FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Lehrstuhl für Informatik 7

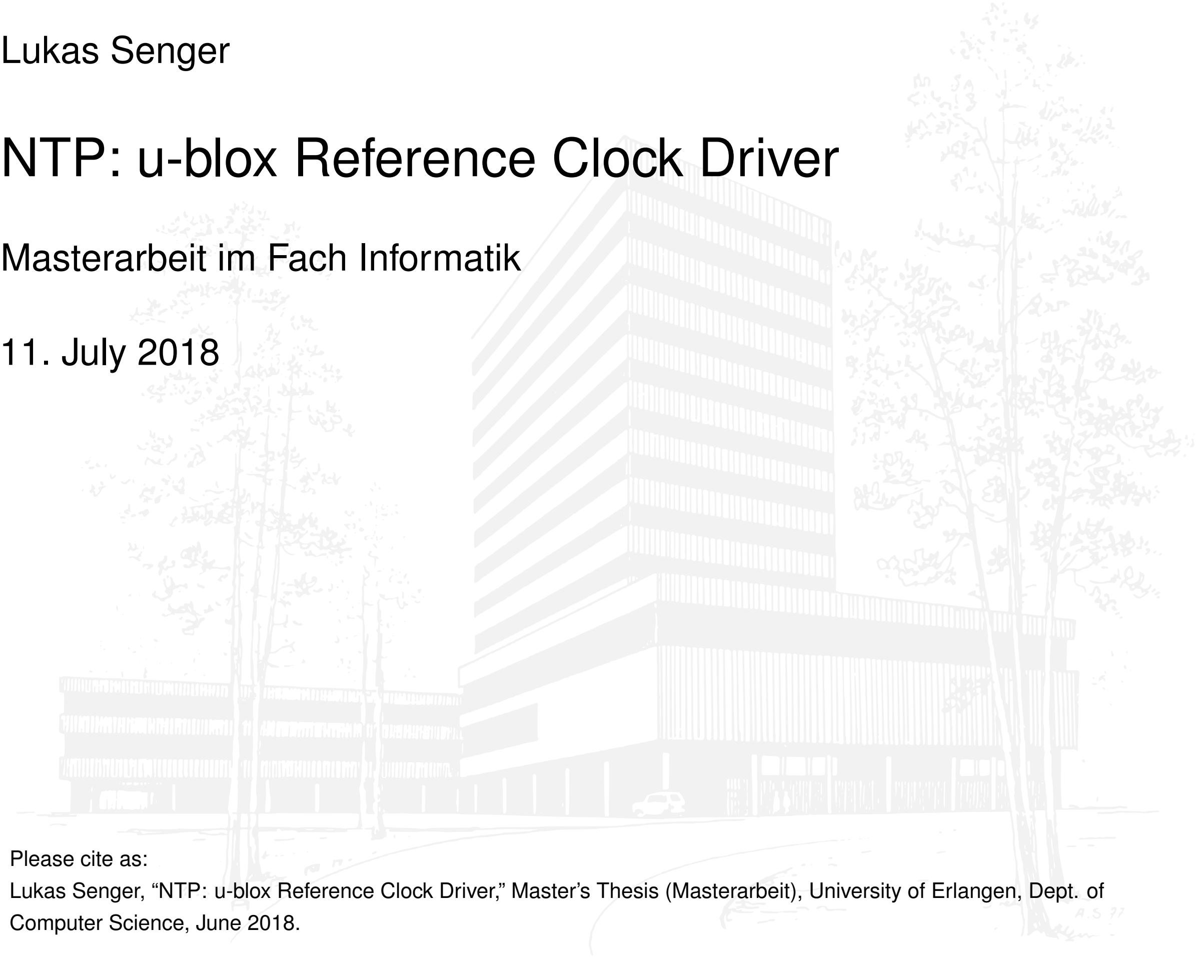
Rechnernetze und Kommunikationssysteme

Lukas Senger

NTP: u-blox Reference Clock Driver

Masterarbeit im Fach Informatik

11. July 2018



Please cite as:
Lukas Senger, “NTP: u-blox Reference Clock Driver,” Master’s Thesis (Masterarbeit), University of Erlangen, Dept. of Computer Science, June 2018.

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department Informatik
Rechnernetze und Kommunikationssysteme

Martensstr. 3 · 91058 Erlangen · Germany

http://www7.cs.fau.de/

NTP: u-blox Reference Clock Driver

Masterarbeit im Fach Informatik

vorgelegt von

Lukas Senger

geb. am 25. Oktober 1992
in Marburg

angefertigt am

Lehrstuhl für Informatik 7
Rechnernetze und Kommunikationssysteme

Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg

Betreuer:	**Dr.-Ing. Kai-Steffen Jens Hielscher**
Abgabe der Arbeit:	**26. Juni 2018**

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäSS übernommen wurden, sind als solche gekennzeichnet.

Declaration

I declare that the work is entirely my own and was produced with no assistance from third parties.
I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

(Lukas Senger)
Erlangen, 26. Juni 2018

Abstract

In this master's thesis we collect basic information about NTP and about GPS reference clocks. We use this knowledge to develop a reference clock driver for modern u-blox receivers which supports features on a level similar to the existing Oncore driver. Additionally, the new driver supports timemark mode, a novel mechanism to synchronize reference clocks which avoids local interrupt latency. We show that the new driver performs well and that timemark mode can be seen as an improvement over traditional PPS operation. Furthermore we show that the driver can be used to measure PPS latency. In particular the influence of CPU affinity and CPU load on PPS latency are investigated.

Kurzfassung

In dieser Masterarbeit sammeln wir grundlegende Informationen über NTP und GPS Referenzuhren. Wir verwenden dieses Wissen um einen Referenzuhrtreiber für moderne u-blox-Empfänger zu entwickeln, dessen Funktionalität auf gleichem Niveau wie die des existierenden Oncore-Treibers ist. Des weiteren unterstützt der neue Treiber den Timemarkmodus, einen neuartigen Mechanismus zur Synchronisation von Referenzuhren, welcher lokale Interruptlatenz umgeht. Wir zeigen, dass der neue Treiber gute Leistung erbringt und dass der Timemarkmodus als Fortschritt gegenüber von traditionellem PPS-Betrieb gesehen werden kann. Außerdem zeigen wir, dass der Treiber genutzt werden kann um PPS-Latenz zu messen. Als Beispiel wird der Einfluss von CPU-Affinität und CPU-Last auf die PPS-Latenz untersucht.

Contents

Abstract **iii**

Kurzfassung **v**

1 Introduction **1**

2 Fundamentals **3**

2.1 Timescales 3

2.2 NTP 4

2.2.1 Introduction 4

2.2.2 Mechanisms 5

2.2.2.1 Timestamps 5

2.2.2.2 On-wire Protocol 6

2.2.2.3 Architecture and Algorithms 7

2.2.2.4 Discipline 9

2.2.3 Reference Clocks 10

2.2.3.1 Examples 10

2.3 Pulse-Per-Second API 11

2.4 GNSS 12

2.4.1 Basic principles of GNSS positioning 12

2.4.2 Available GNSSs 13

2.5 The Receivers 15

2.5.1 M12M-T Oncore 15

2.5.2 u-blox 16

2.5.2.1 Timing Variants 17

2.5.2.2 UBX Protocol 17

3 Implementation **19**

3.1 Basic features 19

3.1.1 Data Types 20

3.1.2 The Receive Function 20

3.1.3 Receiver Setup 23
3.2 Leap Seconds 23
3.3 Configuration 24
3.4 Survey-In 26
3.5 PPS 26
3.5.1 Setup 26
3.5.2 Synchronization 27
3.6 Timemark 29
3.6.1 PPS Latency 31
3.6.2 Kernel Modifications 31
3.7 Monitoring 32

4 Evaluation **35**
4.1 Comparison with Motorola Oncore 35
4.1.1 Setup 35
4.1.2 Free Running 36
4.1.3 Disciplined 38
4.1.3.1 Setup 38
4.1.3.2 Results 39
4.1.3.3 Hardpps 41
4.1.4 Timemark Mode 42
4.2 PPS Latency 44
4.2.1 FreeBSD 44
4.2.1.1 Interrupt Load 46
4.2.1.2 Level Converters 46
4.2.2 CPU Affinity 47
4.2.3 CPU Load 50
4.3 Windowsill Measurements 51

5 Conclusion **55**

Bibliography **61**

Chapter 1

Introduction

Timekeeping is an essential part of modern computing. Logging, database and network applications rely on accurate timestamps, to only name a few. In order to make these timestamps comparable, computer clocks have to be synchronized. One way to achieve this is NTP, the Network Time Protocol.

NTP synchronizes a hierarchy of servers, where time is passed from top to bottom. Additionally to communication with other NTP instances, servers may have access to so-called reference clocks. These are accurate hardware clocks that report Coordinated Universal Time (UTC). Reference clocks might for example be telephone modems or GPS receivers. [1] [2]

GPS or GNSS (Global Navigation Satellite System) receivers lend themselves to this application as they are synchronized directly to the atomic clocks of their satellites and are fairly cheap. Some manufacturers even offer dedicated receivers optimized for timing applications.

Especially devices that provide a PPS (pulse per second) signal can be used to achieve good performance. [3] The PPS signal is sent by the receiver at the top of the second and raises an interrupt on the computer (usually via the serial port) which is then timestamped. The difference between the timestamp and the top of the second then gives us the offset of our local clock.

However, there are several delays between PPS generation and timestamping. Probably the largest and most variable of which is the interrupt latency. There are receivers, that offer another way to synchronize an external (to the receiver) clock to the receiver clock. Timing receivers by u-blox for example, allow access to an interrupt line. Rising and falling edges on that line are then timestamped on the receiver and those timestamps can be accessed. [4] The offset is now given by the local time when the edge was raised, minus the receiver time when the edge was detected.

NTP uses a standardized interface to make use of PPS signals. The PPSAPI defines a common way to access PPS signals for UNIX-like operating systems. One of its optional features is the sending of an "echo" pulse directly after timestamping an incoming PPS pulse. [5]

In this thesis we develop a reference clock driver for a u-blox receiver that uses the PPSAPI's echo mechanism to generate an input signal for the receiver's interrupt line. This results in two corresponding timestamps, similar to what is produced by only using the PPS signal. However with this method the interrupt latency is moved from the general purpose computer to the GNSS receiver, where it should be lower and more manageable.

We use this mechanism to investigate PPS latency as well as providing an accurate reference time to NTP.

Chapter 2

Fundamentals

2.1 Timescales

There is a number of different timescales that are important for this thesis. They all somehow relate to *UTC*. UTC is the basis for civil timekeeping in most countries today. It is based on two other timescales. Its rate corresponds exactly to that of International Atomic Time (TAI). A second according to TAI and the International System of Units (SI) is defined in reference to the period of certain radiation produced by the caesium 133 atom.

Historically humans have used celestial bodies like the sun as a reference for time. However TAI is not bound to the duration of a day. Its second was defined as one 86 400th of a day of the year 1820, but earth's rotation is slowing down, so a day is now slightly longer than in 1820. There exist timescales that are tied to the rotation of the earth, one of which is UT1, the second basis for UTC. UTC is kept within roughly 1s of UT1 by introduction of leap seconds in semi-regular intervals. As of the time of this writing, the difference $UTC - TAI$ equals $-37s$ [6]. [7]

Related to UTC is *GNSS time*, the timescales of the respective GNSSs. They are kept with atomic clocks belonging to the respective GNSS and as such their particulars are different in each case. However their offsets to UTC are regularly computed by the GNSS operators and uploaded to their satellites, where in turn they are available to receivers. [8]

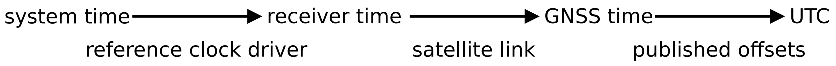


Figure 2.1 – The relation of local system time to UTC.

The receivers usually keep time with a quartz oscillator. They try to synchronize their *receiver time* as close as possible to the GNSS time. The last step in the chain and the focus of this thesis is the synchronization of *system time*, i.e. the local time on a computer, to receiver time. Figure 2.1 shows how system time can be traced back to UTC.

2.2 NTP

2.2.1 Introduction

Most clocks today, including those in computers, also use quartz-crystal oscillators to keep time. These oscillators perform quite well on small timescales, however if you look at larger intervals, even small deviations from the SI second will result in an ever growing offset from UTC. [9]

In order to prevent this, the clock has to be disciplined in regular intervals. In the case of wristwatches, we do this by setting them when we deem the offset too large. In the case of atomic standards, the quartz is stabilized by atomic transitions in a feedback loop. In the case of computer clocks, neither of these methods is practical. [9]

NTP solves this problem by synchronizing computer clocks over a network (e.g. the Internet) with some of those computers synchronized to more accurate clocks. It uses a hierarchical architecture of time servers where the primary servers are connected directly to external time sources (reference clocks) while other secondary servers are connected to these primary servers. Servers are assigned a stratum number beginning with "1" at the primary servers and increasing with every hop. [10]

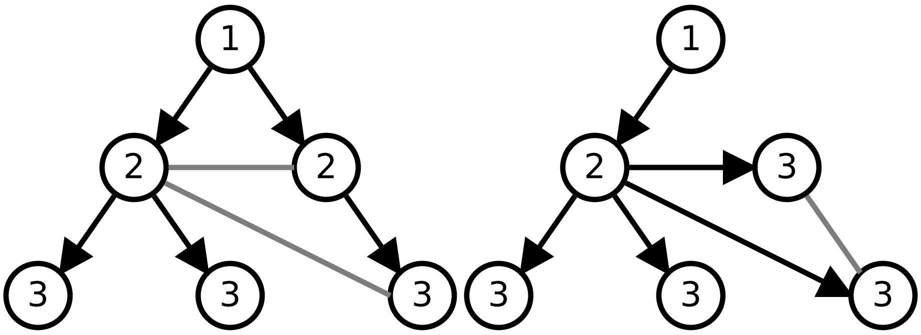


Figure 2.2 – Left: A typical NTP subnet with one primary server (stratum 1).
Right: The same subnet with one active link removed. [10]

A higher stratum number means lower achievable accuracy. Figure 2.2 shows an example network with stratum numbers noted at the nodes, active synchronization links marked as arrows and backup paths marked as grey lines. The system can reconfigure itself in case of outages. The network on the right side shows what the left side would look like with one of its links out of service. [10]

NTP's history can be traced back to 1979 and the protocol has seen continuous development since. [11] In 2014 it was estimated to be in use on over 24 million PCs on the Internet and in private networks. It has even been deployed on the seabed and in space. [12] [13] It is currently specified as version four in RFC 5905 [1] with extensions defined in RFC 7822 [14]. However, some newer modifications are not formally specified. [15]

There are alternatives like PTP (Precision Time Protocol) and DTSS (Digital Time Synchronization Standard), however in this thesis we focus solely on NTP.

2.2.2 Mechanisms

2.2.2.1 Timestamps

The basic building block for calculations in a timekeeping application is a timestamp. NTP has two principal formats to represent points in time. Timestamps, which are used in the on-wire protocol and wherever a small memory footprint is of concern, are shown on top in figure 2.3. They consist of 32 bits numbering the seconds and 32 fractional bits. Timestamps span an era of roughly 136 years with a precision of 232ps. [16] [17]

Given a prime epoch of 1900, this may raise concern about what happens in 2036. For calculations where rollover might be a problem, NTP internally uses the datestamp format shown also in figure 2.3. It is essentially a timestamp with and era counter of 32 bits inserted in the front and 32 additional fractional bits appended to the end. This allows representation of times beyond the age of the universe with precision below what is practically measurable. [16] [17]

It can be shown that using these timestamp formats means an NTP client set to within 68 years of a server will synchronize to that servers time without ambiguity. [16] [17]

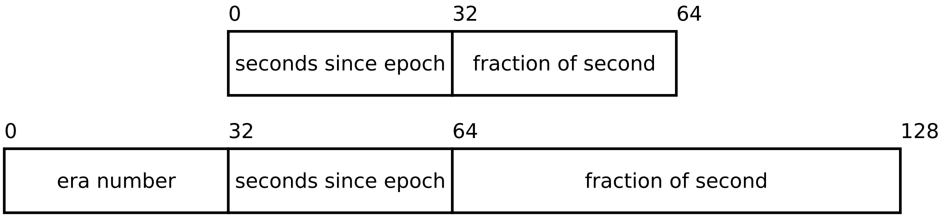


Figure 2.3 – Top: The 64 bit timestamp format. Bottom: The 128 bit datestamp format. [16]

2.2.2.2 On-wire Protocol

In normal master-slave operation, i.e. one server A synchronizing to a lower stratum server B, NTP takes four timestamps to calculate the offset between the two clocks and the round-trip delay between the two computers. As figure 2.4 shows, timestamps t_1 and t_2 are taken when A sends the first message and when B receives it. t_3 and t_4 are taken when B sends the reply and A receives it.

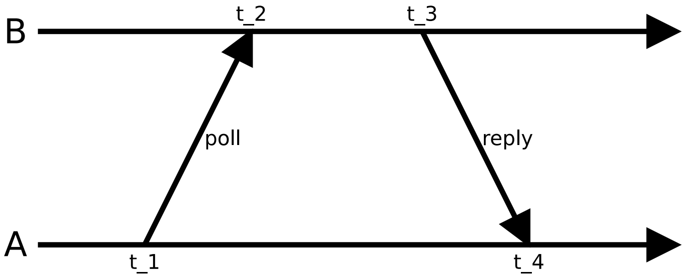


Figure 2.4 – The four timestamps used by NTP. [18]

Assuming that the two paths A→B and B→A have identical network delays, we can now calculate $offset$ and $delay$ as follows: [16] [18]

$$offset = \frac{(t_2 - t_1) + (t_3 - t_4)}{2} = \frac{t_3 + t_2}{2} - \frac{t_1 + t_4}{2} \tag{2.1}$$

$$delay = (t_4 - t_1) - (t_3 - t_2) \tag{2.2}$$

If the network delays are not equal, the offset will be off by half their difference, as we can show by adding additional delays to both directions: [18]

$$offset + \Delta_{offset} = \frac{((t_2 + d_{AB}) - t_1) + (t_3 - (t_4 + d_{BA}))}{2} \tag{2.3}$$

$$= \frac{(t_2 - t_1) + (t_3 - t_4)}{2} + \frac{d_{AB} - d_{BA}}{2} \tag{2.4}$$

$$\Delta_{offset} = \frac{d_{AB} - d_{BA}}{2} \tag{2.5}$$

Note that t_1 and t_4 relate to the clock of A, whereas t_2 and t_3 relate to that of B. Figure 2.5 shows the offset by shifting the arrows in relation to one another. It also visualizes the offset and delay calculation. If the arrows were shifted back into alignment (by adjusting A's clock forward), the red area and thereby the offset would disappear and the blue delay would be symmetrical.

Other modes of operation exist but for a basic understanding of how NTP works, understanding the above principles is enough.

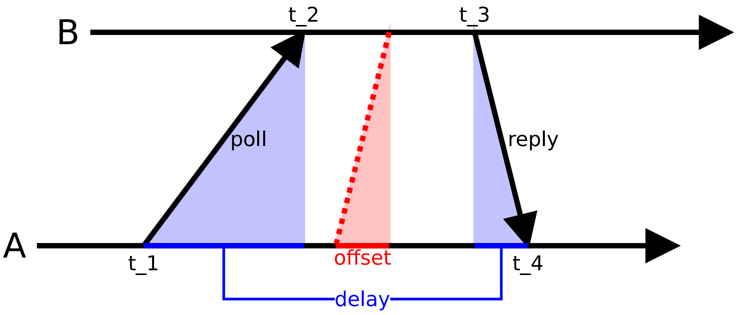


Figure 2.5 – Delay and offset calculation with the clock on B running ahead.

2.2.2.3 Architecture and Algorithms

.

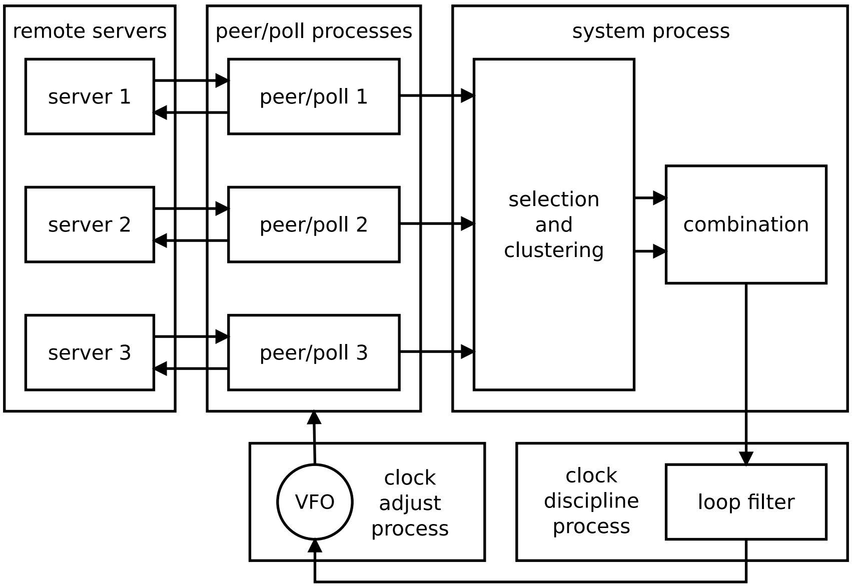


Figure 2.6 – The overall architecture of NTP as presented in RFC 5905 [1]. Clock statistics move from remote servers (or reference clocks) through the system where they are narrowed down and combined. Eventually they are used to discipline the variable frequency oscillator (VFO) of the computer.

The on-wire protocol is implemented in the peer/poll process. Polling happens at intervals, ranging from several seconds to several days, optimized for accuracy and low network load. The process runs a number of access control and security checks. If these checks are passed the calculations described in the previous section are used to get offset and delay values. [1]

At this point the *offset*, *delay* and *jitter* variables for this peer are determined. For each peer the offset and delay samples are ran through a filter algorithm, which returns the sample with the lowest delay out of a sliding window with eight slots. The jitter is set to the root mean square (RMS) of the differences of the other offsets to the selected one. Figure 2.7 shows the effect of the filter algorithm. [1] [19]

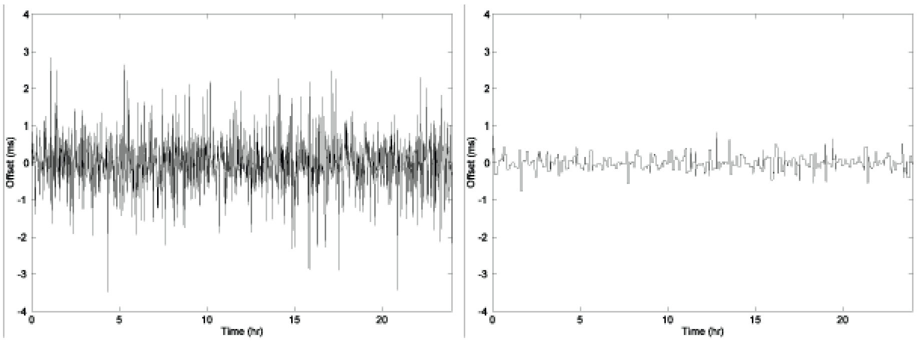


Figure 2.7 – Left: Raw offset values plotted over time. Right: Filtered values. [19]

The peers are ran through a set of selection and clustering algorithms to narrow them down to a number of *survivors*. First, the peers that pass some basic sanity checks are considered *selectable*. The clock select algorithm narrows this down to the *truechimers* with a technique gleaned from DTSS. [1]

A correctness interval is defined around the offset of each peer. The true offset θ of a peer lies somewhere in $[\theta_0 - \lambda, \theta_0 + \lambda]$ where θ_0 is the offset and λ the so-called root distance, a measure for the potential error of θ_0. The algorithm now determines an intersection interval, which is the "smallest interval containing points from the largest number of correctness intervals" [20]. Peers with a correctness interval that intersects with the intersection interval are considered truechimers and the rest are discarded as *falsetickers*. If the number of truechimers is lower than half the peers, the algorithm fails. Figure 2.8 shows an example with three truechimers and one falseticker. [1] [20]

Finally the cluster algorithm narrows down the truechimers to survivors. The algorithm discards one peer each round, based on a statistical metric called the selection jitter. The algorithm terminates either when the minimal number of survivors (by default three) is reached or when the maximal selection jitter is lower than the minimal peer jitter, i.e. the statistical outliers have been removed. [1] [21]

All survivors are used to calculate a combined offset with peers weighted by their root distance. This combined offset is used to adjust the clock. [22]

The survivor with the lowest root-distance is designated the *system peer*, unless this is prevented by some precautions against excessive hopping between clocks. This can happen in fast local networks. The system peer's statistical values are passed on to dependant peers. [22]

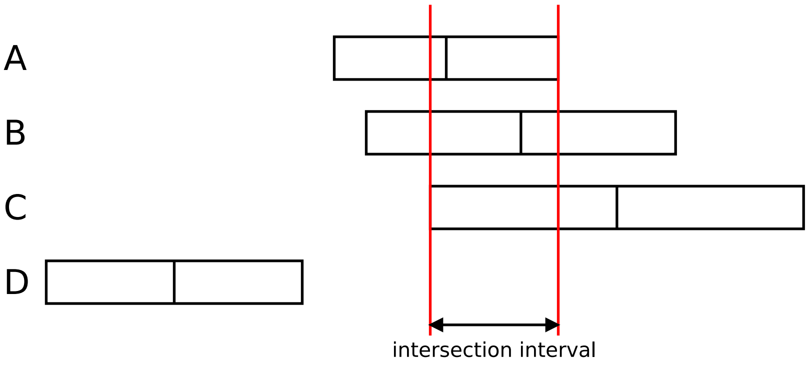


Figure 2.8 – Correctness intervals for three truechimers (A, B and C) and one falseticker (D) with their respective offsets θ_0 marked. Note that a truechimer's offset can lie outside the intersection interval. [20]

2.2.2.4 Discipline

After the algorithms described in the previous section have carefully prepared a final offset, the VFO has to be adjusted. This process is commonly called the *discipline*. In NTP the discipline can be described as a feedback loop as seen in figure 2.9, where V_s is the final surviving offset. In NTP version 4 specifically the loop is a hybrid between a phase-locked loop (PLL) and a frequency-locked loop (FLL). [23]

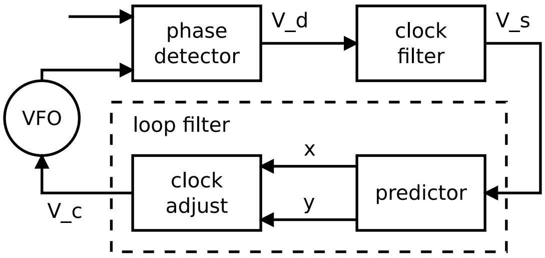


Figure 2.9 – The NTP clock discipline as shown and described in [23].

The clock adjust process uses three predictors to adjust the clock frequency. The offset is used for phase correction and two separate predictors are used for frequency correction. The PLL component predicts the frequency correction as an integral of $V_s\tau$, where τ is the poll interval. Whereas the FLL prediction is an exponential average of V_s/τ. The two frequency predictions are then combined weighted by the RMS error over previous predictions. Without going into further detail, this results in the PLL prediction dominating during high network jitter and the FLL prediction dominating when high oscillator wander is present. [23]

2.2.3 Reference Clocks

The mechanisms described in the previous section form the basis for time synchronization in NTP. However for them to be useful in the context of real life, the synchronized system as a whole needs to be synchronized to UTC. Otherwise the time kept in the NTP network is not necessarily related to that used outside of it.

In NTP the interface to the outside world are so-called *reference clocks*. These are hardware clocks which provide accurate UTC, typically via a serial, parallel or USB port. Most reference clock drivers communicate with GPS/GNSS receivers, however, there are also drivers for modem clocks which get the current time via a telephone connection, for shortwave radio clocks as well as a number of special drivers. [24]

Reference clocks are largely treated the same as other peers. They are assigned a fictional stratum of 0 and metrics like the root distance mentioned above are also assumed 0. [25] Reference clocks have a special address of the form `127.127.t.u` where `t` is a number specifying the driver type and `u` a number from 0 to 3 differentiating the driver instance or *unit*. Like peers, they are polled at regular intervals and the resulting offsets go through the same algorithms described in section 2.2.2.3. [24]

GNSS reference clocks usually provide updates more often than the polling interval (usually once per second). Before being transferred to NTP the offsets are therefore stored in a circular buffer. When the buffer is read, a median filter is applied to its contents to reduce jitter. [25]

2.2.3.1 Examples

One reference clock driver that deserves special mention is the GPSD driver (type 46), because it encompasses a large number of GPS receivers. [26] GPSD is a GPS service daemon which acts as a layer between GPS devices available to a computer and applications using GPS data. The NTP driver acts as a client application to GPSD. [27]

The NMEA driver (type 20) is another generic driver that speaks the NMEA 0182 standard, which is supported by a number of different receivers.

The Oncore driver (type 30) for Motorola Oncore GPS devices was used as a reference for writing the new u-blox driver of this thesis.

All of the above drivers have built-in PPS support. However, some drivers outsource this functionality to the generic PPS driver (type 22). It uses another driver for numbering the seconds and the API described in the next section for aligning the seconds to the PPS pulse.

2.3 Pulse-Per-Second API

RFC 2783 [5] defines a PPS signal as “a series of pulses, each with an "asserted" (logical true) phase, and a "clear" (logical false) phase. The two phases may be of different lengths.”

In order to accurately timestamp PPS pulses, it has to happen immediately after the phase change is detected, usually this is done in an interrupt handler routine. The interrupt is often triggered by a signal transition on the DCD (Data Carrier Detect) line of the serial interface. But other interfaces exist, like GPIO or parallel. The timestamps have to be made available to user space. RFC 2783 defines an interface for accessing those timestamps on UNIX-like operating systems. [5]

The interface contains functions for selecting a PPS source, reading its capabilities, accessing its latest PPS timestamps as well as reading and writing configuration parameters for the time source. Listing 2.10 shows all required functions of the interface.

Function Name	Purpose
time_pps_create()	selection
time_pps_destroy()	
time_pps_setparams()	querying/configuration
time_pps_getparams()	
time_pps_getcap()	
time_pps_fetch()	accessing timestamps

Figure 2.10 – Required functions of the PPSAPI [5]

Among other things, the parameters allow configuration of an echo signal. If activated and implemented, the kernel will generate an echo pulse immediately after timestamping the incoming PPS pulse. [5] This is required for timemark mode (see section 3.6), unfortunately it is only sporadically implemented.

The specification does not give implementation details, however, implementors are urged to give some import to the initial timestamping, which should happen as soon as possible after the interrupt is raised. [5]

The optional function *time_pps_kcbind* allows the user to specify consumers inside the operating system kernel which should be provided the PPS information. In particular, this is meant to support the *hardpps()* kernel consumer. If activated, it disciplines the system clock inside the operating system kernel. [5]

2.4 GNSS

2.4.1 Basic principles of GNSS positioning

GNSS stands for global navigation satellite system. As the name says, these are systems that enable navigation globally via satellite communication. The most known system is probably GPS, operated by the US Air Force. The methods for positioning described in this section were first developed and deployed as GPS, however most GNSSs operate in a similar fashion. They are passive systems, which means that the user-side hardware only needs to receive signals. [8]

In order to calculate its own position, the receiver has to know two things: the position of the satellites in view and the distance to those satellites. If both are known, the receiver can calculate a sphere around each satellite with the radius of the sphere given by the distance between receiver and satellite. Assuming that three satellites are in view, this will result in three intersecting spheres. Two spheres intersecting generate a circle of possible positions and the third sphere narrows this down to two intersection points (see fig. 2.11). One of the two points will be closer to earth, it denotes the receiver position. [8]

The position of the satellites is broadcast and therefore known to the receiver. For the distance however, a more elaborate scheme is required. Satellites broadcast a pseudorandom noise (PRN) code, that is known to the receiver. The receiver generates the same PRN code locally. Given that the receiver clock and the satellite clock are synchronized, the PRN code from the satellite will be somewhat delayed with respect to the locally generated PRN code as shown in figure 2.12. This delay is the result of the signal having to travel from the satellite to the receiver. As the speed of light is known (this is somewhat simplified as in reality relativistic and atmospheric effects have to be accounted for), the receiver can now calculate its distance to the satellite from the offset between the two PRN codes. [28]

While the satellites have very accurate atomic clocks, the receivers usually keep time with a crystal oscillator. This means, that even if they were synchronized at some point, the receiver time and the GNSS time will drift apart fairly quickly, resulting in an offset Δt between the two clocks. So the time difference between PRN codes can only produce pseudo distances. To get the real receiver position from the pseudo position Δt has to be known. Overall, this gives us four unknowns, three spacial coordinates and Δt, meaning four satellites are needed to produce a navigation solution. [28]

The fact that Δt is being solved for, means the receiver clock can be kept in sync with the satellite clock. Accurate time can therefore be seen as a side product of positioning. If the receiver used for timing is stationary, accuracy can improve.

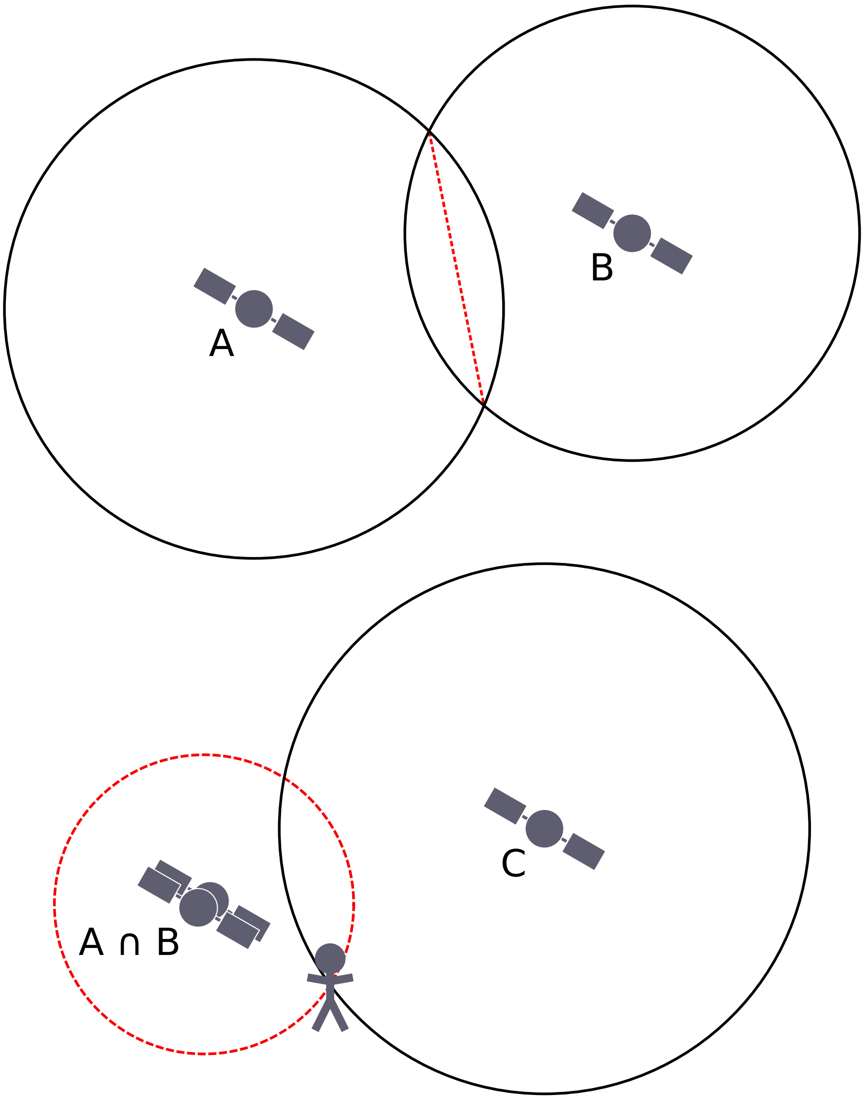


Figure 2.11 – Top: Two spheres intersecting in a circle (marked by the red line). Bottom: The same scene with a third satellite and rotated to face the intersection circle. One of the intersection points between the third sphere and the red intersection circle is the receiver position.

Especially if the position of the receiver is known precisely, in which case a single satellite is theoretically enough to solve for time.

2.4.2 Available GNSSs

Today, there exist several GNSSs, the earliest program, excluding prototypical forerunners of GNSSs, was the American GPS, which started in 1970 and was declared fully operational in 1995. It nominally features 24 satellites, however it is currently

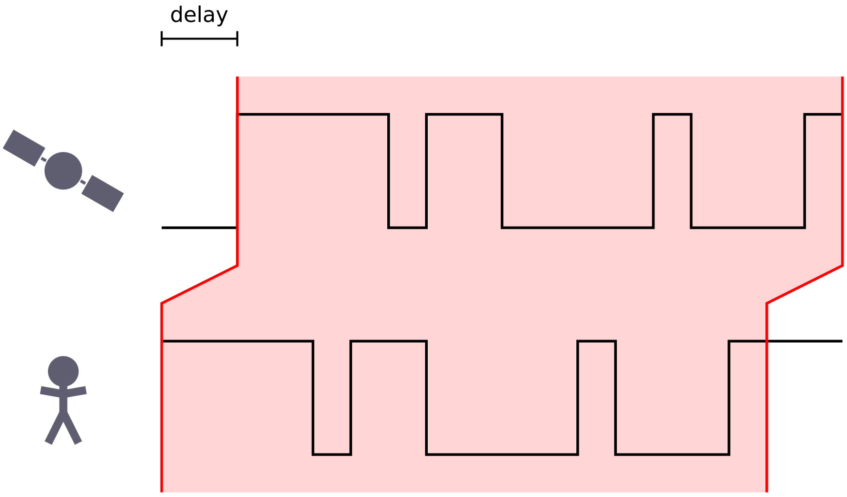


Figure 2.12 – Measuring the transmission delay from satellite to receiver with PRN codes.

overpopulated at 31 satellites. GPS provides two services, one for civilian use and a higher accuracy one for military use. [28]

Similarly the Russian system GLONASS (Global'naya Navigatsionnaya Sputnokiva Sistema) started in the early seventies and was partially opened for civilian access in 1995. However, due to lifetime problems of the satellites and insufficient replenishment, the original 24-satellite constellation soon degraded. A low point of 7 satellites was reached in 2001, which has since been continually improved and reached full 24-satellite service in 2011. Further modernization is in process. [28]

In 1994 China started work on its BeiDou (北斗, meaning "Big Dipper"), also known by its English name COMPASS. The system is currently in phase two, offering regional positioning with 14 satellites. The BeiDou global system is set to be completed in 2020 with 35 satellites in different types of orbit. It will offer an open and an authorized service. [28]

Another younger system is the Galileo system started by the European Union (EU) and the European Space Agency (ESA) in 2003. Currently comprising 22 satellites [29], the complete constellation of 24 satellites plus spares will be completed in 2020. Galileo will provide a free and a higher accuracy commercial service. [28]

There is also a number of regional and complementary systems which can offer improved accuracy. However, for the purposes of this thesis the above overview over the global systems shall suffice.

In order to provide accurate positioning, each GNSS has an internal time scale, kept by the collection of atomic standards on its satellites. This GNSS time is usually uniform to avoid problems around leap seconds, the notable exception being GLONASS, which follows UTC leap seconds. For navigation purposes there is no need to align the GNSS time to any other timescale. In reality however each GNSS is steered towards UTC as realized by the timing laboratories of their respective countries. [30]

This means, that all systems but GLONASS have an integer second offset to UTC. GPS and Galileo share the same offset. The sub-second differences between the various timescales are in practice in the low nanoseconds. GLONASS has slightly worse performance in the hundreds of nanoseconds due to the historical degradation of the system. With the ongoing efforts to improve GLONASS, it is expected to catch up with other systems soon. [30] [28]

2.5 The Receivers

2.5.1 M12M-T Oncore

The starting point for the u-blox reference clock driver was driver type 30 for GPS receivers by Motorola supporting the Motorola Binary Protocol. We chose it because similar features and similar configurability were planned for the u-blox driver. Also a hardware unit was available. The receiver, originally produced by Motorola, now by i-Lotus, comes in a positioning and a timing variant and supports only GPS.

The timing variant has several features that make it useful in timing applications. It has a *Position-Hold* mode that improves performance in stationary applications. To get an accurate value for its current position, it can perform a *site survey*. This means it automatically collects 10.000 position samples and averages them. Afterwards, it forces the receiver into Position-Hold mode. [31]

When in Position-Hold mode, the receiver only requires one satellite to solve for time, as the receiver position is known. If multiple satellites are being tracked, the surplus measurements can be used to compute a more robust solution. Motorola has developed the *T-RAIM* (T stands for Timing and RAIM for Receiver Autonomous Integrity Monitoring) algorithm for this purpose. It averages measurements and removes outliers that deviate too much from the average. [31]

The receiver communicates via a serial interface. It supports NMEA messages as well as the Motorola Binary Protocol. Messages in the binary protocol start with the characters `@@` and two letters identifying the message, followed by a variable length payload. The message ID and payload are used to compute a checksum, which terminates the message along with a carriage return and line feed. More details can be found in the user manual [31] and in section 3.1.1.

For timing purposes the receiver can output a 1 PPS signal and a 100 PPS signal. The reference clock driver uses the 1 PPS. It accesses the PPSAPI directly without using the PPS driver (see section 2.2.3.1). The driver also supports an additional configuration file. Among other things, it can be used to set the position (without a site survey), configure the PPSAPI and toggle T-RAIM. It also allows setting a masking angle below which satellites are ignored, it defaults to 10° on the M12M-T receiver. [31] Figure 2.13 shows an overview over all configuration options.

Option	Explanation
MODE	Set one of four modes depending on how much initialization is needed. This, for example, controls whether a site survey is performed.
LONG	Set longitude.
LAT	Set latidute.
HT	Set height above GPS ellipsoid.
DELAY	Set antenna delay.
OFFSET	Set offset of PPS pulse from top of second.
ASSERT/CLEAR	Define which edge of the PPS pulse should be used.
SHMEM & (POSN2D/POSN3D)	Save position data to a shared memory file.
CHAN	Manually set the number of channels supported by the receiver.
TRAIM	Manually toggle T-RAIM.
MASK	Set the satellite mask angle.
PPSCONTROL	Allows toggling PPS depending on receiver state for certain receivers.

Figure 2.13 – All configuration options supported by the Oncore reference clock driver. Further explanations can be found in the source code.

2.5.2 u-blox

The receivers by u-blox, which are the target of this thesis, come in a variety of variants. We focus on the timing variants, with the option of falling back onto standard precision positioning receivers.

In contrast to the Oncore receiver, the u-blox receivers are capable of using all four major GNSSs and combining their measurements into one navigation solution. In addition, they support several GNSS augmentation systems, which are not relevant for this application.

The interpretation of GNSS measurements can be aided by setting a "dynamic platform model", which describes the receiver's operation environment. Among the options is a *stationary* mode, which restricts the receivers velocity to zero for more accurate timing solutions.

The receivers communicate via several serial communication ports (DDC, UART, USB and SPI). Via these ports, messages in NMEA format, UBX (proprietary u-blox protocol) format and RTCM (Radio Technical Commission for Maritime Services) format can be sent. The ports can be freely assigned protocols, including multiple protocols per port and multiple ports per protocol.

U-blox receivers also provide a time pulse that can be configured with similar freedom. However, it defaults to a 1PPS pulse.

2.5.2.1 Timing Variants

The timing variants have some additional features. They can be put into *Time Mode*, which the documentation [4] defines as

> [...] a special receiver mode where the position of the receiver is known and fixed and only the time is calculated using all available satellites. This mode allows for maximum time accuracy, for single-SV solutions, and also for using the receiver as a stationary reference station.

As the definition says, the receiver position has to be known. It can either be set by the user or measured by *Survey-in*. This is similar to the Oncore receivers site survey, with the difference being, that it doesn't average a set number of samples (10.000 for the Oncore), but requires two configurable conditions to be met. First, a minimum observation time can be defined and the receiver will not stop the Survey-in before this time has passed, no matter how many actual samples were taken in that time. The second condition is a maximal standard deviation for the position. [4]

For accurate timing the position, whether set manually or determined by Survey-in, should be known to within less than 1m. For the minimum observation time, the documentation recommends values between a few minutes and one day. [4]

Additionally, the timing variants have the *timemark* feature, which is very important for this thesis. It allows input signals to be timestamped on receiver side. For this purpose, the receiver provides an external interrupt line and a dedicated UBX message. Rising edges on the interrupt line will be timestamped and the result sent to the computer at the next navigation epoch. [4]

2.5.2.2 UBX Protocol

To take full advantage of the receiver features, the u-blox proprietary protocol (UBX) has to be used. The protocol uses 8-bit binary data and each message is protected by a checksum. It features a modular design based on message classes and types. Messages are referred to by shorthands for their class and type (e.g. CFG-PRT for configuring I/O ports). [4]

Configuration messages (i.e. class CFG) as well as some other messages will be answered by a positive or negative acknowledgement, depending on whether the message was processed correctly. Many messages can be polled by sending the same message type without payload or with a single variable defining the poll request. [4]

Figure 2.14, which is taken from the u-blox M8 receiver description [4], shows the structure of a UBX message frame. Two header characters are followed by a two-stage message id. The next two bytes hold the length of the subsequent payload in bytes. The message is terminated by two checksum bytes. The checksum is calculated over everything but the two header characters and the checksum itself using the 8-bit Fletcher algorithm. [4]

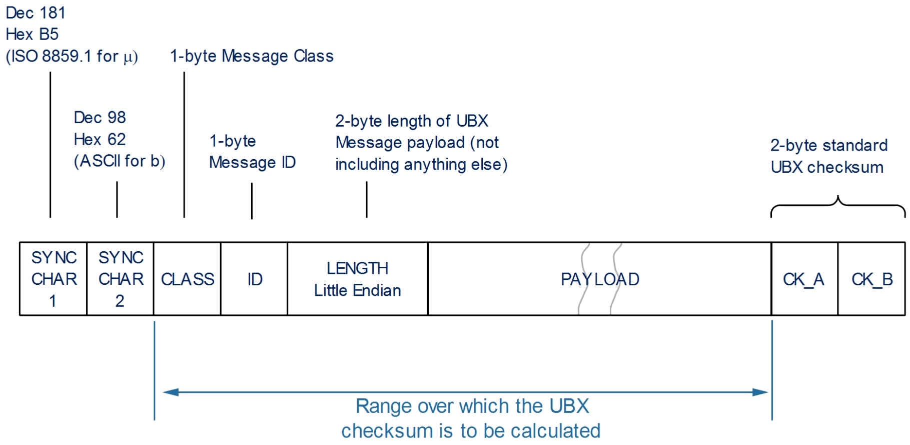


Figure 2.14 – Structure of a UBX frame [4]

The payload consists of a number of variables depending on the message type. Variables can have a number of different types as well. These are listed in the receiver description. Payloads are designed in such a way, that they can be represented in code as structures, without structure packing being a problem. [4]

Chapter 3

Implementation

3.1 Basic features

To be of any use, the u-blox reference clock driver has to be able to get the current time from the receiver. In order to do that is has to be able to read and write messages from and to the receiver. It has to be able to parse those messages and use them to set up the receiver as needed. The driver also needs to be able to pass on the time to NTP.

The interface between NTP and the driver is made up of two parts. There are *driver functions* that need to be implemented on the driver side to be called from NTP and there are *interface functions* that can be called from driver-side code. The driver functions can be described as *startup*, *shutdown*, *receive*, *poll* and *timer*.

Startup is for setting up internal house-keeping. Drivers should use this function to open serial devices, register the driver with NTP and set up the *unit struct*, a structure that holds all information required for a single instance of the driver. In the u-blox driver we open the device `/dev/ubx.serial.%u`, where `%u` is the unit number (see section 2.2.3) for serial communication. Then the configuration is read and initialization messages are sent to the receiver. The code for sending messages is an adapted version of the relevant code in GPSD.

Eventually the driver will be terminated. Before this happens, shutdown is called, to give it a chance to clean up everything that was set up in startup. Between those two points the main workload happens in receive.

Receive is called whenever chunks of data arrive on the serial interface. The chunks are assembled into messages, checked for sanity, and routed to handling functions depending on their class/type identifier, where they are cast into structures for improved readability. If we are operating without PPS, the local time is given by the arrival timestamp of the first chunk of a message. The receiver time can be parsed from the message and the pair is written to the circular buffer (see section 2.2.3) in one of the message handlers.

The poll and timer functions are called periodically. In poll the interface function receive (not to be confused with the driver function by the same name) is called to simulate packet reception. The timer is called once per second and its task depends on the driver. We use it monitor the progress of site surveys.

3.1.1 Data Types

The Motorola Binary Protocol uses, somewhat cryptic, two letter names for its messages. In combination with the rather unstructured message contents, this makes the Oncore reference clock driver very hard to read.

To make reading the code as easy as possible for the u-blox driver, most names are taken from the protocol specification. As mentioned above, once parsed, messages can be accessed as structs. Their members have the same names as the corresponding message elements. The only difference is that camel case has been converted to underscores between multiple words in one name. The message names have also been converted to lower-case underscore notation for the struct names.

Variables can have one of thirteen specified types. Figure 3.1 shows the corresponding C types used in the driver. Discrete variable values are available as enums with the names converted to upper-case underscore notation.

Short Name	Type	Size	C Type
U1	Unsigned Char	1	u_char
RU1_3	Unsigned Char	1	unused
I1	Signed Char	1	char
X1	Bitfield	1	bitfield1
U2	Unsigned Short	2	uint16_t
I2	Signed Short	2	int16_t
X2	Bitfield	2	bitfield2
U4	Unsigned Long	4	uint32_t
I4	Signed Long	4	int32_t
X4	Bitfield	4	bitfield4
R4	IEEE 754 Single Precision	4	unused
R8	IEEE 754 Double Precision	8	unused
CH	ASCII/ISO 8859.1 Encoding	1	unused

Figure 3.1 – Data types in the UBX protocol and in C. [4]

3.1.2 The Receive Function

Since the receive function is at the core of the driver, it deserves a closer look. The function receives binary data in chunks. We can make no assumption about the size and content of these chunks because they vary wildly depending on the connection between receiver and computer. In some situations the data arrived in single bytes, whereas in other situations multiple complete message were contained in one chunk. For understanding this section, it is useful to look at the structure of a UBX frame again (fig. 3.2).

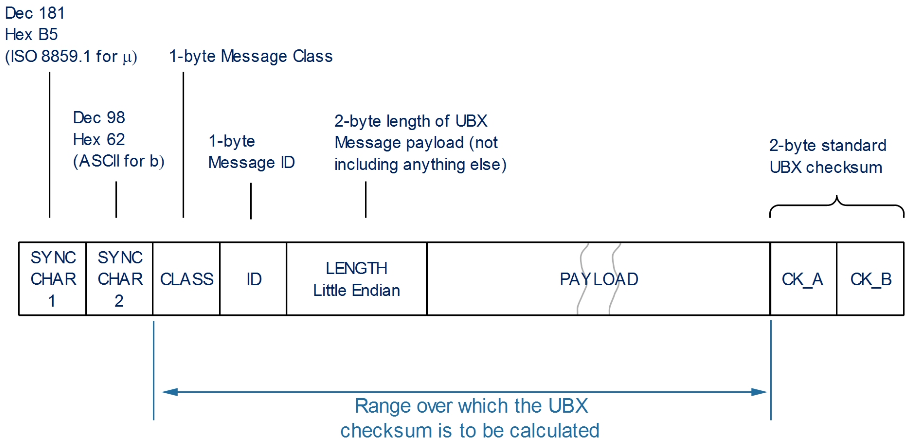


Figure 3.2 – Structure of a UBX frame [4] (same as figure 2.14)

We use a buffer of length 1000 bytes (the Oncore driver uses 500 bytes) to store incomplete messages. Every time a chunk is stored to the buffer, we go through the buffer, looking for the two byte preamble "μb". Once we find this combination we can be reasonably sure to have found the start of a new message. To find out if all of the message is available to us, we need to be able to read the length field, which means we need at least six bytes. If they are not available or if the length field tells us that we don't have enough data yet, we wait for the next receive call. Figure 3.3 shows an example of four messages arriving in five chunks. The state of the receiving buffer and the knowledge about messages is shown for each call to receive before and after processing the new chunk.

If we do have enough data, we compare the checksum. If it is correct, the message is passed on to the function `ubx_msg()`. If not, we return to the start of the faulty message and restart searching for a preamble. To avoid finding the same one again, we skip the first synchronization character. Once all messages in the buffer have been found, any remaining data is probably the beginning of a new message and is moved to the front of the buffer. When the next chunk arrives, it is appended to this partial message and the parsing process starts again.

`ubx_msg()` is essentially a switch statement on the `CLASS` and `ID` fields of the message frame. Each message gets passed on to its specific handler. In the handler

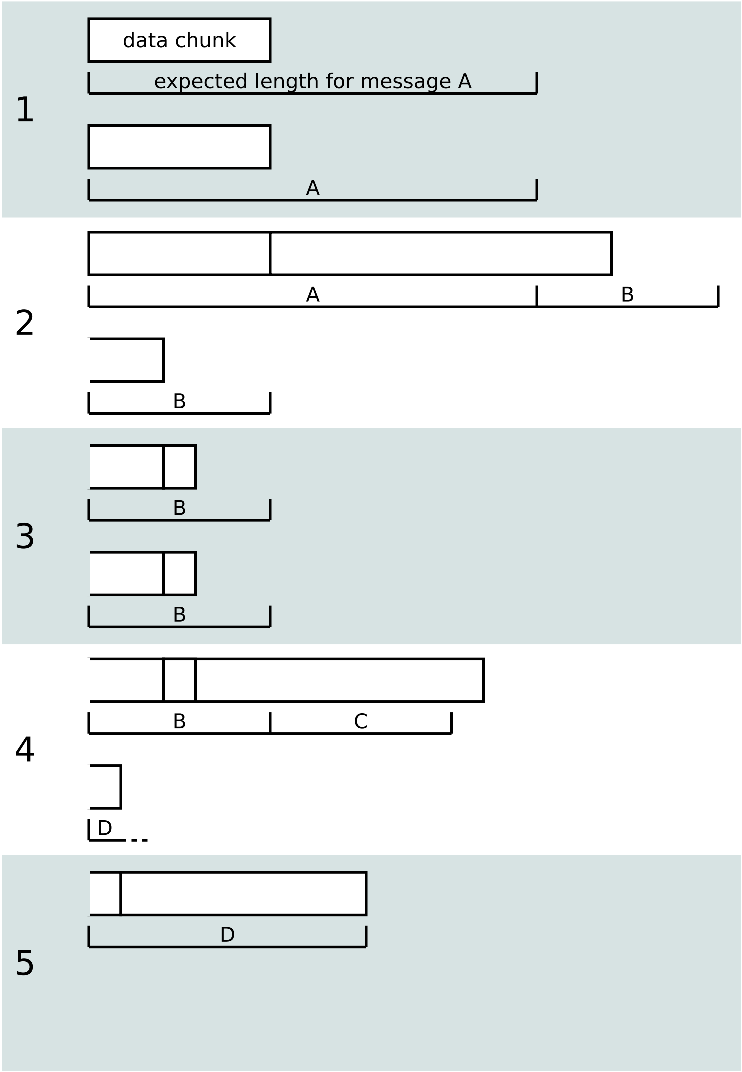


Figure 3.3 – Four messages A - D arriving in five chunks of different sizes. Each chunk corresponds to one call to receive. The graphic shows the reception state before and after processing the new chunk. The handler for message A is called in 2, for B and C in 4 and for D in 5. The first part of D is not long enough to determine the length of the message.

it gets copied to the location of a local struct that fits the message type and can then be comfortably accessed through the struct.

For `UTC` mode the message arrival time is important. Ideally, we want to use the arrival timestamp of the first chunk that contained part of the message. There are essentially two cases we have to handle to achieve this. First, when the message buffer is empty, we know that the next chunk starts a message and we can save the arrival time. Second, whenever we encounter a message `B` (complete or partial) after a complete message A has been parsed, we know that B arrived with the last chunk of A. This must be the current chunk, otherwise A would have been parsed already. So we can use the arrival time of the current chunk as the message arrival time for B.

In figure 3.3, message A fits case one and all other messages fit case two. Message B receives the timestamp from call 2 even though it is only handled two calls later in call 4. Messages C and D start in the same chunk and so they receive the same timestamp.

3.1.3 Receiver Setup

One of the main tasks during startup is proper setup of the receiver. By default the u-blox receivers send NMEA data on their UART and USB ports. First, we send two `CFG-PRT` messages to disable NMEA and enable UBX on each port. During the protocol switch messages can get lost, so we wait for two seconds before continuing with the receiver configuration. It would also be possible to wait for the `CFG-PRT` acknowledgment and resume configuration then, however, this proved slightly unreliable, since the `ACK-ACK` message may be lost too.

Once communication is established, we set the receivers dynamic model to stationary and set the configured masking angle for satellites in a `CFG-NAV5` message. We then configure the receivers timepulse to one pulse per second with `CFG-TP5` and specify the time mode parameters in a `CFG-TMODE2` message. This includes position for fixed mode or the end conditions for a survey-in.

The receiver documentation recommends disabling SBAS for timing applications. [4] On older receivers this can be accomplished with a simple `CFG-SBAS` message. This message is deprecated, so for newer receivers we poll the GNSS configuration with `CFG-GNSS`, switch off SBAS, and send it back. We always send both messages, since configuring it twice (on receivers that support both methods) is no problem and otherwise the unsupported message will simply have no effect.

The receiver is now configured for a timing application. By default however, it doesn't send any periodic messages, so we have to enable the ones we need. The driver uses `NAV-TIMEUTC` for utc times, `TIM-TM2` for timemarks and `NAV-TIMELS` for leap seconds.

3.2 Leap Seconds

There are several ways for an NTP instance to know about leap seconds. Overriding all others is the *leapfile*, a list of all announced leap seconds. If installed, NTP will parse it and check if a leap second is due at the end of the month. If no leapfile is present, secondary servers can get the information from their peers. If more than half of the survivors (see section 2.2.2.3) announce a leap second, NTP prepares for a leap second at the end of the month. Primary servers may also be able to get the leap second announcement from their reference clocks. In principal GNSS and some other time services broadcast leap second information, however, not all receivers make it available and not all drivers are able to parse it. [32] Both the Oncore driver and the new u-blox driver make use of leap second information in slightly different ways.

The Oncore driver itself has two methods of getting leap information, depending on the receiver. Some receivers provide detailed information about the upcoming leap second, in which case the driver can simply check if the date of is within the current month and arm the leap condition accordingly. Other receivers only provide the binary information of whether a leap second has been announced. As this may happen up to six months in advance, the driver can not simply assume the information is true for the current month. In practise all leap seconds have happened either at the end of December or June, so the driver assumes this to be the case for future leap seconds as well and only checks for leap seconds in those months.

For u-blox receivers the `NAV-TIMELS` message provides all the needed information. During receiver setup, we configure this message to be sent periodically at the largest possible interval of once per 255 navigation solutions or in our case once per 255 seconds. If the date is within 25 days of the current time, we can be certain that the leap second will be inserted at the end of the current month and arm the leap condition.

3.3 Configuration

Using this basic infrastructure, the driver supports multiple modes and configurations through its own configuration file. This is very similar to the way the Oncore driver can be configured. In fact the parsing code is an adapted version of the Oncore parsing code. However, there is a different set of keywords and the meaning of overlapping keywords can be different.

The `MODE` option, for example, is used to set one of four modes that are specific to the u-blox driver. For the simplest mode of operation, set this to `UTC`. The driver will now only use time messages (NAV-TIMEUTC) and the time of their arrival for synchronization. Because of its low accuracy this mode is not meant for productive use. However, when setting up a system, it can be useful to troubleshoot the serial connection and the PPS signal separately from each other, which is why we provide a mode that doesn't require a PPS signal.

The other modes are `PPS` for standard PPS operation, `MEASUREMENT` for PPS latency measurements and `TIMEMARK` for PPS echo operation. They will be explained in detail in later sections. The four modes are the result of combining the three synchronization options offered by the u-blox receivers, NAV-TIMEUTC messages (utc), PPS (pps) and external interrupts (echo). Figure 3.4 shows how their possible values relate to the modes.

utc	pps	echo	
0	0	0	no synchronization
1	0	0	`UTC`
0	1	0	PPS without second numbering
1	1	0	`PPS`
0	0	1	no PPS → no PPS echo
1	0	1	no PPS → no PPS echo
0	1	1	`TIMEMARK`
1	1	1	`MEASUREMENT`

Figure 3.4 – Theoretically possible modes of operation based on time synchronization methods. Unsupported combinations are grayed out.

Another configuration option that defines the general behaviour of the driver is `TMODE`. It can be used to control the u-blox receivers time mode (see section 2.5.2.1). Time mode can be turned completely off with `OFF`. For improved performance however, it should be set to `FIXED` in combination with a known antenna location or to `SURVEY` to perform a site survey. Both of those modes can be configured in detail with additional options.

For ultimate flexibility the two options `PREMSG` and `POSTMSG` can be used to send arbitrary messages to the receiver. The initial keyword should be followed by the message to be sent in hex format with spaces between bytes. The two header bytes and the checksum should not be included. The payload length fields are necessary, however, their value does not need to be correct. Listing 3.1 shows an example.

```
PREMSG 06 09 0D 00 FF FF 00 00 00 00 00 00 FF FF 00 00 03
```

Listing 3.1 – Example PREMSG option that resets the receiver configuration before the driver does its setup.

Messages are sent in the order in which they appear in the configuration file. The number of `PREMSG` statements is not limited but there may be no more than twenty `POSTMSG` lines. `PREMSG` messages are sent as the configuration file is read, i.e. before the driver and receiver are configured. This can be useful for resetting the receiver as shown in listing 3.1. `POSTMSG` messages on the other hand, are sent after receiver setup. An example use of this would be to select which GNSSs the receiver should use.

Overall the options shown in figure 3.5 are available.

Option	Explanation
MODE	Set one of the four modes UTC, PPS, MEASUREMENT, TIMEMARK.
TMODE	Configure time mode. Possible values: OFF, FIXED, SURVEY
X, Y, Z	Set ECEF-coordinates in cm.
MINDURATION & ACCLIMIT	Set the parameters of the site survey. (See section 2.5.2.1)
DELAY	Set the user configurable delay.
MASK	Set the satellite mask angle.
HARDPPS	Turn kernel discipline on or off.
TMEDGE	Use fallin or rising edge of PPS echo signal in timemark mode.
POSFILE	Save Survey-in result in this file and try to read coordinates from it in fixed time mode.
DATAFILE	Save PPS latency measurements to this file.
PREMSG, POSTMSG	Send arbitrary messages before and after receiver configuration.

Figure 3.5 – All configuration options supported by the u-blox reference clock driver.

3.4 Survey-In

If `TMODE` is set to `SURVEY`, the receiver is configured to perform a site survey. The minimal duration defaults to one hour and the accuracy limit to ten meters. We recommend explicitly setting those values to fit your use case.

As mentioned in section 3.1, we use the timer function to monitor the survey progress. The function polls the survey state once per second with a `TIM-SVIN` message, even if no survey has been started. The handler for the replies prints the received information to the log. It also checks if a survey is in progress. If not, i.e. it has finished or not been started, the handler disconnects the timer function, stopping further polling. At this point survey results, if any, are written to the file configured with `POSFILE` and the receiver automatically switches to fixed mode. [4]

3.5 PPS

3.5.1 Setup

An accurate known position means we will be able to receive high quality PPS pulses. Before we can access those, the PPSAPI needs to be set up. The behaviour of the u-blox driver and the Oncore driver are slightly different here.

Both drivers open the PPS file and read its current parameters. The idea is to then modify them as needed and write them back to the PPSAPI, which is what the u-blox driver does. The Oncore driver discards the old parameters and replaces them entirely. This is likely unintended behaviour and has been reported to the NTP developers along with a patch. At the time of this writing there has been no reaction.

We use this method to toggle assert timestamps and to switch the kernel discipline on or off, depending on the configuration.

Another difference is that the Oncore driver only switches parameters to true, whereas the u-blox driver also switches parameters off. The Oncore behaviour is slightly more respectful to other programs which might be using the API, however, it creates a usability problem: Once the kernel discipline has been enabled, there is no way to disable it through NTP. The only options are an external program or a system reboot. If the user is unaware of this behaviour, they might also believe they are running NTP without the kernel discipline, while in fact it remains switched on from a previous run.

3.5.2 Synchronization

During operation, we fetch PPS timestamps while handling NAV-TIMEUTC messages. Because pulses are dispatched on the receiver at the top of the second, the message time without sub-second offset gives us the receiver time. The PPS timestamp gives us local time. With both available, we can compute the offset between receiver and computer.

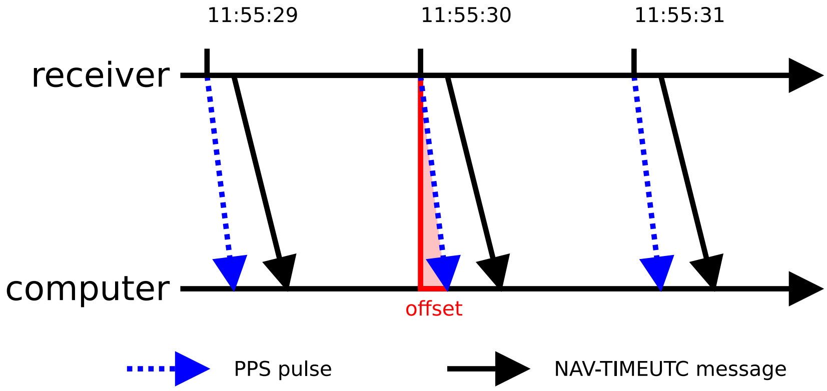


Figure 3.6 – Offset calculation between receiver and computer using a PPS pulse and utc time messages.

This is possible since we can rely on the receiver generating the message after the pulse, which means the latest PPS timestamp will be available for fetching once the message arrives, as shown in figure 3.6. Duplicate pulse timestamps are recognized and ignored in case a pulse gets lost for some reason. This prevents the faulty association shown in 3.7.

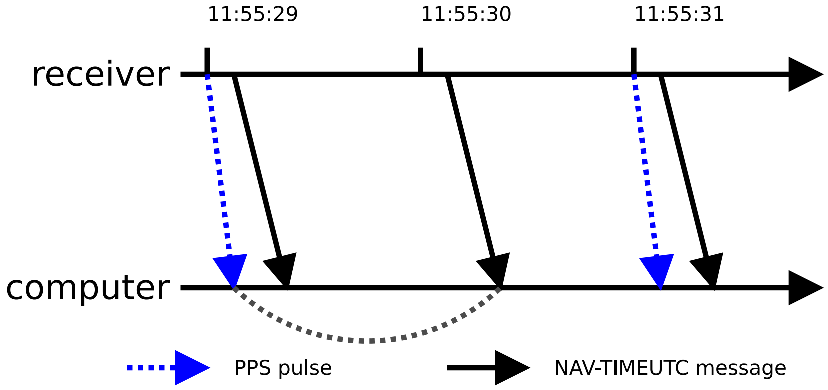


Figure 3.7 – False association (grey) in case of a lost PPS pulse. The driver will not use this pulse/message pair.

The driver also discards a pulse/message pair, if the message arrives more than half a second after the pulse. If this is the case, something has probably gone wrong and the timestamps should not be used (see figure 3.8).

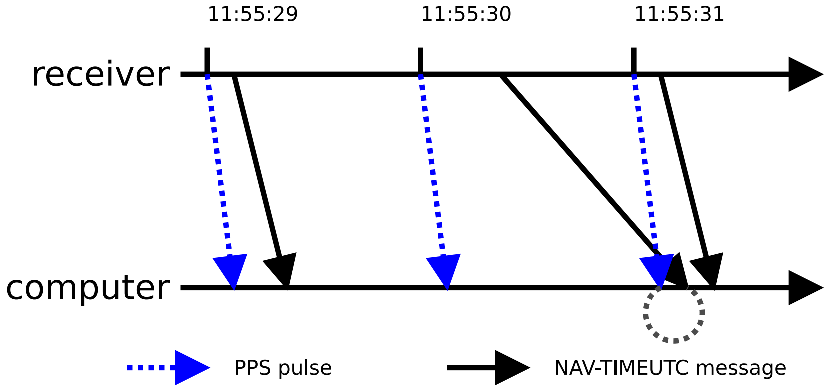


Figure 3.8 – False association (grey) in case of a delayed utc message. The driver will not use this pulse/message pair.

3.6 Timemark

Timemark mode also uses the PPSAPI. By setting the `PPS_ECHOASSERT` flag, it tells the API to generate an echo signal every time a PPS pulse is timestamped. The output of this echo signal is connected to the external interrupt line on the receiver. Every epoch, the receiver generates timestamps for the last rising and falling edges it sees on the external interrupt line and sends that information back in the `TIM-TM2` message. Figure 3.9 shows which flanks will be visible in `TIM-TM2` messages.

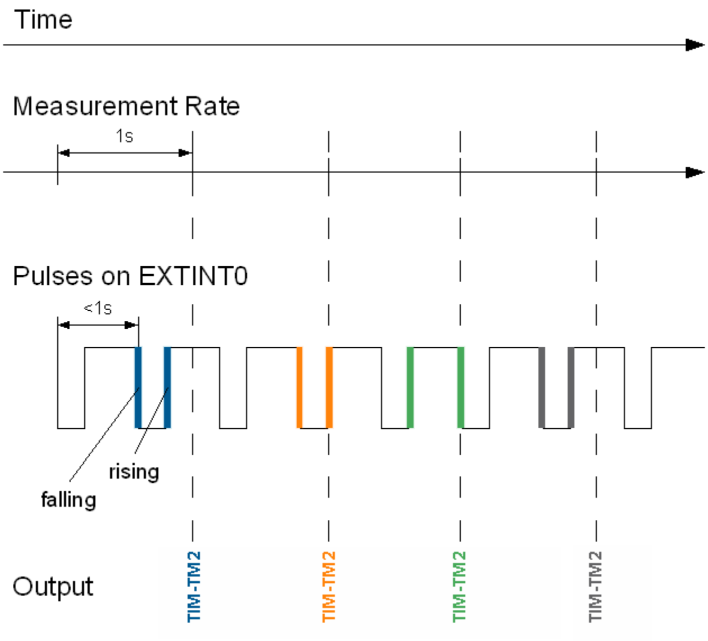


Figure 3.9 – Rising and falling edges and the `TIM-TM2` messages which will report their timestamps. Note that it is quite possible for edges to be lost, however, this occurs much less often when the frequency of the interrupt signal is the same as the measurement frequency (1Hz in this case). [4]

This gives us two timestamps, a local one from the PPS pulse and one on the receiver side from the echo pulse. As figure 3.10 shows, this allows us to synchronize the two clocks. It is important to note that the difference between these timestamps does not include local interrupt latencies. There is a delay between the PPS pulse arriving and the interrupt handler, which timestamps the pulse, being triggered. As this delay is quite variable, it increases jitter and can not be easily removed by calibration. [33]

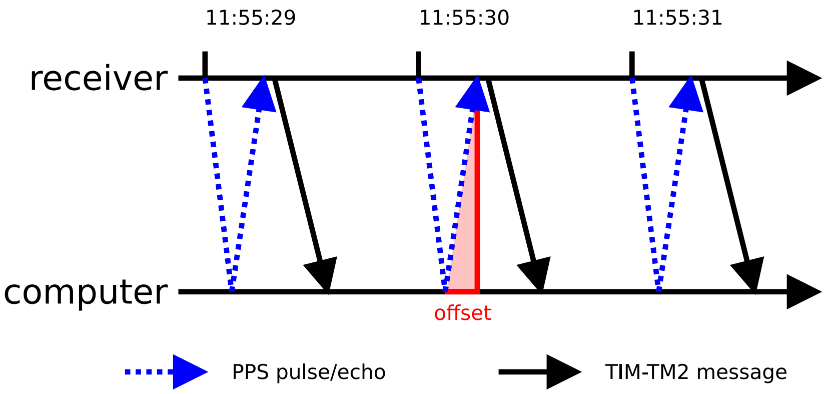


Figure 3.10 – Offset calculation in timemark mode. The local timestamp comes from the PPSAPI, the receiver timestamp is reported in a `TIM-TM2` message.

In timemark mode however, the timestamping for the PPS pulse and the generation of the echo pulse happen immediately after one another, both in the interrupt handler, with very little delay. [33] This means that both timestamps happen after the local interrupt delay, removing it from the measured offset. Instead the interrupt delay on the receiver side is now included in the offset, however this should be much lower and more predictable.

Strictly speaking, the PPS pulse isn't even necessary for this process. One could simply generate a pulse on the external interrupt line once per second at a known time. However having the echo signal triggered by a PPS pulse instead of a timer function or something similar, has several advantages. Having a common API makes the implementation much more portable, without it, we would have to write code for every different piece of hardware which can be used to send the echo signal (GPIO, parallel port,...). Another advantage of this method is that the timestamping and the signal generation happen inside the kernel, in an interrupt context. This means that the time between those two events is not increased by context switches and has a very low chance of being preempted by other interrupts.

It is important to correctly associate the PPS timestamps and the `TIM-TM2` messages when using this method. In figure 3.10 we assume that the echo signal reaches the receiver before the next `TIM-TM2` message is sent. This is not always the case. Testing showed that for stretches of time the signal would hit after the receiver sent the message.

Figure 3.11 shows the switch from one situation to the other. The `TIM-TM2` message at 30 seconds is missing because the preceding measurement period did not contain an echo signal (because it hits too late). From 31 seconds onwards the handler for the message would falsely associate the latest PPS timestamp with the echo timestamp from one second earlier. To prevent this, the driver always keeps the last two PPS timestamps and compares both of them to the current timemark and only uses the one with the lower difference.

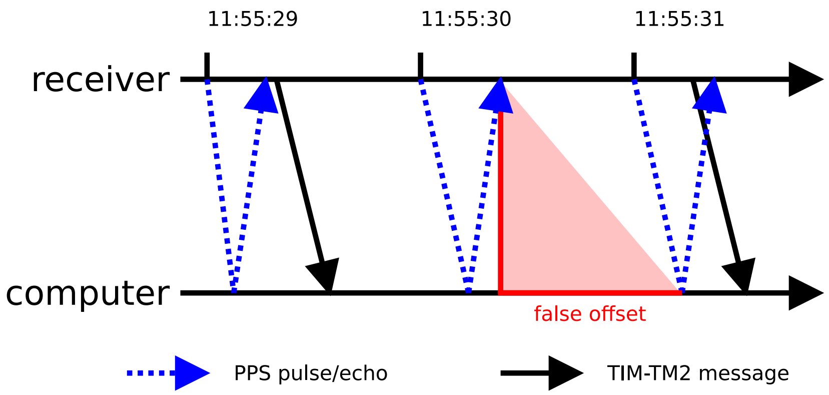


Figure 3.11 – If the receiver tries to send the `TIM-TM2` message to early it doesn't get sent, because there is no new timestamp to report (second 30). This leads to an offset of almost 1s unless precautions are taken.

This leads to the PPS/echo pair from second 30 being lost. Additionally when the situation switches back again another pair will be lost. This does not pose a problem, since we pass a buffer of 8 or more samples to NTP. One or two samples missing from that buffer should not pose a problem.

3.6.1 PPS Latency

Additionally to using the timemark functionality directly for synchronization as described above, it can also be used to measure what we call *PPS latency*. In `MEASUREMENT` mode, we use the original PPS pulse for synchronization and use its echo signal to get an idea of the time it took for the PPS pulse to be handled as an interrupt.

Assuming there is no delay between the generation of the echo signal and its timestamping on the receiver side, the echo timestamp now gives us the time between PPS pulse generation and PPS interrupt handling. Given that the propagation delay of the PPS pulse from receiver to computer should be mostly constant, this PPS latency measurement can give us an idea of how interrupt latencies influence the accuracy and stability of synchronization via PPS. This idea is described in further detail in [33].

3.6.2 Kernel Modifications

All of this heavily relies on the PPSAPI being implemented correctly and with the echo feature included. In practice, there were some hurdles to overcome, depending on the operating system. The patches resulting from this are, as of the time of this writing, not yet included in the respective upstream projects.

As described in chapter 4, one of the development and evaluation platforms was a Raspberry Pi running Linux [34]. The kernel reported the PPS echo feature as being implemented for the Raspberry Pi's GPIO pins. This was not actually the case, it only wrote a line to the system log instead of actually generating a signal. We implemented this by changing the interrupt handler to a threaded interrupt handler. After raising a GPIO pin in the interrupt handler, the subsequent thread sleeps for 100ms and lowers the pin again. This results in an echo pulse of roughly 100ms length.

The second evaluation platform runs FreeBSD and uses the parallel port as a PPS interface. The FreeBSD kernel actually implements an echo pulse, however, the signal is both raised and lowered inside the interrupt handler. This results in an echo pulse that is too short for the u-blox receiver to recognize. To fix this we added a timeout function that lowers the echo signal. The resulting echo pulse is roughly 200ms long.

3.7 Monitoring

To be able to compare the performance of the drivers and receivers, we need to extract some performance metrics. NTP can be configured to generate a *clockstats* file and a *loopstats* file, among others, for this purpose. [35]

NTP writes one line to the loopstats file every time it updates the system clock. Along with other information, each line contains the clock offset, the jitter and the duration of the respective feedback loop iteration. Figure 3.12 shows an example line along with field explanations from the NTP documentation. [35]

```
50935 75440.031 0.000006019 13.778 0.000351733 0.013380 6
```

#	Example	Units	Description
1	50935	MJD	Modified Julian Date
2	75440.031	s	time past midnight
3	0.000006019	s	clock offset
4	13.778	PPM	frequency offset
5	0.000351733	s	RMS jitter
6	0.013380	PPM	RMS frequency jitter (aka wander)
7	6	log_2s	clock discipline loop time constant

Figure 3.12 – An example of a line in the loopstats file with individual fields explained. [35]

The clockstats file is reference clock driver dependant. Both the Oncore and the u-blox driver output some general logging information, mostly at the beginning. The Oncore driver then writes a line of detailed information about every clock update it receives. This information allows us to compute the offset for each update. The u-blox driver simply prints every offset it generates to the file.

The u-blox driver has one additional method of gathering data. When in `MEASUREMENT` mode, a *datafile* can be specified, which will contain the PPS latencies measured.

Chapter 4

Evaluation

With a basic grasp of the technologies involved and an understanding of the implementation, we can now take a look at the evaluation results. A good way to judge a clock is to look at its stability. In a stable clock, phase and frequency offsets can be removed through calibration. One way to measure this would be to take successive offsets between our clock (the GNSS disciplined VFO) and a stable reference clock and look at the second differences. In the context of this thesis no such reference clock was available, but NTP assesses stability by calculating the jitter as described in section 2.2.2.3. We will mostly look at jitter as a metric for evaluating the different setups and modes. But we will also use measurement mode to investigate PPS latency.

4.1 Comparison with Motorola Oncore

4.1.1 Setup

Since the Oncore reference clock driver served as a guide for the u-blox driver, we will first compare the performance of the two drivers in classic PPS mode. The drivers and their corresponding receivers will be compared as a unit, we will not differentiate between effects caused by driver implementation and effects caused by superior or inferior hardware. However, if the two receivers show comparable performance, differences are likely going to be the result of hardware differences, as at this point there are no newly developed driver techniques at play.

For the measurements in this section we used a M12M-T Oncore timing receiver produced by i-Lotus and a u-blox NEO-M8T, which is also a timing receiver. Both were connected to a desktop PC running FreeBSD. The PC contains a 64 bit Intel Core2 CPU with four cores running at 2.83GHz. On both receivers we used development boards for easier access to pins and connected them to the serial interface. The PPS pulse was connected through a parallel port card, which was plugged into a PCI slot.

Since parallel bus connections use a 5V signal level, a level converter was needed to connect the 3.3V PPS signals from the receivers. The MOS-FET based level converter introduces a slight signal delay, details about this can be found in [36] and in section 4.2.1.2. Figure 4.1 shows the general setup. It does not include the fact, that the computer was connected to a network for remote access.

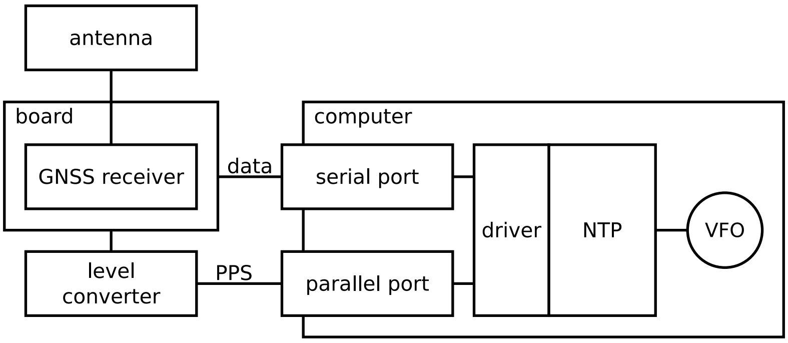


Figure 4.1 – Experimental setup showing how the GNSS signal eventually reaches and disciplines the VFO.

The antenna was located on top of the twelve story faculty building, with clear view of the surrounding skies. The position of the antenna was determined via site survey on the u-blox receiver with an accuracy limit of 5cm.

The FreeBSD version 11.1 was modified as explained in section 3.6.2 and configured to allow PPS with the configuration file in listing 4.1. NTP version 4.2.8p10 was installed through the FreeBSD ports system with all FreeBSD-specific patches applied first and our u-blox patches applied second.

```
include GENERIC
ident PPSKERNEL

options PPS_SYNC

device pps
```

Listing 4.1 – FreeBSD kernel configuration

4.1.2 Free Running

Before actually comparing the receivers, we can take a look at their general behaviour within this setup, without actually disciplining the computer clock.

NTP comes with the `pps-api` utility. If executed with a PPS device as a parameter, the program will output all PPSAPI assert timestamps along with some more information. This allows us to compute an offset from the top of the second. With these offsets we can generate an Allan Deviation plot that characterizes the interplay of the receiver clock and the computer clock.

Doing this over 24h, once for each receiver, gives the curves in figure 4.2. The y-value tells us how stable the two clocks (local and receiver) behave in relation to one another, with lower values corresponding to less expected drift. The stability prediction depends on how many samples are averaged. For small cluster sizes short term effects dominate, bigger cluster sizes depend on longer term variations.

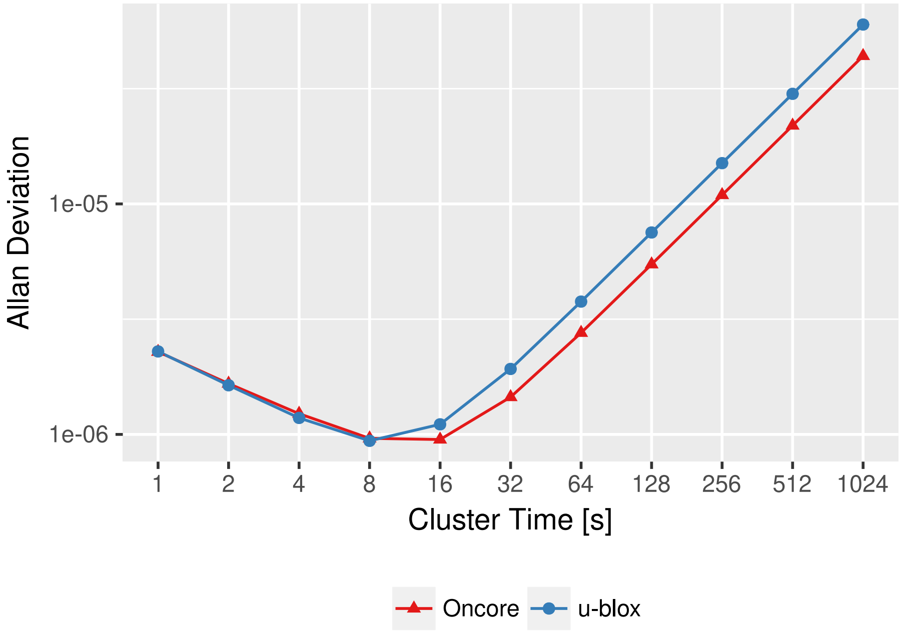


Figure 4.2 – Allan deviation of both receivers running freely without disciplining the VFO.

While the graphs look very similar for lower cluster sizes, the u-blox receiver shows slightly higher values on the upwards slope. As mentioned above, this area is dominated by longer term effects. It could, for example, be the result of environmental influences like temperature on the two quartz oscillators.

For us the most interesting information we can gather from these graphs is the minimum. It tells us how many samples we have to average in order to maximize stability. We can set this value as the poll interval in NTP to minimize jitter. For the Oncore receiver the minimum is at $x = 2^4$ and for the u-blox receiver it is at $x = 2^3$.

4.1.3 Disciplined

4.1.3.1 Setup

While the free running measurements give us some insight into the receiver behaviour, we are more interested in their performance together with NTP and with the reference clock drivers. To evaluate this, we ran `ntpd` with the following configuration file (listing 4.2):

```
server 127.127.47.0 minpoll 3 maxpoll 3 true prefer #ubx
server 127.127.30.0 minpoll 4 maxpoll 4 true prefer #oncore

driftfile /home/user/driftfile
statsdir /home/user/stats/
filegen loopstats type pid
filegen clockstats type pid
```

Listing 4.2 – NTP configuration

Only one of the first two lines was activated at a time to switch between receivers. The lower part of the configuration tells NTP to generate loopstats and clockstats in a directory called `stats` and to make use of a driftfile. In this file NTP saves the average clock drift it observes and uses this information to remove the drift from the VFO. For the following measurements, a driftfile was generated over seven days. This master driftfile was then used as a starting point in subsequent runs.

The driver-specific configuration files are shown in listing 4.3 for the Oncore driver and listing 4.4 for the u-blox driver.

```
MODE 1
LAT 49.573826
LONG 11.027068
HT 389.72 M
SHMEM /var/cache/ntp/oncore
```

Listing 4.3 – Oncore driver configuration

The `SHMEM` line is needed for the Oncore driver to function. Apart from that we only set the mode and position. Mode 1 is described in the driver code as “NO RESET, Read Position, delays from data file, lock it in, go to 0D mode.” The u-blox driver uses ECEF coordinates. The geodetic coordinates were calculated with the method by Heikkinen as presented in [37]. The original coordinates are shown in listing 4.5.

Most of the u-blox configuration should be self-explanatory. We use the falling edge timemark because the parallel port line used for echo signals is inverted. The

```
MODE PPS
TMEDGE FALLING
HARDPPS OFF
TMODE FIXED
MINDURATION 3600
ACCLIMIT 50
POSFILE /home/user/posfile
DATAFILE /home/user/datafile
```

Listing 4.4 – u-blox driver configuration

file referenced in `POSFILE` is shown in listing 4.5. The `DATAFILE` is not actually used in PPS mode, but the configuration file is later reused with measurement mode.

```
X 406779785
Y 79269430
Z 483248203
```

Listing 4.5 – u-blox `POSFILE`

4.1.3.2 Results

Figure 4.3 shows the offsets logged by NTP over 24h, once for the u-blox driver and once for the Oncore driver. The first hour ist cut off for better scaling. The graph shows how both receivers approach an offset of zero and then swing around it. Whether the initial offset is positive or negative depends on the state of the VFO before NTP is started and is not indicative of receiver or driver behaviour. The graph shows that the receivers settle in at a comparable range, however, the amplitude and amount of fluctuation is hard to compare on this graph.

Figure 4.4 shows the jitter for the same dataset and duration. Again, we can see comparable behaviour. The values concentrate at the lower end of the spectrum with some spikes. It does not look like there are long-periodic changes to the jitter. While the overall structure is very similar, the graph suggests that the u-blox receiver does exhibit a slightly lower average jitter.

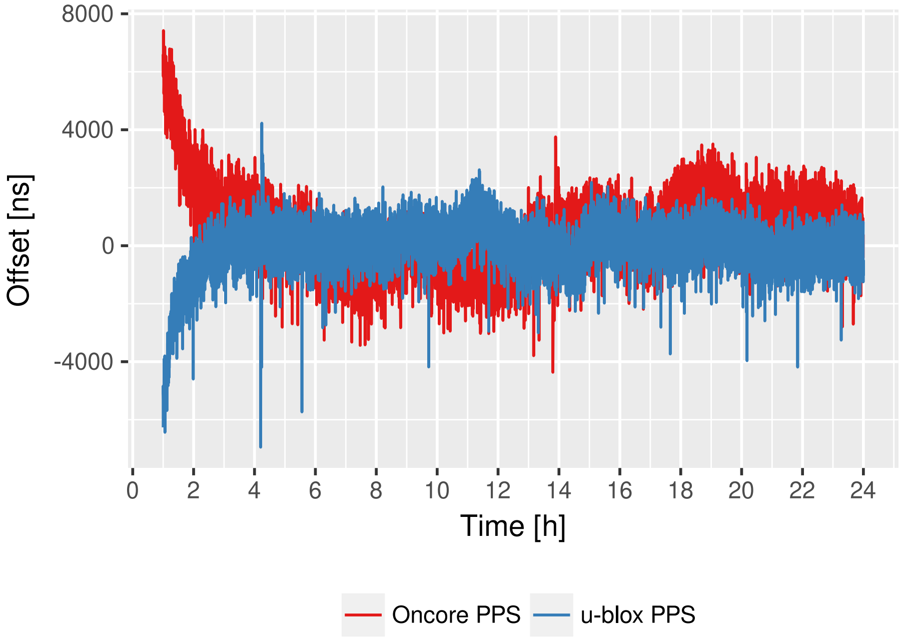


Figure 4.3 – Offsets of a 24h measurement as reported in loopstats over time.

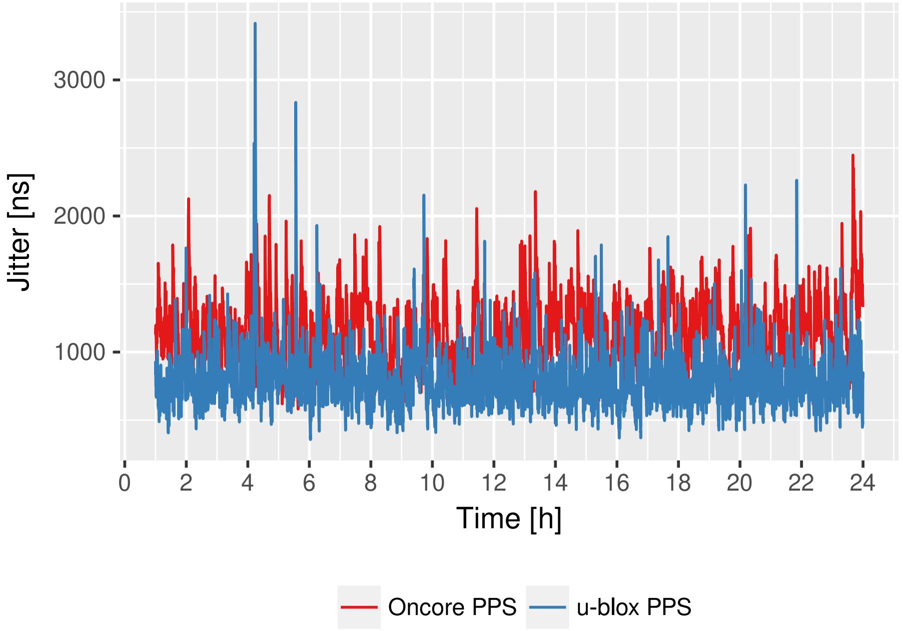


Figure 4.4 – Jitter of a 24h measurement as reported in loopstats over time.

Calculating the mean, we can see that with an average of 804ns the u-blox receiver did show less jitter than the Oncore receiver with 1141ns. Figure 4.5 shows the results, with one sample standard deviation marked in each direction.

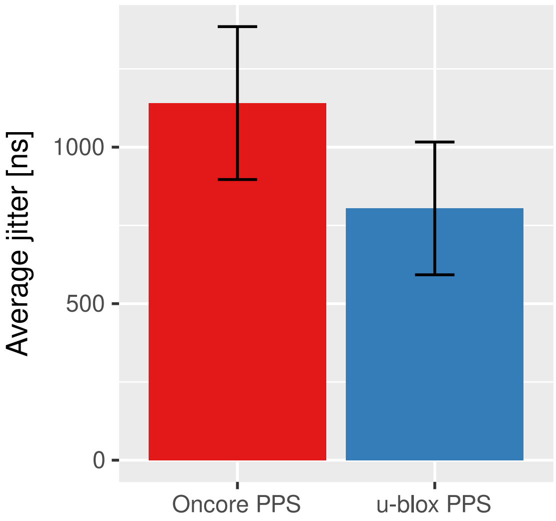


Figure 4.5 – Average jitter of the receivers compared. The new u-blox driver/receiver combination performs about 300ns better than its counterpart.

At this point no novel techniques have been used. This suggests that the base part of the implementation, which mostly emulates the behaviour of the Oncore driver, performs as expected. When looking at jitter as a performance metric, the new driver/receiver combination can even perform better than its counterpart.

4.1.3.3 Hardpps

We also performed the above measurement with the Oncore receiver and the kernel discipline enabled. Because this lead to far worse performance (see figure 4.6), we did not employ hardpps in any of the other measurements. The reasons for the high jitter are not clear and were not investigated further. It is possible, that the setup can be configured differently to improve performance, but we chose to focus on the NTP implementation of the discipline.

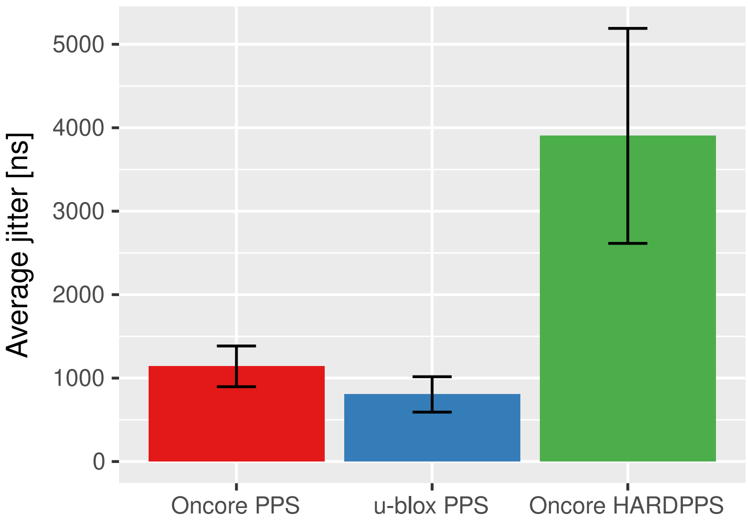


Figure 4.6 – Average jitter of the Oncore receiver with the `HARDPPS` configuration option set, compared to the results from above.

4.1.4 Timemark Mode

We have shown that the u-blox driver performs well in PPS mode. However, its most distinguishing feature is timemark mode. To compare it with PPS mode, we ran the same 24h szenario as above, with the u-blox mode changed. Looking at the offset in figure 4.7, we can clearly see that the green timemark offsets form a narrower band.

Figure 4.8 confirms, that the timemark jitter lies consistently below PPS jitter with only some of the higher peaks reaching worse values than above average PPS samples. We can also see that none of the peaks are as high above their baseline as some of the peaks for Oncore ore u-blox PPS.

Further analyzing the data, we see that with an average of 222ns, timemark mode showed less than a third of the jitter seen with PPS mode on the same receiver in this scenario. The difference is even larger compared to the Oncore receiver, which exhibits about five times the as much jitter. Figure 4.9 shows a comparison of the average jitter for all three receiver/mode combinations. The improvement is likely the result of removing local interrupt latency from the offset calculation since this is the most variable latency factor.

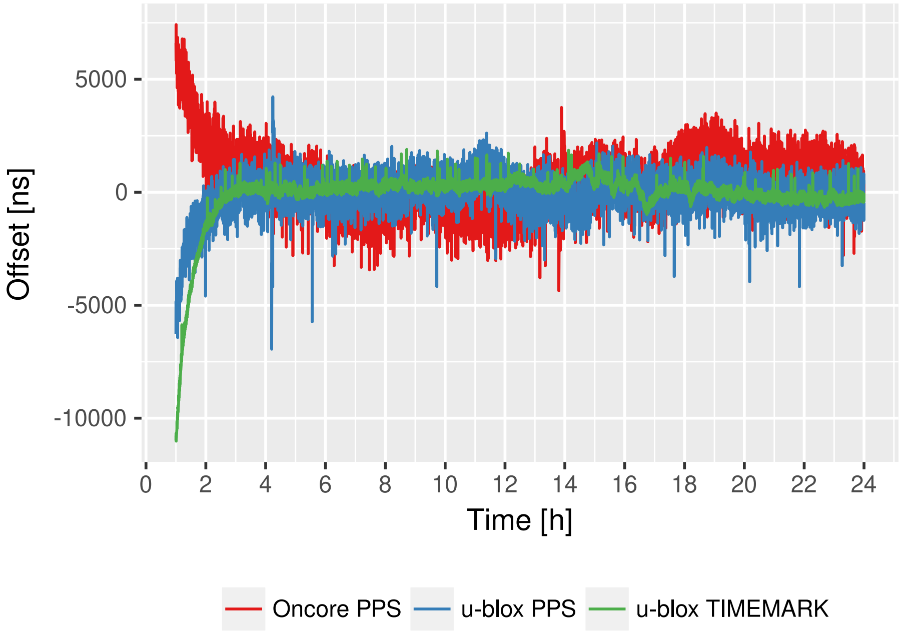


Figure 4.7 – Offsets in timemark mode (green, narrow band) compared to PPS.

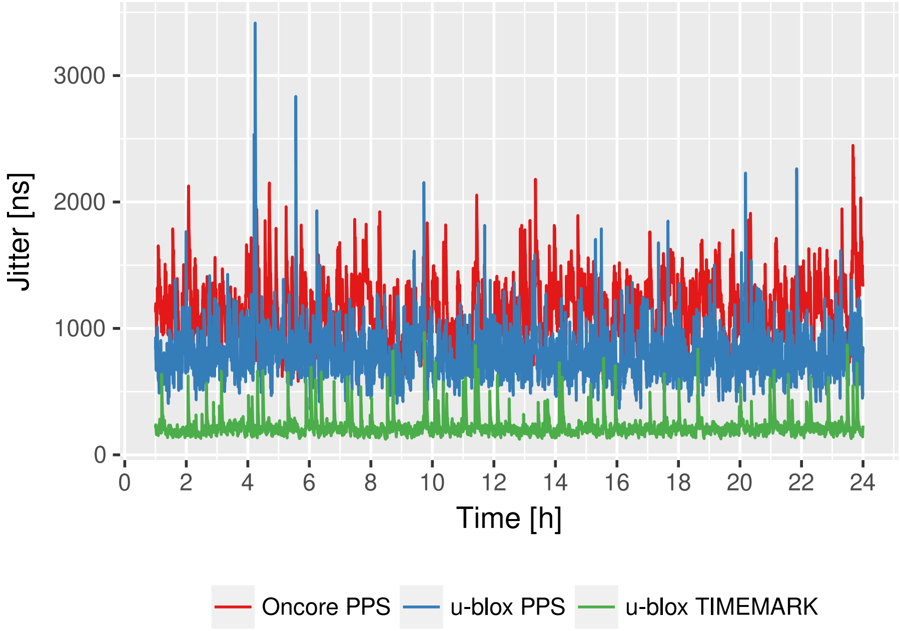


Figure 4.8 – Jitter in timemark mode (green, bottom) compared to PPS mode.

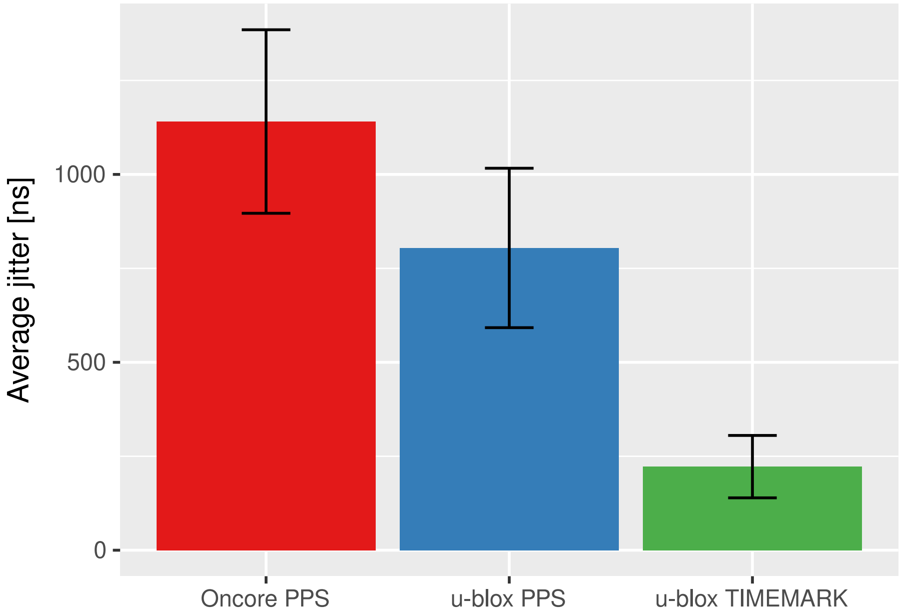


Figure 4.9 – Reduction in average jitter with timemark mode compared to the conventional PPS mode.

4.2 PPS Latency

4.2.1 FreeBSD

The previous section showed the effects of removing interrupt latency, however, we can also use the new driver to measure PPS latency which can be seen as an approximation of interrupt latency. Section 3.6.1 describes how this works and how PPS latency relates to interrupt latency.

To test this new measurement infrastructure in practise, we started by collecting 24h worth of PPS latency data. The general setup was the same as the one described section 4.1.1.

Figure 4.10 shows only 12h of the collected data to make certain effects more visible. Two features are most noticeable: There are two main bands of latencies with a more sparse area in between. And there are two vertical lines at around 14:00 and 15:15. The first 12h showed no similar artefacts.

To make the band effect more visible, we can display the full dataset as a histogram (figure 4.11). The majority of PPS pulses take between 13 and 20μs to be processed. This is followed by a smaller peak from 20 to 23μs, followed by a small number of higher latencies of up to 44μs.

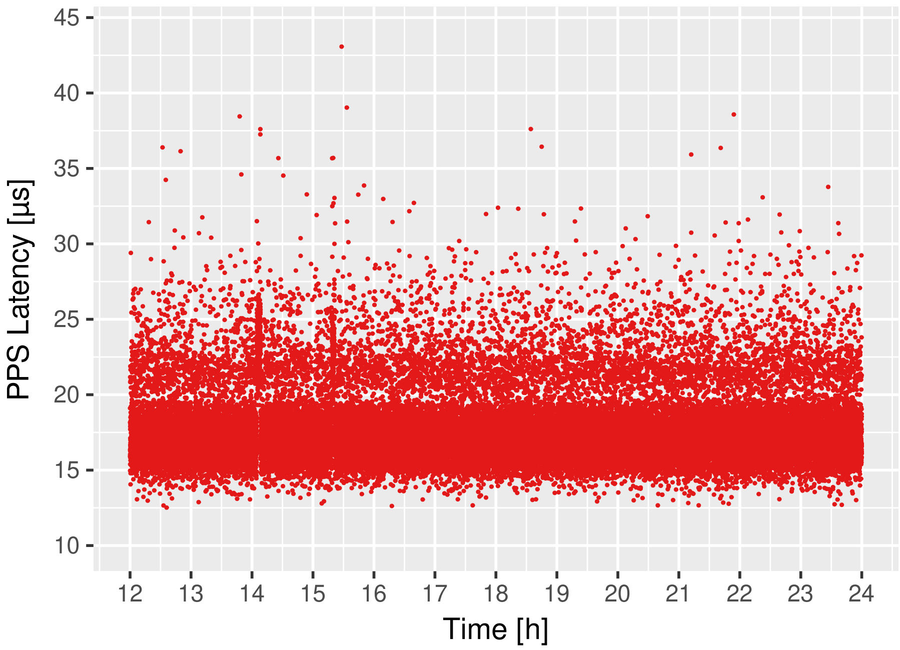


Figure 4.10 – PPS latency measured on the test platform.

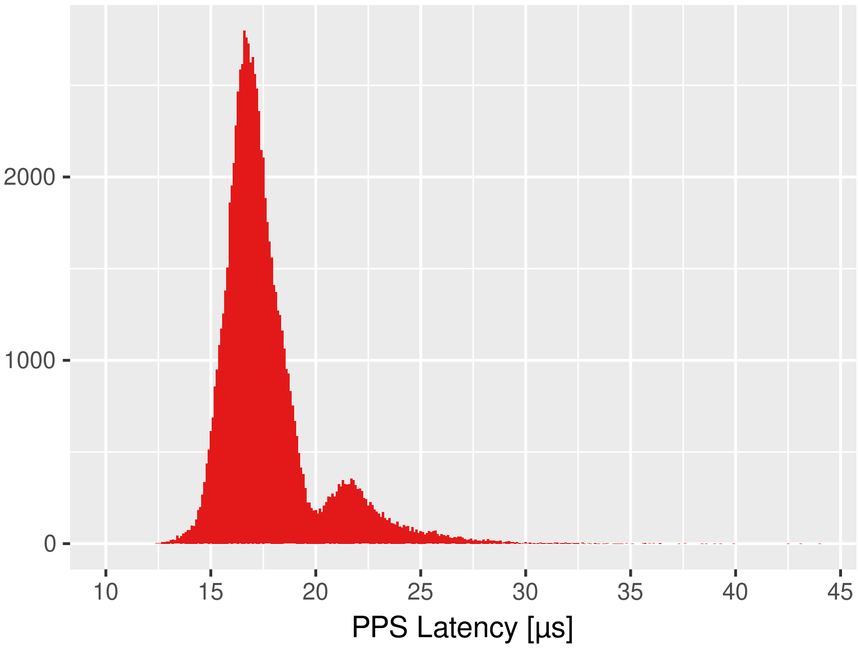


Figure 4.11 – Distribution of PPS latencies over 24h with bins of 100ns.

4.2.1.1 Interrupt Load

If we zoom in onto the first vertical artefact, we obtain figure 4.12. We can see that for about 200s the bulk of PPS latencies is shifted upwards by about 5μs.

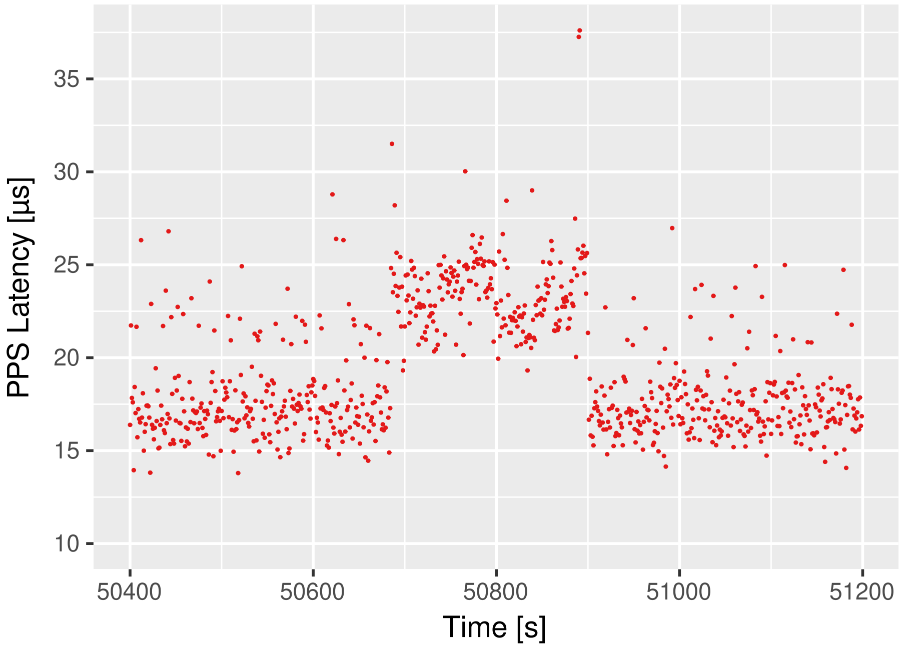


Figure 4.12 – The data from figure 4.10 zoomed in on the first artefact.

We were able to produce a similar effect by artificially causing a high interrupt load for a number of seconds (see figure 4.13). However, these latencies were measured on a slightly different setup than the above. On the experimental setup used for results shown here, we were not able to reproduce the effect. The interrupts were generated by running `ping -f <remote host>` ("flood ping") on the NTP server with a different host in the local network as a target.

These results are inconclusive, especially since they were not reproducible. Simply generating a large number of interrupts does not seem to automatically result in the observed pattern. However, figure 4.13 does indicate that interrupt load is one of the ingredients. Further experiments would need to be done to determine the exact cause of such spikes in PPS latency.

4.2.1.2 Level Converters

As mentioned in section 4.1.1, the level converter used in the experimental setup is MOS-FET based. [36] Towards the end of the thesis we were able to compare the PPS latency of the MOS-FET based level converter with that of a triple Schmitt-trigger

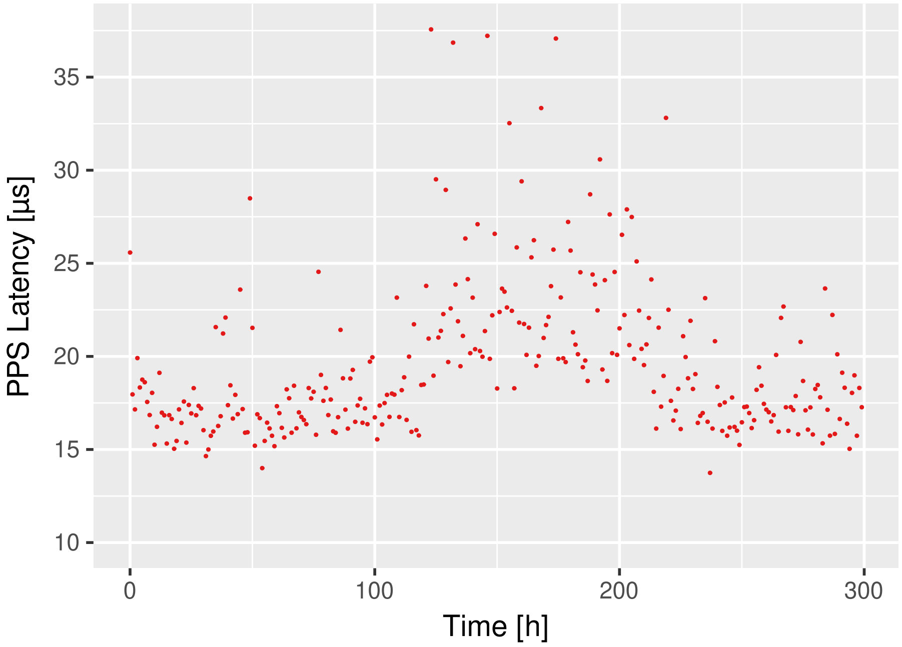


Figure 4.13 – An artificially generated latency spike similar to that from figure 4.12.

buffer. According to its specification, the delay introduced by the triple Schmitt-trigger buffer is below 11ns. [38] This amount of delay is negligible compared to the tens of microseconds of PPS latency measured above. The goal of comparing these two level converters was to see if the MOS-FET based converter introduces a more significant amount of delay.

Figure 4.14 compares the two level converters in the same setup over 24h. The two converters show roughly the same distribution of latencies. This suggests, that a MOS-FET based level converter was a valid choice for the experimental setup in this thesis.

4.2.2 CPU Affinity

The results from the section above give us a baseline of PPS latency on the experimental setup. This allows us to investigate possible ways of influencing it, which might give insight into how to optimize PPS performance. FreeBSD offers the possibility of binding interrupt numbers to CPUs with the `cpuset` command. Theoretically, this enables us to isolate PPS interrupts on a dedicated CPU core and measure how this impacts PPS latency.

We used the command `vmstat -i` to find out which interrupts are occuring on the computer. Listing 4.6 shows an example output of this command.

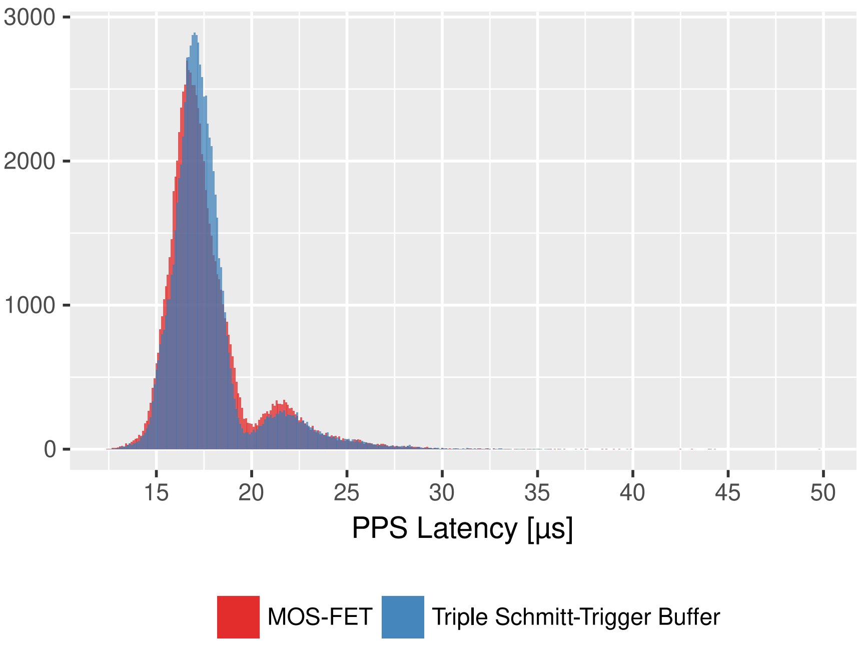


Figure 4.14 – TODO

```
interrupt                          total       rate
irq1: atkbd0                         257          0
irq4: uart0                     27484569        323
irq6: fdc0                             9          0
irq18: atapci0 ppc*                84849          1
irq20: uhci0 uhci3+               683885          8
irq256: hpet0:t0                 1707786         20
irq257: hpet0:t1                 6566505         77
irq258: hpet0:t2                 3366810         40
irq259: hpet0:t3                 3322284         39
irq264: em0                       152616          2
irq265: hdac0                         56          0
irq268: ahci0:ch0                 153583          2
irq269: ahci0:ch1                     29          0
Total                           43523238        512
```

Listing 4.6 – Example output of `vmstat -i`

We can see that the PPS interrupt number is 18, which is the only PCI interrupt and has a rate of one. The output also shows that the largest producer of interrupts is the UART interface, which handles the serial communication with the receiver. Since this serial connection is required for the driver to function correctly, there is no way to remove these interrupts but to bind them to a different CPU core.

Using `cpuset -l 0 -x 18` irq18 was assigned to core 0. Except for irq256 and irq269 we were able to bind all other interrupts to a different core. The reason for irq269 is not clear, but since it produces a low number of interrupts it should not have a big impact. Irq256 is a timer interrupt specific to core 0. It can be moved but this produces error messages and it is not clear what side effects this has, so it was left on core 0.

Using these settings we ran another 24h PPS latency measurement. Figure 4.15 shows the results along with the baseline from above. We can see that the main peak is shifted somewhat to the left and more compact. However, the amount of latencies outside that core peak seems to have increased and is not concentrated as compactly as the second baseline peak.

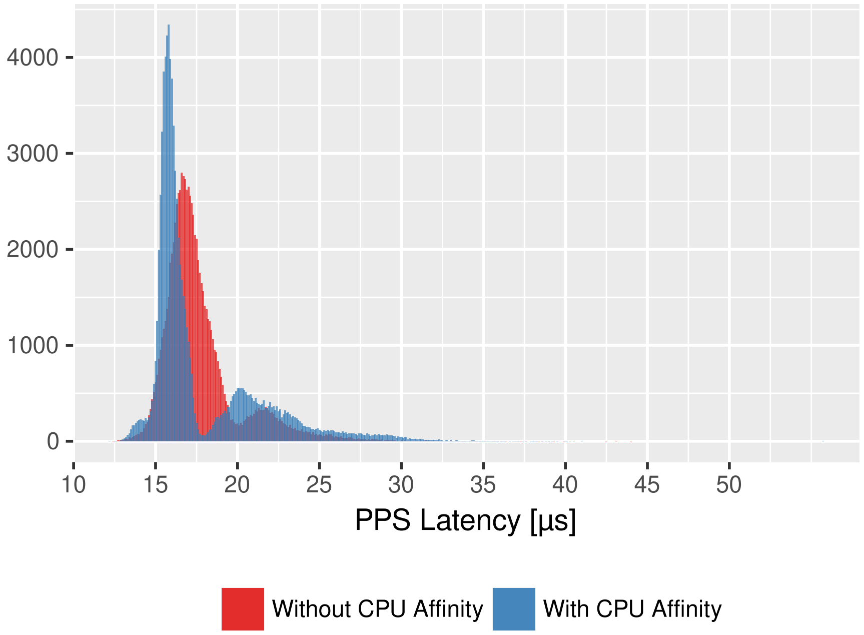


Figure 4.15 – Distribution of PPS latencies with CPU affinity enabled compared to the distribution from figure 4.11.

This ambivalent result does not imply a prediction as to whether this would lead to better PPS synchronization. To get a clearer picture we can look at jitter. Figure 4.16 shows how the jitter from this measurement run compares to previous jitter measurements from this thesis. It is immediately clear that isolating the interrupt as described above worsened performance. There might be other ways of using CPU affinity to achieve better results. However, judging from this experiment, this does not seem to be a promising path.

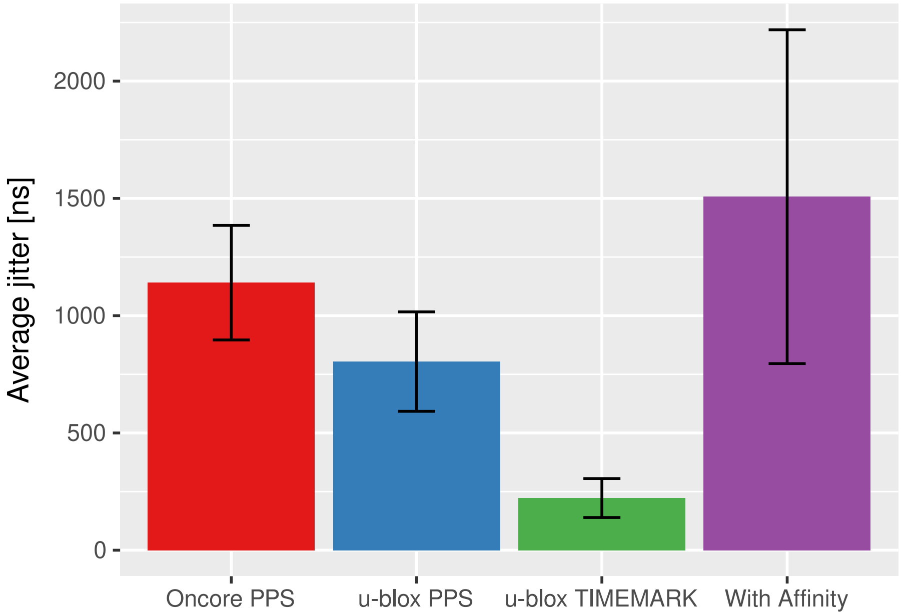


Figure 4.16 – Using CPU affinity results in high jitter.

4.2.3 CPU Load

Another interesting factor is CPU load. We can gauge its influence on PPS latency by running infinite loops during a measurement. For this we used the simple shell script in listing 4.7. The script was executed four times which resulted in nearly 100% CPU load on all four cores for the duration of the measurement.

```
#!/bin/sh

while true; do :; done;
```

Listing 4.7 – A simple infinite loop

The PPS latencies observed in this scenario are shown in figure 4.17 next to the latencies on an idle CPU. We can see that overall the busy CPU shows lower latencies. This somewhat unexpected result might be explained by powersaving features of modern CPUs and operating systems. Section 4.3 sheds some further light on this.

Given that the distribution is also more compact, we can expect PPS operation to also exhibit lower jitter in this situation. Figure 4.18 shows exactly that. The average jitter is much closer to what can be achieved with timemark mode. However, the variance in the data, visible here as the standard deviation, is much higher.

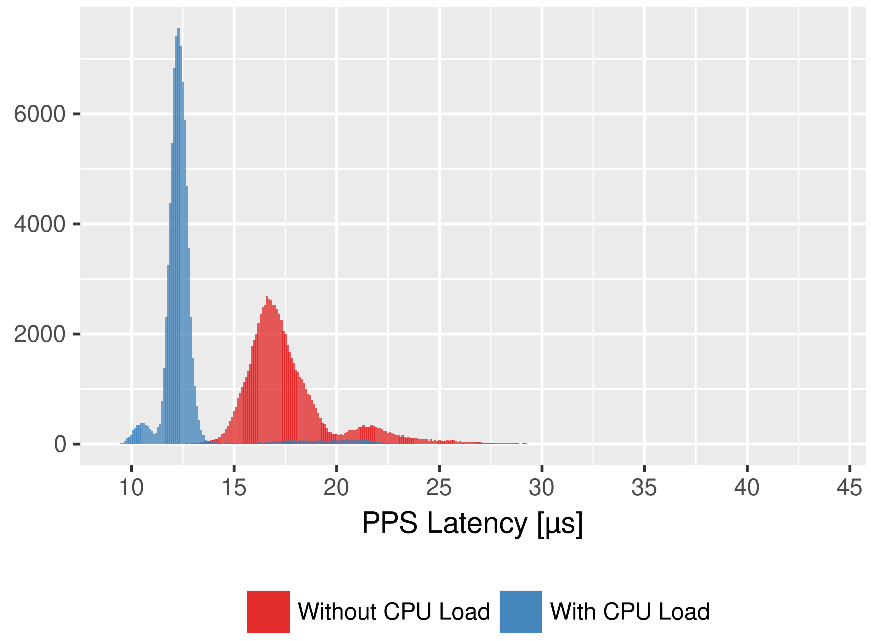


Figure 4.17 – Distribution of PPS latency with high CPU load compared to figure 4.11.

This, paired with the fact that this method wastes a lot of processing power compared to other modes of operation, makes timemark mode a more viable option. Although it must be said, that it might be possible to configure the CPU in a way that no actual processing is needed to achieve the improved performance. Section 4.3 shows such an attempt, albeit on a different system.

4.3 Windowsill Measurements

One of the goals of this thesis was to make accurate timekeeping and timekeeping experimentation accessible in a low-cost environment. In this section we will show how the new reference clock driver can be used to gather new insights in a very minimal setup.

The host platform is a Raspberry Pi 2 Model B V1.1 running Raspbian 4.9.80 with the patch from section 3.6.2 applied. The u-blox receiver is connected via the GPIO pins and its antenna is fixed to a windowsill in a residential area with limited view of the sky.

We know from section 4.2.3 that higher CPU load leads to better PPS performance. This is possibly due to dynamic frequency scaling. To check this claim further we can use the *cpufreq* interface on Linux to change the CPU scaling governor. If it has

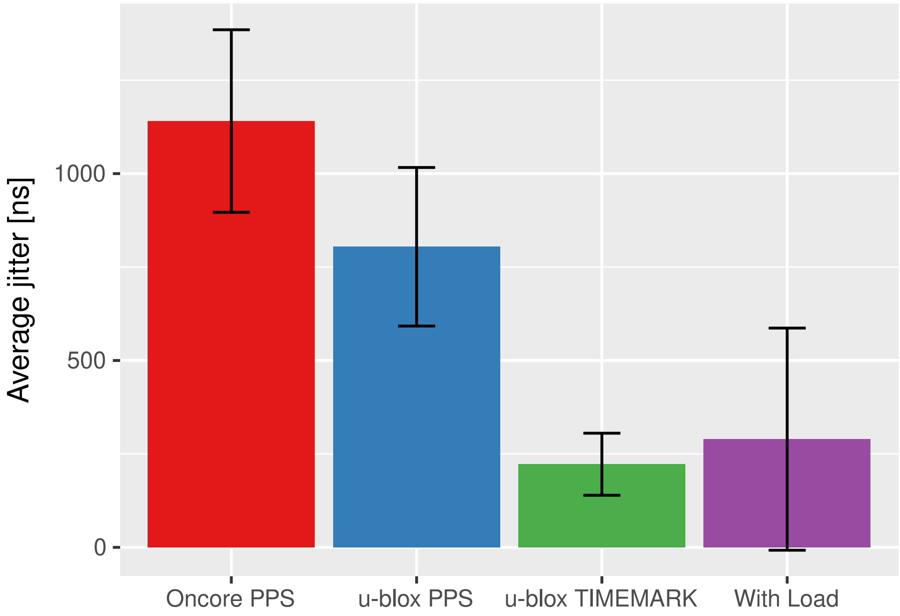


Figure 4.18 – Average Jitter of u-blox PPS mode with high CPU load compared to previous results.

any influence on interrupt latency we should also see this change in PPS latency measured with the u-blox driver.

By default Raspbian uses the *ondemand* governor, which dynamically changes the CPU frequency depending on load. Among others, there are also the static governors *powersave* and *performance*, which set the frequency to the lowest respectively highest one possible. If our assumption is correct, the performance governor should lead to lower PPS latency with the other two exhibiting similar values.

Figure 4.19 shows the three governors in use, each for 90s. The first third is the default ondemand governor, followed by performance, followed by powersave. A clear reduction in PPS latency is visible for the duration during which the performance governor was active. This supports the explanation that frequency scaling is responsible for decreased PPS latency under heavy CPU load. It also implies that using the performance governor can improve timekeeping performance.

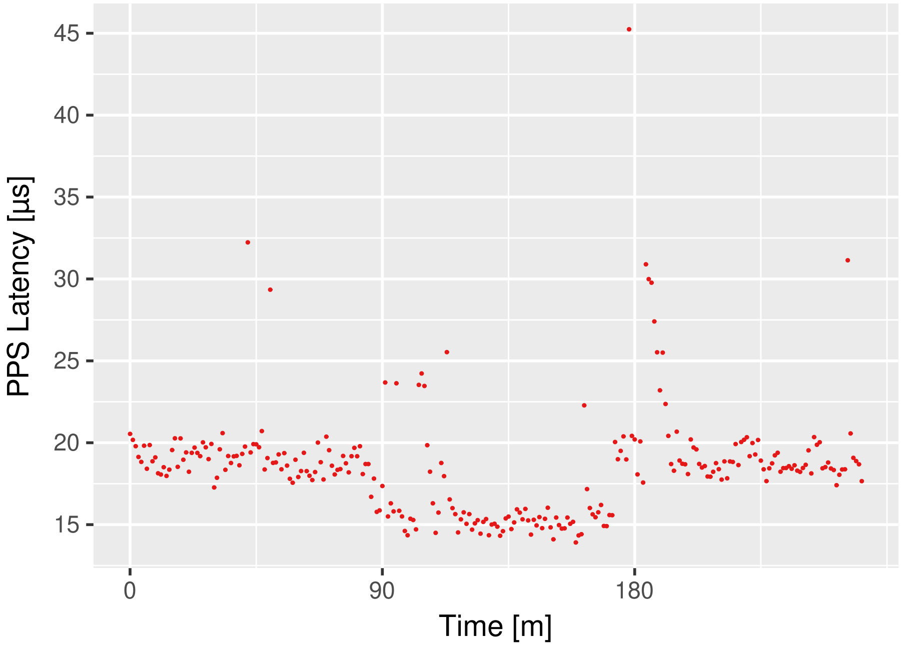


Figure 4.19 – PPS latency measured with three different scaling governors enabled. Ondemand and powersave (left and right) show similar latencies, performance (middle) shows decreased latencies.

Chapter 5

Conclusion

The results show that we have created a usable NTP reference clock driver for u-blox receivers. The driver has expected features like configurability, Survey-In and various synchronization modes. The driver is on par and can exceed performance of the comparable Oncore driver.

Configuration is very flexible. There is a driver specific configuration file which allows more freedom than NTPs mode bits and lets the user configure the driver and receiver for most common usage scenarios. Among other things a Survey-In can be configured and started. If additional control over the receiver is required, the configuration file can be used to send arbitrary messages.

The modes include a basic UTC mode for easy setup and a mode for conventional PPS synchronization. Additionally we introduce timemark mode, which uses the PPSAPI echo feature and the u-blox timemark feature to generate offsets which don't include local interrupt latency. On top of this, measurement mode can be used to record PPS latency for individual PPS pulses.

In comparison to the Oncore driver, we show that the new driver exhibits comparable or lower jitter in PPS mode. Timemark mode shows an even more pronounce improvement, suggesting that the removal of local interrupt latency is a valid strategy for improving reference clock synchronization.

We go on to show how measurement mode can be used to produce valuable insight into interrupt latency behaviour and PPS mode performance. In particular we investigate the influence of CPU affinity and CPU load with some surprising results.

Finally, we test the new driver in a non-professional, low-cost environment and show that the new driver is a valuable new tool in this environment as well. The results from this more informal test also point at further interesting research topics.

There is a broad range of operating system configuration options pertaining to CPU behaviour. They could be collected and their influence on PPS/interrupt latency measured. Another interesting path would be the influence of interrupt load on PPS latency. The topic was also looked at in this thesis, however not exhaustively and not conclusively. The results could be used to improve conventional PPS setups as well as other applications which rely on interrupts.

Overall it can be said that the new u-blox NTP reference clock driver improves upon the status-quo and opens up further areas of research.

List of Figures

2.1 The relation of local system time to UTC. 3
2.2 Left: A typical NTP subnet with one primary server (stratum 1). Right: The same subnet with one active link removed. [10] 4
2.3 Top: The 64 bit timestamp format. Bottom: The 128 bit datestamp format. [16] 5
2.4 The four timestamps used by NTP. [18] 6
2.5 Delay and offset calculation with the clock on B running ahead. . . . 7
2.6 The overall architecture of NTP as presented in RFC 5905 [1]. Clock statistics move from remote servers (or reference clocks) through the system where they are narrowed down and combined. Eventually they are used to discipline the variable frequency oscillator (VFO) of the computer. 7
2.7 Left: Raw offset values plotted over time. Right: Filtered values. [19] 8
2.8 Correctness intervals for three truechimers (A, B and C) and one falseticker (D) with their respective offsets θ_0 marked. Note that a truechimer's offset can lie outside the intersection interval. [20] . . . 9
2.9 The NTP clock discipline as shown and described in [23]. 9
2.10 Required functions of the PPSAPI [5] 11
2.11 Top: Two spheres intersecting in a circle (marked by the red line). Bottom: The same scene with a third satellite and rotated to face the intersection circle. One of the intersection points between the third sphere and the red intersection circle is the receiver position. 13
2.12 Measuring the transmission delay from satellite to receiver with PRN codes. 14
2.13 All configuration options supported by the Oncore reference clock driver. Further explanations can be found in the source code. 16
2.14 Structure of a UBX frame [4] 18

3.1 Data types in the UBX protocol and in C. [4] 20
3.2 Structure of a UBX frame [4] (same as figure 2.14) 21

3.3 Four messages A - D arriving in five chunks of different sizes. Each chunk corresponds to one call to receive. The graphic shows the reception state before and after processing the new chunk. The handler for message A is called in 2, for B and C in 4 and for D in 5. The first part of D is not long enough to determine the length of the message. 22
3.4 Theoretically possible modes of operation based on time synchronization methods. Unsupported combinations are grayed out. 25
3.5 All configuration options supported by the u-blox reference clock driver. 26
3.6 Offset calculation between receiver and computer using a PPS pulse and utc time messages. 27
3.7 False association (grey) in case of a lost PPS pulse. The driver will not use this pulse/message pair. 28
3.8 False association (grey) in case of a delayed utc message. The driver will not use this pulse/message pair. 28
3.9 Rising and falling edges and the `TIM-TM2` messages which will report their timestamps. Note that it is quite possible for edges to be lost, however, this occurs much less often when the frequency of the interrupt signal is the same as the measurement frequency (1Hz in this case). [4] 29
3.10 Offset calculation in timemark mode. The local timestamp comes from the PPSAPI, the receiver timestamp is reported in a `TIM-TM2` message. 30
3.11 If the receiver tries to send the `TIM-TM2` message to early it doesn't get sent, because there is no new timestamp to report (second 30). This leads to an offset of almost 1s unless precautions are taken. . . . 31
3.12 An example of a line in the loopstats file with individual fields explained. [35] 32

4.1 Experimental setup showing how the GNSS signal eventually reaches and disciplines the VFO. 36
4.2 Allan deviation of both receivers running freely without disciplining the VFO. 37
4.3 Offsets of a 24h measurement as reported in loopstats over time. . . 40
4.4 Jitter of a 24h measurement as reported in loopstats over time. 40
4.5 Average jitter of the receivers compared. The new u-blox driver/receiver combination performs about 300ns better than its counterpart. 41
4.6 Average jitter of the Oncore receiver with the `HARDPPS` configuration option set, compared to the results from above. 42
4.7 Offsets in timemark mode (green, narrow band) compared to PPS. . 43
4.8 Jitter in timemark mode (green, bottom) compared to PPS mode. . . 43

4.9 Reduction in average jitter with timemark mode compared to the conventional PPS mode. 44
4.10 PPS latency measured on the test platform. 45
4.11 Distribution of PPS latencies over 24h with bins of 100ns. 45
4.12 The data from figure 4.10 zoomed in on the first artefact. 46
4.13 An artificially generated latency spike similar to that from figure 4.12. 47
4.14 TODO 48
4.15 Distribution of PPS latencies with CPU affinity enabled compared to the distribution from figure 4.11. 49
4.16 Using CPU affinity results in high jitter. 50
4.17 Distribution of PPS latency with high CPU load compared to figure 4.11. 51
4.18 Average Jitter of u-blox PPS mode with high CPU load compared to previous results. 52
4.19 PPS latency measured with three different scaling governors enabled. Ondemand and powersave (left and right) show similar latencies, performance (middle) shows decreased latencies. 53

Bibliography

[1] D. Mills, J. Martin, J. Burbank, and W. Kasch, “Network time protocol version 4: Protocol and algorithms specification,” Internet Requests for Comments, RFC Editor, RFC 5905, June 2010, http://www.rfc-editor.org/rfc/rfc5905.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc5905.txt

[2] NTP Developers. (2014) Reference clock support. [Online]. Available: https://www.eecis.udel.edu/~mills/ntp/html/refclock.html

[3] ——. (2014) Pulse-per-second (pps) signal interfacing. [Online]. Available: https://www.eecis.udel.edu/~mills/ntp/html/pps.html

[4] *u-blox 8 / u-blox M8 Receiver Description*, 14th ed., u-blox, October 2017.

[5] J. Mogul, D. Mills, J. Brittenson, J. Stone, and U. Windl, “Pulse-per-second api for unix-like operating systems, version 1.0,” Internet Requests for Comments, RFC Editor, RFC 2783, March 2000.

[6] C. Bizouard, “Information on utc - tai,” INTERNATIONAL EARTH ROTATION AND REFERENCE SYSTEMS SERVICE, Bulletin C 55, January 2018, https://datacenter.iers.org/eop/-/somos/5Rgv/latest/16. [Online]. Available: https://datacenter.iers.org/eop/-/somos/5Rgv/latest/16

[7] R. A. Nelson, D. D. McCarthy, S. Malys, J. Levine, B. Guinot, H. F. Fliegel, R. L. Beard, and T. Bartholomew, “The leap second: its history and possible future,” *Metrologia*, vol. 38, no. 6, p. 509, 2001.

[8] M. Bauer, *Vermessung und Ortung mit Satelliten: globale Navigationssatellitensysteme (GNSS) und andere satellitengestützte Navigationssysteme*. Berlin u.a.: Wichmann, 2011.

[9] J. Levine, “Introduction to time and frequency metrology,” *Review of scientific instruments*, vol. 70, no. 6, pp. 2567–2596, 1999.

[10] D. L. Mills, “Internet time synchronization: the network time protocol,” *IEEE Transactions on communications*, vol. 39, no. 10, pp. 1482–1493, 1991.

[11] ——, “A brief history of ntp time: Memoirs of an internet timekeeper,” *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 2, pp. 9–21, 2003.

[12] ——. (2012) Executive summary: Computer network time synchronization. [Online]. Available: https://www.eecis.udel.edu/~mills/exec.html

[13] ——. (2012) Time synchronization for space data links. [Online]. Available: https://www.eecis.udel.edu/~mills/proximity.html

[14] T. Mizrahi and D. Mayer, “Network time protocol version 4 (ntpv4) extension fields,” Internet Requests for Comments, RFC Editor, RFC 7822, March 2016.

[15] D. L. Mills, graduate students, and volunteers. (2012) Network time synchronization research project. [Online]. Available: https://www.eecis.udel.edu/~mills/ntp.html

[16] D. L. Mills. (2012) Ntp timestamp calculations. [Online]. Available: https://www.eecis.udel.edu/~mills/time.html

[17] ——. (2012) The ntp era and era numbering. [Online]. Available: https://www.eecis.udel.edu/~mills/y2k.html

[18] ——. (2012) Timestamp capture principles. [Online]. Available: https://www.eecis.udel.edu/~mills/stamp.html

[19] NTP Developers. (2014) Clock filter algorithm. [Online]. Available: https://www.eecis.udel.edu/~mills/ntp/html/filter.html

[20] ——. (2014) Clock select algorithm. [Online]. Available: https://www.eecis.udel.edu/~mills/ntp/html/select.html

[21] ——. (2012) Clock cluster algorithm. [Online]. Available: https://www.eecis.udel.edu/~mills/ntp/html/cluster.html

[22] ——. (2014) Mitigation rules and the prefer keyword. [Online]. Available: https://www.eecis.udel.edu/~mills/ntp/html/prefer.html

[23] D. L. Mills, “Adaptive hybrid clock discipline algorithm for the network time protocol,” *IEEE/ACM Transactions on Networking (TON)*, vol. 6, no. 5, pp. 505–514, 1998.

[24] NTP Developers. (2014) Reference clock support. [Online]. Available: https://www.eecis.udel.edu/~mills/ntp/html/refclock.html

[25] D. Mills, *Computer network time synchronization : the Network Time Protocol on Earth and in space*. Boca Raton, FL: CRC Press, 2011.

[26] GPSD Developers. (2017) Compatible hardware. [Online]. Available: http://catb.org/gpsd/hardware.html

[27] ——. (2017) gpsd - a gps service daemon. [Online]. Available: http://catb.org/gpsd/

[28] P. Teunissen and O. Montenbruck, *Springer Handbook of Global Navigation Satellite Systems* -. Cham, Heidelberg, New York, Dordrecht, London: Springer International Publishing, 2017.

[29] European GNSS Service Centre. (2018) Programme. [Online]. Available: https://www.gsc-europa.eu/galileo-gsc-overview/programme

[30] W. Lewandowski and E. Arias, "Gnss times and utc," *Metrologia*, vol. 48, no. 4, p. S219, 2011.

[31] *M12M GPS Receiver User's Guide*, i-Lotus, May 2008.

[32] NTP Developers. (2014) Leap second processing. [Online]. Available: http://doc.ntp.org/current-stable/leap.html

[33] K.-S. J. Hielscher, "Measurement-based modeling of distributed systems," 2008.

[34] Kernel Developers. (2018) Github: raspberrypi/linux. [Online]. Available: https://github.com/raspberrypi/linux

[35] NTP Developers. (2014) Monitoring commands and options. [Online]. Available: https://www.eecis.udel.edu/~mills/ntp/html/monopt.html

[36] H. Schutte, "Bi-directional level shifter for i2c-bus and other systems," *Application note AN97055, Philips Electronics NV*, 1997.

[37] J. Zhu, "Conversion of earth-centered earth-fixed coordinates to geodetic coordinates," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 30, no. 3, pp. 957–961, 1994.

[38] T. I. Incorporated, "Sn74lvc3g17 triple schmitt-trigger buffer," 2015.