter has its own shortages. First, due to the lack of the storage media, the results have to be logged by random observations, i.e., the sample space is not intact enough to be convincing. Secondly, the result does not reflect the effect of accidental network spikes on the PPS. Thirdly, only frequency accuracy is reflected in the results, but not the accumulative frequency error – phase error.

6.2.1.2 Oscilloscope

With the help of an oscilloscope, the PPS phase offsets can be easily observed on the graph. The slave clock PPS is compared with the GPS PPS. Additionally, in order to figure out the reason of slave PPS offsets, the PPS of 5320 slave clock is used as a reference. If both slaves present drifts, the network PDV or GPS PPS might be the reason. If only one slave shows drift, the PDV in its link or the clock tuning algorithm could be the reason. Figure 6.10 shows the set up.

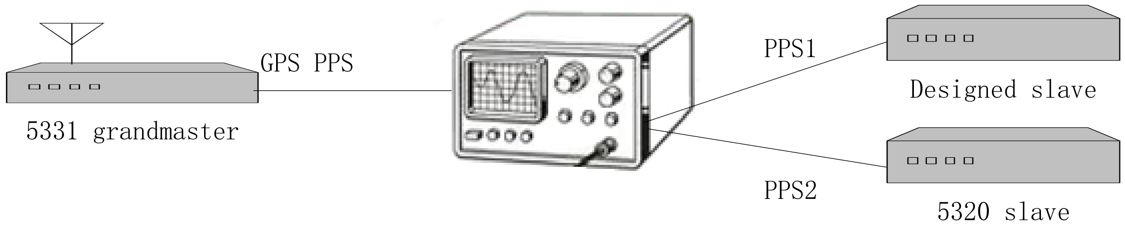


Figure 6.10 *Connections between oscilloscope and clocks.*

The slave PPS signal is shown in Figure 6.11, with a pulse width of 20 ns and a voltage of 2 V. The distortion of the PPS signal is caused by the signal reflection introduced by the test cable. In an earlier test, there were 2 test lines (simultaneous frequency counter and oscilloscope methods), the distortion was larger and showed up at symmetrical places. The distortion may cause ambiguities to the following electronic components, but is believed that once all the probes are removed, the PPS signal would not have such a reflection effect.

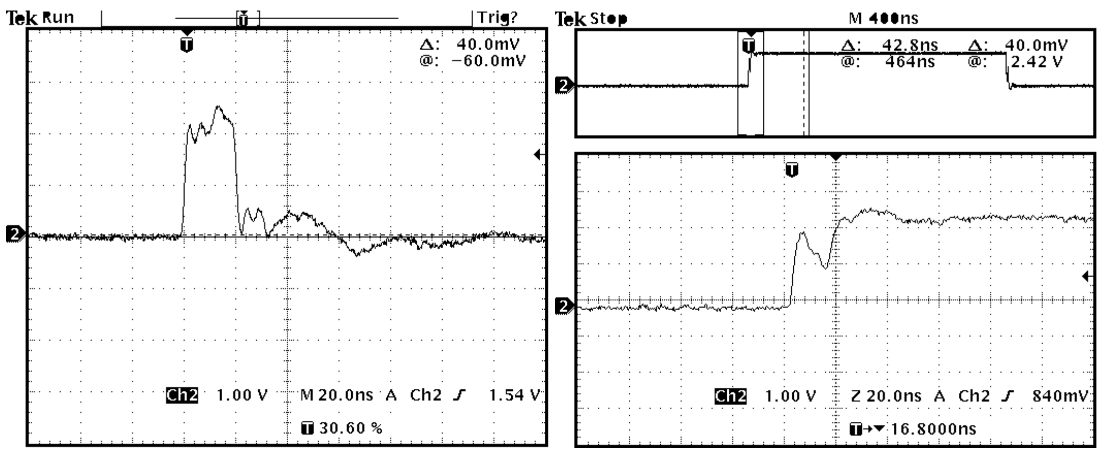


Figure 6.11. *The slave PPS signal with one test line (left) and two test lines (right).*

The width of the GPS PPS from the grand master 5331 is 200 ns, so the trigger mode is turned on. In Figure 6.12, GPS PPS is shown in purple and it also served as the trigger source. The blue and green pulses stand for the designed slave and the 5320 slave, respectively. From this figure one can see that the phase difference between slave and grandmaster is about 700 ns (the oscilloscope scale is 200 ns per division) and about 100 ns between slave 5320 and grandmaster. The ripple distortion in the slave PPS is caused by the test line. The large offset between the GPS PPS and the designed slave PPS was because the *TMR_ALARM1* register was not properly configured as described in the manual.

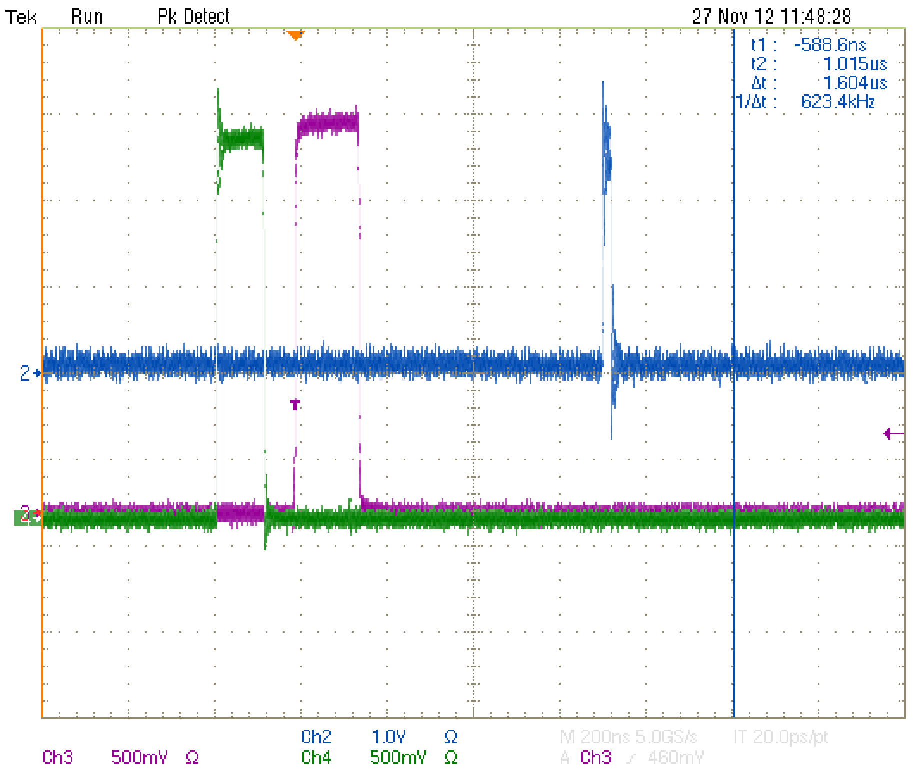


Figure 6.12. *5331 GPS PPS (purple), slave PPS (blue), 5320 slave PPS (green).*

In accordance with Figure 6.7, addend, offset, filtered offset, delay and filtered delay were collected. The analysis was conducted with the following configurations:

- *Sync* rate: 0.5 per second
- Sample size: 9797 tuples.
- Test time: about 5pm to 8am (day+1).

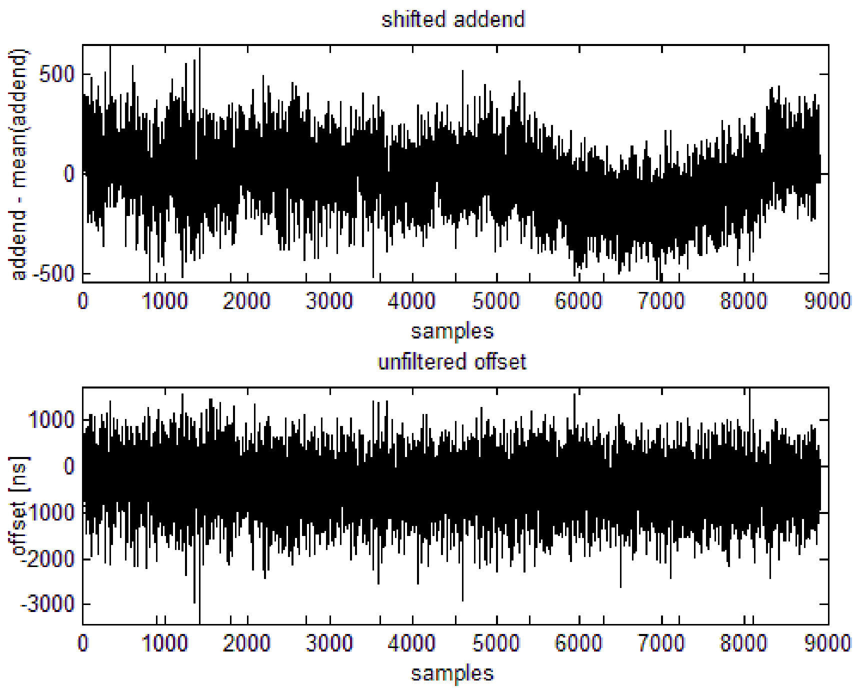


Figure 6.13. *Observed mean addend (top) and unfiltered offset (bottom) over an observation interval of about 1.5 hours.*

Figure 6.13 shows the shifted addend and unfiltered offsets. In order to keep the offset close to 0 ns, the adjusted addend exhibits a shape of "V" from time to time, the most obvious one is located between the tuple 5000 to 9000. Because the clock actually is the system clock divided by the addend, the "V" of addend means "A" of the system clock frequency.

Figure 6.14 shows the relation between the delay and offset. The same as in the case of FLL, differences between two consecutive samples are closely related, but a lot less on the absolute values. Two important phenomena differ from those in FLL. Firstly, the correlation is 0.477 (relative) and 0.347 (absolute) – both positive. Secondly, in the delay-offset figure, a second "area" shows up. The reason for this second circle has not been identified yet. After checking Figure 6.15, the delays do not exhibit a second level. Besides, there was no obvious interference, such as a connection failure or HW faults, introduced to the system between samples 100 to 9000. And that excludes the possible existence of a second steady-state establishing process, where the system tries to recover from a huge interference.

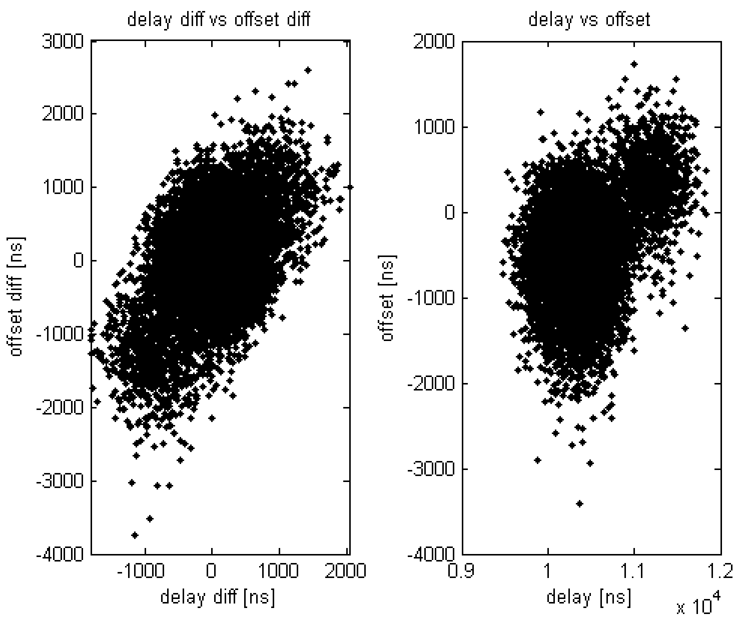


Figure 6.14. *Comparison of the relations between relative/absolute delay and offset.*

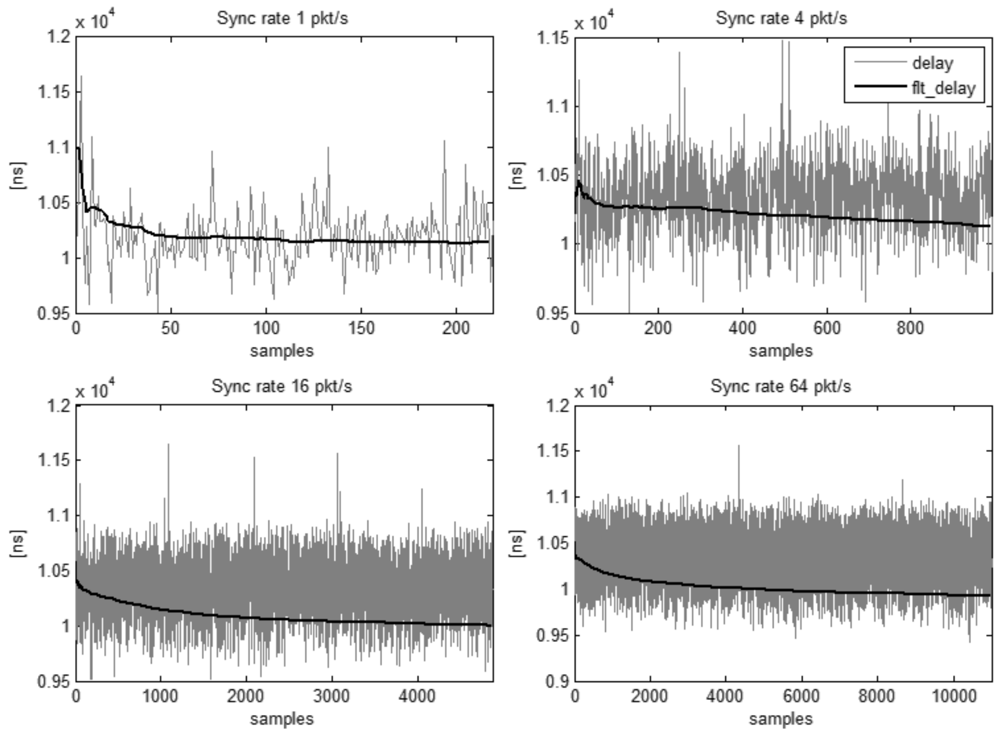


Figure 6.15. *Delay and filtered delay for four different Sync rates.*

The biggest challenge in terms of slave clock design is how to get the reliable time offset as the system input. Because the time offsets are the adjustment basis, they should be trustable to reflect only the oscillator information, instead of the network PDV. However, both Figure 6.6 and 6.14 show that the variation of the offset is more or less affected by that of the delay. And this is harmful to the system, especially in the case of FLL. If PDV cannot be removed from the offset, PLL would be a better choice. It can be seen from Figure 6.15 that the PDV in PLL is already dramatically suppressed. The correlation between the relative filtered-offset and delay is -0.441.

Figure 6.16 shows that the clock is adjusted more than its actual frequency drifts. In the next several periods, it is tuned to an opposite direction. For example, if currently the clock is tuned downwards, resulting in a negative spike in addend, the addend would exhibit a positive spike at the next adjustment. But how much is it over-adjusted? If *A* is the addend value, its relation with system clock frequency f_s satisfies

$$f_s \cdot \frac{A}{2^{32}} \cdot 10 = 10^9 \tag{6.4}$$

Assume x difference on A would lead to 1 ns difference:

$$f_s \cdot \frac{x}{2^{32}} \cdot 10 = 1 \tag{6.5}$$

$$\frac{x}{A} = \frac{1}{10^9} \tag{6.6}$$

If the system frequency is 200 MHz, addend value is 2^{31}, every 2.15 change made to addend would cause the clock frequency to wander by 1 ns. Since the system clock frequency varies, the actual figure depends on the instantaneous system clock frequency. In Figure 6.16, the maximum addend variation is 545; correspondingly, the maximum frequency variation is 253 ppb (545 divided by 2.15)

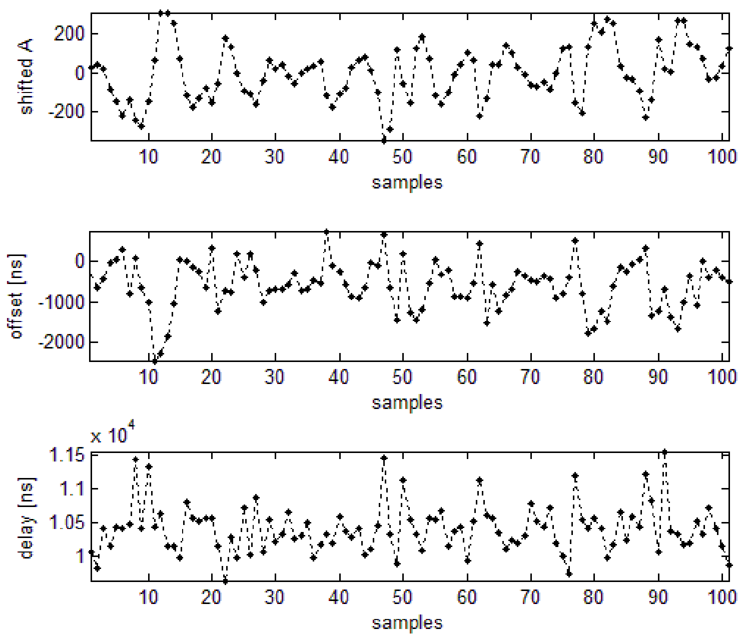


Figure 6.16. *The observed shifted addend, offsets and delays.*

The effect of PI controller on offset/addend is shown in Figure 6.17. Although the initial phase was not controlled to be the same during the test, it takes almost the same time for the system to reach a steady state regardless of the *Sync* rate (about 50, 120, 450, 1700 and 4000 samples for the *Sync* rate of 0.5, 1, 4, 16 and 64 pkt/s respectively). This is because the system bandwidth is remained the same when the sampling rate changes.

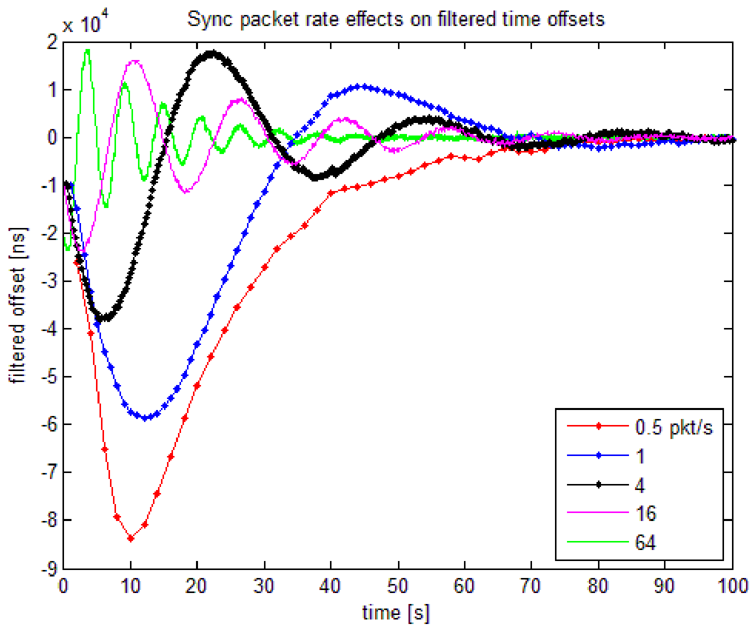


Figure 6.17. *Offset start-up for 5 Sync rates.*

In another test, the performance of different *Sync* message rates is studied. The statistical features are given by the oscilloscope. Because of the unstable connection between the pin on the slave and the oscilloscope probe, it was difficult to conduct the test. The slave PPS failed to be sensed by the oscilloscope from time to time and this brought tremendous impact on all the statistical features (variance, average and minimum). Due to this reason, the oscillator observation period was not more than 10 minutes each time, before features were recorded.

The results are shown in Table 6.1. Higher rate of the *Sync* messages usually brings a smaller variance, but larger mean phase error. Regardless of the network type, when the *Sync* message rate is 0.5, the slave encounters the largest variance. The minimum *Sync* rate is 0.25 allowed by the grandmaster.

Table 6.1 *Sync* message rate effect.

Sync packet rate [pkt/s]	average offset	Min. offset	Max. offset	Standard deviation [ns]
Cisco switch, LAN with about 100 hosts.				
64	3.645426 µs	3.498 µs	3.841 µs	68.19
16	3.5703562 µs	3.48 µs	3.686 µs	57.62
4	3.4023047 µs	3.207 µs	3.602 µs	124.0

1	3.2802392 µs	3.197 µs	3.516 µs	59.24
0.5	3.3465021 µs	3.015 µs	3.7 µs	232.5
Switch 2, LAN with 3 hosts.				
4	729.09863 ns	689.8 ns	782.6 ns	21.27
1	719.42054 ns	658.2 ns	781.5 ns	25.73
0.5	742.42115 ns	651.0 ns	822.2 ns	49.59

The slave is adjusted every time a valid tuple is received and not discarded by filters. When the achieved adjustment is much more fluctuated than the oscillator frequency drift, the PDV introduces bigger error in the slave calculated offset. When the *Sync* rate is 64 pkt/s, the addend would be updated 64 times in one second. This is beyond the short-period frequency drift due to the temperature fluctuation (not to mention the oscillator aging speed). The clock is more likely to be tuned based on the PDV other than its own frequency drift. An optimized slave SW should have the ability to evaluate the quality of its local oscillator (frequency wander) out of the level of path noise. Figure 6.18 shows Table 6.1 in graph, it is easy to recognize that 1 *Sync* message per second is the optimum.

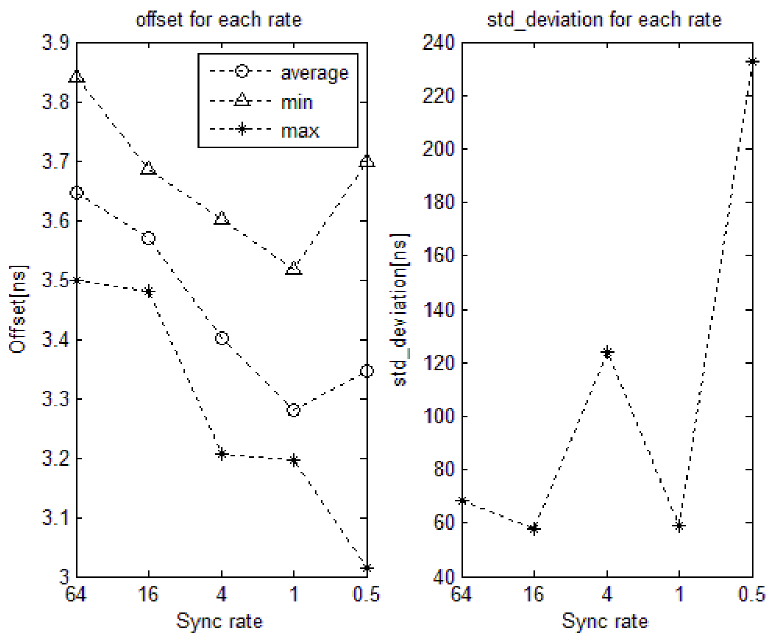


Figure 6.18. *Plot of the results of the first scenario in Table 5.2.*

7. CONCLUSIONS AND FUTURE WORK

7.1. Conclusions

Based on the tuning information source, there are two methods to adjust the clock – the phase and the frequency. They both share a feedback control structure, except that FLL method also takes the time interval into account. In general, FLL is a good idea if the clock frequency drift is dominant, instead of the PDV. And since it directly changes the clock frequency, and indirectly changes the phase (recall that phase is an accumulation of frequency), it needs to step the clock when the phase offset is too big.

Stability of an oscillator mainly refers to its frequency accuracy. The FLL method exhibits a better stability when the frequency information is gained by averaging several (such as 10) frequency samples. Too many samples would fail to characterize the oscillator, while too few samples would invoke huge fluctuation. The FLL method uses the NTP loop filter. The legendary Huff-n'-puff filter seems not to be helpful on reducing the asymmetric path delay. This is probably because the network environment is different from what it was designed for. Huff-n'-puff usage was claimed to be based on the positive correlation between the delay and offset. In fact, the positive correlation was observed between their deviations.

The PLL method is better than FLL in both the frequency accuracy (stability) and the phase accuracy. PLL method only measures the phase offset. Both delay and offset are handled not only by step controller and popcorn spike suppressor that are introduced in NTP, but also by IIR filter (on delay) and FIR filter (on offset) that are introduced in PTPd. In the PLL method, because the *TMR_ALARM1* register was not properly configured as described in the chip manual, a big offset between GPS PPS phase and designed slave PPS signal appeared. In a better implementation, the ideas of [14, p 787] should be followed to generate a phase aligned PPS in P2020. Other than this, three conclusions can be made:

- The absolute delay causes the phase offset, while the PDV causes frequency error.
- PPS variations of the designed slave and OSCILLOQUARTZ 5320 slave appeared at the same time on the oscilloscope screen, meaning that they were caused by large PDV and/or unstable GPS signal (unlikely).
- The predictable variation of "offset" and "addend" shows up from time to time.

The 3rd item was also observed on the oscilloscope screen. The slave PPS stepped towards the grandmaster PPS for continuous *N* triggers, and then stepped away from grandmaster PPS for next continuous *M* triggers. During this process, one can almost predict where it is walking to. Moreover, the relative offset and relative delay show an obvious relation. That means, the offset which should only reflect the clock bias, is polluted by the PDV. The effect of different *Sync* rate was also analyzed at the end of this section. The conclusion is that, higher *Sync* message exchange rate is better to describe the network link condition. But adjusting the local oscillator more frequently than it is affected by the environment or aging, would lead to a poor result. Therefore it should be avoided.

No matter which method is used, the adjusting target is the frequency of clock through the addend value that is stored in *TMR_ADD* register. By writing to the *TMR_CNT_H/L*, the clock can also be directly stepped. Stepping the clock is equal to changing the phase of the clock. And this only happens when the phase deviation is too big in both methods.

7.2. Future work

The effect of missing event packets deserves a future study. As can be seen from the PLL method, the interval between obtained offsets is not involved in tuning the clock. However, it is claimed in [18] that, when *SYNC* messages are missing, the PLL method is worse than the FLL method. The implementation of [18] is based on the Kalman filter. Because the methods are different, the effect of losing *SYNC* should be studied on the PLL of this thesis, with the help of traffic generator and network simulator.

The phase alignment in P2020 was not configured properly as guided in the chip manual. The operation in this thesis was enabling the HW timing, followed by writing the *TMR_ALARM1* register. According to the reference manual, this should be the other way round.

The recognition of the PTP event packets should remain in the network driver, while register operations should be put into an independent clock driver. In this thesis implementation, the registers are visited in the PTP slave SW.

The OSCILLOQUARTZ 5320 has the ability of estimating the path delay on two directions separately. There are a dozen of papers which have studied this approach. For example, [31] describes the use of the statistical average offset (collected in a group) to recover the path delay in each direction.

Moreover, there are many other methods and different filters (such as the Kalman filter mentioned above) are claimed to be helpful for enhancing the accuracy. Some of them are not compatible with the IEEE 1588 recommendation. Implementations of PTP need not only vertically deeper, but also horizontally wider studies.

References

[1] "Time and frequency division," NIST. [Online]. Available: http://tf.nist.gov/general/enc-no.htm#ocxo

[2] "Wireless backhaul synchronization," Ceragon Networks Ltd. [Online]. Available: http://www.ceragon.com/files/Ceragon%20-%20Synchronization%20-%20Technical%20Brief.pdf

[3] "Deployment of precision time protocol for synchronization of GSM and UMTS base stations," Symmetricom Inc, 2008. [Online]. Available: http://www.symmetricom.com/resources/downloads/white-papers/IEEE-1588-PTP-Solutions/Deployment-of-Precision-Time-Protocol-for-Synchronization-of-GSM-and-UMTS-Base-Stations/

[4] "Popcorn and step control (clock discipline algorithm) (computer network time synchronization)," 2008. [Online]. Available: http://what-when-how.com/computer-network-time-synchronization/popcorn-and-step-control-clock-discipline-algorithm-computer-network-time-synchronization/

[5] "Asymmetric networks (xPON etc.) – timing solutions," Calix Inc., 2010. [Online]. Available: http://www.chronos.co.uk/files/pdfs/itsf/2010/Day3/02-Asymmetric_Network_Timing_Solutions.pdf

[6] "Support for IEEE 1588 protocol in PowerQUICC and QorIQ processors," Networking and Multimedia Group Freescale Semiconductor, Inc., Austin, TX, Sep 2010. [Online]. Available: www.freescale.com/files/32bit/doc/app_note/AN3423.pdf

[7] D. W. Allan, N. Ashby, and C. C. Hodge, *The Science of Timekeeping Appliction Note 1289*. Hewlett Packard, 1997.

[8] J. Bausch, "TCXO vs. OCXO," Aug 2011. [Online]. Available: http://www.electronicproducts.com/Passive_Components/Oscillators_Crystals_Saw_Filters/TCXO_vs_OCXO.aspx

[9] L. Cosart, "IEEE 1588 frequency and time transfer measurements and analysis: Clock, PDV, and load," in *Proc. of the 42nd Annual Precise Time and Time Interval (PTTI) Meeting*.

[10] J. C. Eidson, *Measurement, Control, and Communication Using IEEE 1588*. London, UK: Springer-Verlag, 2006.

[11] *Terrestrial trunked radio (TETRA); Voice plus data (V+D); Part 2: Air Interface (AI)*, ETSI Technical Specification 100 392-2 Version 3.3.1, 2008.

[12] *Universal Mobile Telecommunications System (UMTS); Base station (BS) radio transmission and reception (FDD)*, ETSI Technical Specification 125 104 Version 8.6.0, 2010.

[13] J.-L. Ferrant and S. Ruffini, "ITU-T q13/15 updates," Paris, FR, Mar 2012. [Online]. Available: http://www.ietf.org/proceedings/86/slides/slides-86-tictoc-8

[14] *P2020 QorIQ Integrated Processor Reference Manual*, Freescale, 2010.

[15] *Support for IEEE 1588 Protocol in PowerQUICC and QorIQ Processors*, Freescale Semiconductor, Inc., 2010.

[16] T. Frost, “IEEE-1588 standard version 2 tutorial,” Agilent Technologies Inc., 2006. [Online]. Available: http://www.webcitation.org/5qaJpYqCH

[17] ——, “Deployment considerations for IEEE1588 in telecommunication networks,” in *Proc. The 5th Intenational Telecoms Sync Form (ITSF)*, London, UK, Nov. 13–15, 2007, pp. 1–21.

[18] G. Giorgi and C. Narduzzi, “Robustness to SYNC packets loss in network synchronization,” in *Precision Clock Synchronization for Measurement Control and Communication (ISPCS, International IEEE Symposium on))*, Munich, Sep. 12–16, 2011, pp. 120–125.

[19] *Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, IEEE Std. 1588-2002, 2002.

[20] *Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, IEEE Std. 1588-2008, Rev. IEEE1588-2002, 2008.

[21] *Network Time Protocol Version 4: Protocol and Algorithms Specification*, IETF RFC 5905, 2010.

[22] *Design objectives for digital networks – definitions and terminology for synchronization networks*, ITU-T Recommendation G.810, 1996.

[23] J. Jaspernetite, K. Shehab, and K. Weber, “Enhancements to the time synchronization standard IEEE1588 for a system of cascaded bridges,” in *Factory Communication Systems. Proc. IEEE International Workshop on*, Sep. 22–24, 2004, pp. 239–244.

[24] T. Kovacshazy and B. Ferencz, “Performance evaluation of ptpd, a IEEE 1588 implementation, on the x86 linux platform for typical application scenarios,” in *Instrumentation and Measurement Technology Conference (I2MTC), 2012 IEEE International*, May 13–16, 2012, pp. 2548–2552.

[25] Q. Li and D. L. Mills, “Jitter based delay boundary prediction of wide area networks,” *Networking, IEEE/ACM Transactions on*, vol. 9, pp. 578–590, Oct. 2001.

[26] D. L. Mills, “Modelling and analysis of computer network clocks,” Electrical Engineering Department at University of Delaware, Newark, DE, Tech. Rep. 92-5-2, Sep. 1992.

[27] ——, “Adaptive hybrid clock discipline algorithm for the network time protocol 1,2,” *Networking, IEEE/ACM Transactions on*, vol. 2, pp. 505–514, Oct. 1998.

[28] ——, “A brief history of ntp time: confessions of an internet timekeeper,” *ACM Computer Communications Review*, vol. 33 (2), pp. 9–22, Apr. 2003.

[29] ——, *Computer Network Time Synchronization: The Network Time Protocol*, 1st ed. Boca Raton, FL: CRC Press, 2006.

[30] ——, "Network time protocol version 4 reference and implementation guide," Electrical Engineering Department at University of Delaware, Newark, DE, Tech. Rep. 06-06-1, Jun. 2006.

[31] T. Murakami, Y. Horiuchi, and K. Nishimura, "A packet filtering mechanism with a packet delay distribution estimation function for IEEE 1588 time synchronization in a congested network," *ISPCS 2011. IEEE International Symposium on*, pp. 114–119, Sep. 2011.

[32] F. L. Walls and J. R. Vig, "Fundamental limits on the frequency stabilities of crystal oscillators," *Ultrasonics, Ferroelectrics and Frequency Control, IEEE Transactions on*, vol. 42, pp. 576–589, Jul. 1995.

[33] R. Zarich, M. Hagen, and R. Bartos, "Transparent clocks vs. enterprise ethernet switches," in *ISPCS 2011. IEEE International Symposium on*, Munich, Sep. 2011, pp. 62–68.

[34] A. Zhou and X. Duan, "Requirements and viewpoints for backhaul synchronization," China Mobile Ltd., 2008. [Online]. Available: http://tools.ietf.org/agenda/71/slides/tictoc-6.ppt

APPENDIX 1: PLL METHOD SIMULATION

%% main.m:

```
% % delay(d) samples are colleted from the real network
% % t2 = t1 + delta + dMS
% % t4 = t3 - delta + dSM
% % d = 0.5 * (dMS + dSM)
% % all in nanoseconds
%
close all
clc
clear all
format long;

load '.\11.19\offset.txt'
% % for delay simulation
% load '.\11.19\delay.txt'

% State variable
DMSHEAVIER = 1;
DSMHEAVIER = 2;
EQUALDELAY = 3;

% %result analyze
%
% %
% @init stands for the whole system input, the system properties are
% examined by changing the value here. It is defined as:
%       [initial_offset, ai,     ap,    p,   state      loopcnt]
init = [30000          0.01    0.19    1    1    3000];
initial_offset = init(1);
ai = init(2);
ap = init(3);
p = init(4);
state = init(5);
loopcnt = init(6);

%% oscillator offers system frequency
% system frequency fs
global fs
% oscillator frequency error estimation
D_scale = 10 * rand(1);
D = D_scale * 10^-11;
% nominal system frequency
fs_nominal = 200000000;
% biased system frequency
fs = fs_nominal + D*fs_nominal;

%% delay simulation based on the live collected delays
%* 1) Basic statistical calculations are made to achieve the independ-
ent
%* two direction delays, and their sum satisfies the variance and ex-
pectation
%* of the live collected delays;
%* 2) Because the status variable for every simulation could be only
one,
%* in order to test all the three cases, they should use the same sim-
ulated
```

```
%* delays. Thereafter, using function dlmwrite() to store them. And
call
%* function load() to read the stored delays in every simulation.
%* Therefore, this block (i.e., delay simulation) runs only one time
by
%* uncomment.
% [a b] = max(delay);
% new_d = [delay(1:(b-1)); delay((b+1):end)];
% var_delay = var(new_d, 1); %var_delay is variance
% e_d = mean(new_d);
% % Assume dms and dsm are independent, (in real case impossible)
% % each satisfies Gausian distribution with expectation: e_dms or
e_dsm
% % variance var_dms or var_dsm, and standard deviation: c_dms or
c_dsm
% % although it would be naturaly asymmetric already in this way, but
there
% % can be one direction more traffic than the other, and this is con-
trolled
% % by r
% % dms = r*dsm
% % E[(dms+dsm)/2] = e_d
% % <=> (e_dms + e_dsm) / 2 = e_d          (1)
% %      e_dms = r*e_dsm                    (2)
% %  =>  e_dsm = 2*e_d / (r+1)   ,   e_dms = 2*e_d - e_dsm
% %
% % D[(dms+dsm)/2] = var_delay
% % <=>  var_ms + var_sm + 2*cov(dms,dsm)= 4*var_delay , assuming dms
and dsm
% % are independent, so cov(dms, dsm) = 0
% %      var_ms = r*r*var_sm
% %  =>  var_sm = 4*var_delay / (r*r+1), var_ms = 4*var_delay - var_sm
% r = 1;   % dms / dsm = r
% e_dsm = 2*e_d / (r+1);
% e_dms = 2*e_d - e_dsm;
% var_sm = 4*var_delay / (r*r+1);
% c_sm = sqrt(var_sm);
% var_ms = 4*var_delay - var_sm;
% c_ms = sqrt(var_ms);
% dms = e_dms + c_ms * randn(1,loopcnt);
% dsm = e_dsm + c_sm * randn(1,loopcnt);
%
% d = 0.5*dms + 0.5*dsm;
% % peak_on_delay is used to control the percentage of peaks that are
% % observed during the live test. Because of this operation, the var-
iance
% % the final simulated delays would be much bigger than that in the
live
% % test.
% peak_on_delay = rand(1,loopcnt);
% for i=1:length(d)
%     if peak_on_delay(i) > 0.992
%         d(i) = d(i) *(1+rand(1)/5);
%     end
% end
% figure, plot(d),title('simulation d')

% %% - 2013.5.12
% dlmwrite('simulateddms.txt', dms', 'delimiter', '\t', ...
%           'precision', 6)
% dlmwrite('simulateddsm.txt', dsm', 'delimiter', '\t', ...
%           'precision', 6)
% dlmwrite('simulatedD.txt', d', 'delimiter', '\t', ...
%         'precision', 6)
disp('2013.6.7');
load '.\simulatedDsm.txt'
load '.\simulatedDms.txt'
load '.\simulatedD.txt'

dms = simulatedDms;
dsm = simulatedDsm;
d = simulatedD;

%%
ofs = zeros(1,loopcnt);
biased_ofs= zeros(1,loopcnt);
filtered_delay= zeros(1,loopcnt);
% loopcnt is the number of loops, which is also the number of
% simulated samples

if (state == DMSHEAVIER)
    dms = 2*d-dsm;
    dsm = dsm;
end
if (state == DSMHEAVIER)
    dms = dms;
    dsm = 2*d-dms;
end
if (state == EQUALDELAY)
    dms = dms + 2 * (d - 0.5*(dms+dsm)) * 0.5;
    dsm = 2*d - dms;
end

for i = 1:loopcnt
    if i==1
        % unbiased offset starts with initial_offset
        unbiased_ofs = initial_offset;
    end
    ofs(i) = unbiased_ofs;
    biased_ofs(i) = unbiased_ofs + (dms(i) - dsm(i))/2;
    [flt_ofs, flt_delay]= updateofs(unbiased_ofs, dms(i), d(i), i);
    filtered_delay(i) = flt_delay;
    unbiased_ofs = decision(flt_ofs, unbiased_ofs, i, ai, ap, p);
end

diff = ofs(2:end) - ofs(1:(end-1));
figure, subplot(2,1,1), plot(ofs), title('unbiased offset'), ylim([-
3000 3000])
subplot(2,1,2), plot(ofs), ylim([-12000 12000])
figure, subplot(2,1,1), plot(diff, 'black'), title('unbiased freq off-
set')
subplot(2,1,2),plot(abs(diff), 'black'),
% The adjustments of zoom-in on y:
if state == DMSHEAVIER
    ylim([0 500]),
    title('dms >> dsm'),
end
if state == DSMHEAVIER
    ylim([0 100]),
    title('dsm >> dms'),
end
```

```
if state == EQUALDELAY
    ylim([0 300]),
    title('dsm = dms'),
end
xlim([1 loopcnt]), ylabel('abs(unbiased freq offset) [ppb]')
xlabel('samples')
figure, plot(biased_ofs), title('biased_ofs')
figure, plot(filtered_delay), title('filtered_delay')
%% - 2013.8.15
figure,
subplot(2,2,1), plot(abs(diff), 'black'), xlabel('sample'), yla-
bel('abs(frequency drift) [ppb]')
if (state == DSMHEAVIER)
    title('dsm >> dms'),
    ylim([0 100]),
end
if (state == DMSHEAVIER)
    title('dsm << dms'),
    ylim([0 500]),
end
subplot(2,2,2), plot(ofs, 'black'), xlabel('sample'), ylabel('unbiased
time offset [ns]')
if (state == DSMHEAVIER)
    title('dsm >> dms'),
    ylim([-1000 1000]),
end
if (state == DMSHEAVIER)
    title('dsm << dms'),
    ylim([-5000 5000]),
end

%% -  2013.5.12 for thesis change the pictures
figure,
subplot(2,2,1), plot(offset, 'black'), ylim([-2000 2000]), xlim([0
0.5*length(biased_ofs)])
title('real offset'),xlabel('samples'), ylabel('Time offsets[ns]')

subplot(2,2,3), plot(biased_ofs, 'black'),  xlim([0
0.5*length(biased_ofs)]), ylim([-8000 8000]),
xlabel('samples'), ylabel('Time offsets[ns]')
if state == DMSHEAVIER
    if ap == 0.05
        ylim([ -2000 7000 ]),
    else
        y_range = ylim;
    end
    title('dsm << dms (simulated biased offset)')
end
if state == DSMHEAVIER
    if ap == 0.05
        ylim([ -7000 2000 ]),
    else
        y_range = ylim;
    end
    title('dsm >> dms (simulated biased offset)')
end
if state == EQUALDELAY
    ylim([ -2000 2000 ]),
    title('dsm = dms (simulated biased offset)')
end
```

```
subplot(2,2,4), plot(ofs, 'black'), xlim([0 0.5*length(biased_ofs)])
if ap == 0.05
    ylim([-2000 2000])
else
    ylim([y_range(1) y_range(2)]);
end

xlabel('samples'), ylabel('Time offsets[ns]')
if state == DMSHEAVIER
    title('dsm << dms (simulated unbiased offset)')
end
if state == DSMHEAVIER
    title('dsm >> dms (simulated unbiased offset)')
end
if state == EQUALDELAY
    title('dsm = dms (simulated unbiased offset)')
end


subplot(2,2,2), plot(filtered_delay, 'black'),  xlim([0
0.5*length(biased_ofs)])
title('filtered delay'),xlabel('samples'), ylabel('Filtered de-
lays[ns]')

%% --2013.6.4
figure, subplot(2,1,1), plot(abs(diff), 'black'),
ylim([0 50]),
xlim([1 loopcnt]), ylabel('abs(unbiased freq offset) [ppb]')
title('dms >> dsm'), xlabel('samples'), xlim([0 length(biased_ofs)])
subplot(2,1,2), plot(filtered_delay, 'black'),
xlabel('samples'), ylabel('Filtered delays[ns]')

%%-2013.6.7
figure, subplot(2,1,1), plot(d, 'black'),title('delay'), xla-
bel('samples'), ylabel('simulated delay[ns]')
subplot(2,1,2), plot(filtered_delay, 'black'),
title('filtered delay'),xlabel('samples'), ylabel('Filtered de-
lays[ns]')


var(dms)
var(dsm)
```

%% decision.m:

```
function [ new_ofs ] = decision( flt_ofs, unbiased_ofs, loop_no, ai,
ap, p )
%
global fs
persistent base_addend;
persistent base_adjust;
persistent drift; % I
base_addend = 2147483648; % aka, 2^31
base_adjust = base_addend / (1000000000 / 1024);
if loop_no == 1
    drift = 0;
end
drift = drift + ai * flt_ofs;
adj = flt_ofs * ap + drift;

etemp = base_adjust - adj;
%
% new_adjustment = etemp*base_adjust/1024;
% new_addend = base_addend + new_adjustment;
%
if etemp > 0
    new_adjustment = etemp*base_adjust/1024;
    new_addend = base_addend + new_adjustment;
end
if etemp < 0
    new_adjustment = -etemp*base_adjust/1024;
    new_addend = base_addend - new_adjustment;
end
if etemp == 0
    new_addend = base_addend - new_adjustment;
end


% when next time receive sync message the offset would be using
% biased_ofs here is wrong, because the judgement is made on observed
% offset - biased_ofs, but the tuning is made to the unbiased offset
% - unbiased_ofs because the tunning is done to the addend, it has
% nothing to do with the asymmetric delay
new_ofs = unbiased_ofs + (fs* new_addend / 4294967296 *10 -
1000000000)*1/p;
end
```

%% updateofs.m:

```
function [ flt_o, flt_delay ] = updateofs(unbiased_ofs, dms, lo_d,
loop_no)
% updateofs takes the real offset as input, plus the asynmmetric
% delays and pass delay to delay filter, and takes the minus as
% biased_ofs and return it

persistent flt_d;
persistent last_d;
persistent last_ofs;

if loop_no == 1
    flt_d = lo_d;
    last_d = lo_d;
end
if loop_no >1
    k = lo_d;       % s - start
    s = 0;          %
    while (k>=2)    %
        k = k / 2;  %
        s = s+1;    %
    end             %
    if s==0         %
        s=1;        %
    end             % s - end
    flt_d = (lo_d + last_d)/2 + (s-1)*flt_d;
    flt_d = flt_d/s;
    last_d = lo_d;
end
flt_delay = flt_d;
% delay filter end

% offset filter begin

lo_ofs = unbiased_ofs + dms - flt_delay;

if loop_no == 1
    last_ofs = lo_ofs;
    flt_o = lo_ofs;
end
if loop_no > 1
    flt_o = (lo_ofs + last_ofs)/2;
    last_ofs = lo_ofs;
end
% ofset filter end
end
```

APPENDIX 2: PI CONTROLLER PARAMETER STUDY

```
% calculate the B and C in the notebook
close all
clear all
clc
format('long');
% state
POS_PEAK = 1;
NEG_PEAK = -1;
p = 64;
fs = 2e8;
c2 = 2^32;
c1 = 2^31;
c0 = 2^10;
A = c1 * c0 /1e9;
B = 1/p *(fs*10/c2*(c1+A*A/c0)-1e9)
C = 1/p *(fs*10/c2*(-A)/c0)
% d[n] = d[n-1] + A/p - 1/p*(kp*d[n-1]+kiSIGMAm=0 to m=n-1 d[m])
testStepsI = 0.01:0.01:1;
testStepsP = 0.01:0.01:1;
od_PairsI = zeros(1,length(testStepsI)*length(testStepsP));
od_PairsP = od_PairsI; % testStepsP and I must have the same length
od_nonzeroSum = od_PairsI;
od_Sum = 0;

ud_PairsI = zeros(1,length(testStepsI)*length(testStepsP));
ud_PairsP = ud_PairsI;
ud_peakSum = ud_PairsI + 1;
ud_Sum = 0;

all_PairsI = zeros(1,length(testStepsI)*length(testStepsP));
all_PairsP = all_PairsI;
all_nonzeroSum = all_PairsI;
all_peakSum = all_PairsI + 1;
all_index = 0;
loopcnt = 500;

for ai = testStepsI
    for ap = testStepsP
        d = zeros(1,loopcnt+1);
        d(1) = 0;
        sum_d = 0;
        % ai = ki/p; ap = kp/p;
        for i = 1:loopcnt;
            sum_d = sum_d + ai*d(i);
            % d(i+1) = d(i)+2199  - ap*d(i) - sum_d, because A/p is
too big
            % to plot a clear 3D figure,  10 is to used here to re-
place it
            % for a clear 3D figure
            d(i+1) = d(i) + 10 - ap*d(i) - sum_d;
        end
        [ymin xmin] = min(d);
        %% overdamped or crytical damping
        if (ymin == 0)
            % all
            all_index = all_index + 1;
            all_PairsI(all_index) = ai;
            all_PairsP(all_index) = ap;
            [ymin_t xmin_t] = min(d(2:end));
            all_nonzeroSum(all_index) = xmin_t+1;
            % all_peakSum(all_index) should be 1, therefore it is not
changed
            % overdamped only
            od_Sum = od_Sum + 1;
            od_PairsI(od_Sum) = ai;
            od_PairsP(od_Sum) = ap;
            od_nonzeroSum(od_Sum) = xmin;
        end
        %% underdamped
        if (ymin ~= 0)
            all_index = all_index + 1;
            all_PairsI(all_index) = ai;
            all_PairsP(all_index) = ap;
            all_nonzeroSum(all_index) = 0; %% TODO
            % all_peakSum is assigned after the "underdamped only"
            % underdamped only
            ud_Sum = ud_Sum + 1;
            % ud_peakSum = 1; % for the xmax %% This is already writen
in the inilization phase
            [ymax xmax] = max(d); %% because of the existence of the
constant in the differential expression, the max(d) would for sure not
be the d[1]
            halfT = abs( xmin - xmax ); % d[1]= 0 and positive con-
stant makes this positive forever, but for the calculation precision
reason, it could be negative
            peakIndex = xmax + halfT;
            state = POS_PEAK;
            if ( peakIndex <= (loopcnt+1))
                while (d(peakIndex)*state < 0)
                    peakIndex = peakIndex + halfT;
                    state = -1*state;
                    ud_peakSum(ud_Sum) = ud_peakSum(ud_Sum) + 1;
                    if (peakIndex > (loopcnt+1))
                        break
                    end
                end
            end
            all_peakSum(all_index) = ud_peakSum(ud_Sum);
            ud_PairsI(ud_Sum) = ai;
            ud_PairsP(ud_Sum) = ap;
        end
    end
end
[x,y] = meshgrid(testStepsI,testStepsP);
z = griddata(all_PairsI,all_PairsP,all_peakSum,x,y);
mesh(x,y,z)
hold
plot3(x,y,z)
hold off
xlabel('ai'), ylabel('ap'), zlabel('number of peaks'), title('')

figure,
[x,y] = meshgrid(testStepsI,testStepsP);
z = griddata(all_PairsI,all_PairsP,all_nonzeroSum,x,y);
mesh(x,y,z)
hold
plot3(x,y,z)
hold off
xlabel('ai'), ylabel('ap'), zlabel('climbing duration'),
```