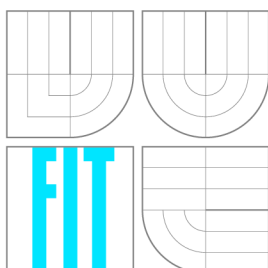


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

NTP KLIENT PRO SYSTÉM CONTIKI

CONTIKI NTP CLIENT

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE

AUTHOR

JOSEF LUŠTICKÝ

VEDOUCÍ PRÁCE

SUPERVISOR

Ing. TOMÁŠ KAŠPÁREK

BRNO 2012

Brno University of Technology - Faculty of Information Technology

Computer Centre

Academic year 2011/2012

Bachelor Project Specification

For: **Luštický Josef**
Branch of study: Information Technology
Title: **Contiki NTP Client**
Category: Operating Systems

Instructions for project work:

1. Study operating system for embedded devices Contiki and mainly its differences from POSIX standard. Study NTP protocol for precise time synchronization.
2. Devise necessary modifications for both Contiki operating system and NTP client and discuss portability of the client for this OS.
3. Implement necessary changes for the OS and port the NTP client for Contiki OS.
4. Provide a demonstration of NTP client functionality, discuss the amount of changes, that were necessary for both the client and OS. Are there any limitation for full NTP service support (daemon mode)?

Basic references:

- The Contiki OS, [online][cit. 2011-09-21]. Available on <<http://www.contiki-os.org/>>
- Network Time Protocol project, [online][cit. 2011-09-21]. Available on <<http://www.ntp.org/>>

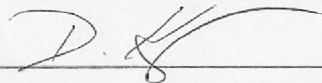
Detailed formal specifications can be found at <http://www.fit.vutbr.cz/info/szz/>

The Bachelor Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Kašpárek Tomáš, Ing.**, CC FIT BUT
Beginning of work: November 1, 2011
Date of delivery: May 16, 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
602 00 Brno, Božetěchova 2



Dušan Kolář

Associate Professor and Head of Department

Abstrakt

Účelem této práce je popis operačního systému Contiki pro vestavěné systémy, popis protokolu NTP pro synchronizaci času a vytvoření návrhu a implementace klienta protokolu NTP pro operační systém Contiki.

Abstract

The purpose of this thesis is to describe the operating system Contiki for embedded systems, NTP time synchronisation protocol and to design and implement an NTP client for the Contiki operating system.

Klíčová slova

operační systém, Contiki, synchronizace času, NTP, SNTP, IP síť, vestavěné systémy, protokol, hodiny, čas

Keywords

operating system, Contiki, time synchronisation, NTP, SNTP, IP network, embedded systems, protocol, clock, time

Citace

Josef Luštický: Contiki NTP Client, bakalářská práce, Brno, FIT VUT v Brně, 2012

Contiki NTP Client

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Tomáše Kašpárka.

.....
Josef Luštický
July 18, 2012

Poděkování

I would like to thank my supervisor Ing. Tomáš Kašpárek for helping me with practical advice and leadership, Prof. Dr. Reinhold Kröger and Kai Beckmann from Dopsy group at RheinMain University of Applied Sciences in Wiesbaden for providing an equipped workplace in the laboratory and helping me with hardware setup, and Lydia Wooldridge and Marcus Thoss for grammar corrections.

© Josef Luštický, 2012.

Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

1	Introduction	3
2	Contiki OS	4
2.1	Features	4
2.2	Protothreads	5
2.3	uIP	6
2.4	Kernel and processes	7
2.5	Timers	9
3	Network Time Protocol	10
3.1	Topology and hierarchy	10
3.2	Time and timescales	11
3.3	Network and timestamps	12
3.4	Algorithms	15
4	Analysis	17
4.1	Hardware clock	17
4.2	Contiki clock interface	18
4.3	Time interface	19
4.4	NTP client application	20
5	Design	22
5.1	Time interface extension	22
5.2	Clock library extension	24
5.3	Contiki NTP client	25
6	Implementation	28
6.1	Time interface extension	28
6.2	Clock library extension	30
6.3	Contiki NTP client	32
6.4	Code metrics	34
7	Measurements	36
7.1	Clock interrupt frequency	36
7.2	Clock offset	37
7.3	Clock phase	38
8	Conclusion	40

A Protothreads Example	44
B Clock Interrupt Frequency Measurements	46
C Clock Offset Measurements	48
D Clock Phase Measurements	50
E CD Contents	52

Chapter 1

Introduction

Nowadays we live in a world where embedded systems are part of almost every electronic device. Modern televisions contain embedded systems to allow you to browse the Web, modern cars use embedded systems to control an engine or to give you a summary of your journey using GPS, even fridges showing the list of things you should buy at a market are becoming popular. Embedded systems are becoming more widespread than ever and so do their needs for a network connection.

Contiki is an operating system targeted at embedded systems and developed by Adam Dunkels from the Swedish Institute of Computer Science in Kista, Sweden. Contiki brings new concepts to the embedded world and supports the Internet Protocol version 6 and 4. Since Contiki aims for maximum portability, it is written in the C programming language. Contiki therefore provides an ideal solution for connecting embedded systems to an existing network on many different hardware platforms.

Time synchronisation is nowadays also important. Almost every modern system needs to know the time - your video-recorder or home cinema automatically starts recording a film at a scheduled time, your washing-machine should have finished a selected program when you return home or your radio should automatically adjust its clock when the time changes due to daylight saving.

Network Time Protocol (NTP) is a ubiquitous time synchronisation protocol between computers in the modern Internet. Though being one of the oldest protocols, NTP is still developed and updated to conform to the latest network standards. The actual version at the time of writing is NTP version 4, which updates its previous version to accommodate Internet Protocol version 6.

This thesis describes the operating system Contiki, its concepts and philosophy, Network Time Protocol version 4 and design and implementation of an NTP client for the Contiki operating system.

Chapter 2

Contiki OS

Only 2% of all microprocessors that are sold today are used in PCs and the remaining 98% of all microprocessors are used in embedded systems [5]. Embedded systems have much smaller amounts of memory than PC computers. Moore's law predicts that these devices can be made significantly smaller and less expensive in the future. While this means that embedded system networks can be deployed to greater extents, it does not necessarily imply that the resources will be less constrained [11]. The memory constraints make programming for embedded systems a challenge.

The operating system Contiki is targeted at embedded systems based on MSP430, AVR, ARM, x86 and other architectures [7]. Contiki aims for maximum portability and therefore is written in C. It is a feature-rich operating system and only some of its features are described and used in this thesis.

Contiki is developed by a group of developers from industry and academia lead by Adam Dunkels from the Swedish Institute of Computer Science. The Contiki team currently consists of sixteen developers from SICS, SAP AG, Cisco, Atmel, NewAE and TU Munich [7]. Contiki is also deployed at RheinMain University in Wiesbaden. The 3-clause BSD license places minimal restrictions on redistribution of Contiki. Version 1.0 of Contiki OS was released in 2002, version 2.0 in 2007 and the latest version at the time of writing was version 2.5, released in 2011. The actual development happens using an online repository accessible on the Contiki homepage at <http://www.contiki-os.org/> by the Git version control system.

2.1 Features

Contiki OS features lightweight stackless threads called Protothreads. Protothreads introduced a new concept to the embedded world. They are extremely lightweight and compatible with standard C [13]. Each Protothread does not require a separate stack, which fits Protothreads perfectly for use in memory constrained embedded systems. Protothreads are discussed in more detail in section 2.2.

Apart from Protothreads, Contiki features a TCP/IP communication stack called uIP (micro IP) that conforms to the Request For Comments memorandums published by the Internet Engineering Task Force. The uIP stack allows Contiki to communicate over both IPv4 and IPv6 [7]. Contiki with its uIP stack is IPv6 Ready Phase 1 certified and therefore has the right to use the IPv6 Ready silver logo [16]. Before Contiki's uIP, the embedded world considered IP to be too heavyweight. All previous IP implementations for general pur-

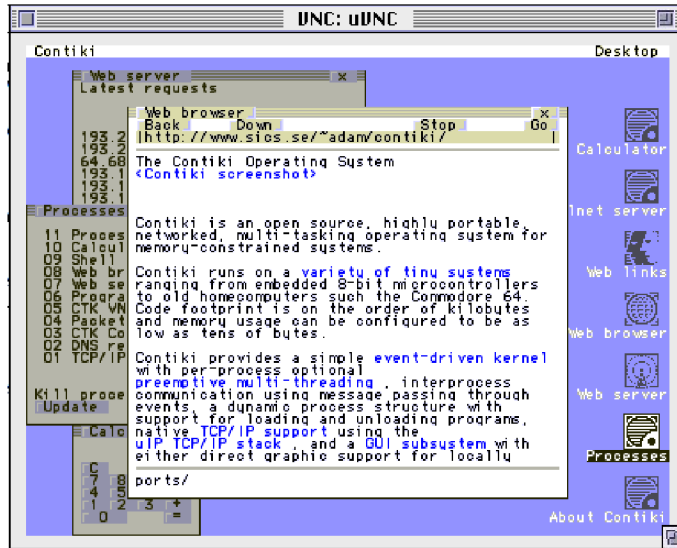


Figure 2.1: Screenshot of running Contiki OS with CTK (source: [7])

pose computers were much bigger than the memory constrained embedded systems could use [30]. The uIP communication stack is further described in section 2.3.

In addition to uIP, Contiki is equipped with another communication stack called Rime. Rime is a layered communication stack for sensor networks and it uses much thinner layers than traditional architectures [6]. Rime is designed to simplify the implementation of communication protocols on low-power radios. The communication primitives in the Rime stack were chosen based on what typical sensor network protocols use - single-hop unicast, single-hop broadcast or multi-hop [7, 6].

Besides Protothreads, uIP and Rime, Contiki further contains a very simple, relatively small and easy to use filesystem called Coffee Filesystem (CFS), a graphical user interface Contiki Toolkit (CTK) shown in figure 2.1, an Executable Linkable Format (ELF) loader for loading object files into a running Contiki system and much more.

The operating system Contiki, uIP and Protothreads are used on embedded devices by hundreds of companies in such diverse systems as car engines, oil boring equipments, satellites, and container security systems [5]. The software is also used in academic research projects and in university project courses all over the world.

2.2 Protothreads

Protothreads provide a way for C functions to run quasi-parallel, that is, a C function works in a way similar to a thread [13]. In Contiki, Protothreads allow to wait for incoming events without blocking the whole system. While waiting for an event to occur, another function could be run. The core of this solution is a C *switch* statement used in conjunction with a variable (called local continuation) containing the position, where the function was blocked [13]. The function continues from this point when it is later invoked again.

The advantage of Protothreads over ordinary threads is that a Protothread does not require a separate stack. The overhead of allocating multiple stacks can consume large amounts of available memory in memory constrained systems [13]. In contrast, each Protothread requires only a few bytes for storing the state of execution.

A Protothread is driven by repeated calls to the function in which the Protothread is running. Each time the function is called, the Protothread will run until it blocks or exits. Protothreads are implemented using the local continuations. The local continuation represents the current state of execution at a particular position in the program, but does not provide any call history or local variables [7].

The Protothreads API consists of four basic operations: initialisation *PT_INIT()*, execution *PT_BEGIN()*, conditional blocking *PT_WAIT_UNTIL()* and exit *PT_END()* [13]. However, experience with rewriting event-driven state machines to Protothreads revealed the importance of an unconditional blocking wait *PT_YIELD()*, which temporarily blocks the Protothread until the next time the Protothread is invoked [5]. After invoked, the Protothread continues executing the code following the *PT_YIELD()* statement. To understand the implementation of Protothreads and how they actually work, please refer to appendix A, where an example of use is shown.

Since Protothreads are implemented in standard C, a library providing Protothreads can be used everywhere, where the C toolchain is available. But there are some constraints to consider. Because Protothreads are stackless, a Protothread can run only within a single C function. There is also no way of storing automatic local variables [7]. And since Protothreads are implemented using a C *switch* statement, which can not be nested, the code that uses Protothreads can not use *switch* statements itself. A workaround for storing local variables is to prepend them with the *static* keyword, which makes them being put into the data segment by the compiler and thus remembering their values between the function calls [13].

2.3 uIP

The TCP/IP protocol suite is often used for communication over the Internet as well as local networks. uIP (micro IP) is a complete TCP/IP communication stack developed by Adam Dunkels for memory constrained systems such as embedded systems.

Before uIP, the TCP/IP architecture was considered to be heavyweight because of its perceived need for processing power and memory. The IP protocol was seen as too large to fit into the constrained environment - existing implementations of the IP protocol family for general purpose computers would need hundreds of kilobytes, whereas a typical constrained system has only a few tens of kilobytes of memory [30]. Several non-IP stacks were developed for this reason.

In the early 2000's, however, this view was challenged by lightweight implementations of the IP protocol family for smart objects such as the uIP stack [30]. uIP showed that the IP architecture would fit nicely into the typical constrained system without removing any of the essential IP mechanisms. Note that these resources, which are considered constrained today, are fairly close to the resources of general purpose computers that were available when IP was designed [30]. The uIP stack has become widely used in networked embedded systems since its initial release [30, 5].

uIP provides two different application programming interfaces to programmers: a BSD sockets-like API called Protosockets and a raw event-driven API. Protosockets are based on Protothreads, which puts the same limitations on them - such as no way to store the local variables and an impossibility to use C *switch* statements. Protosockets only work with TCP connections [7]. Because NTP uses the User Datagram Protocol (UDP), Protosockets will not be further discussed in this thesis. For more information about Protosockets please refer to the Contiki documentation [7].

uIP contains only the absolute minimum of required features to fulfill the protocol standard. It can handle only a single network interface and contains the IP, ICMP, UDP and TCP protocols [7]. In order to reduce memory requirements and code size, the uIP implementation uses an event-based API, which is fundamentally different from the most common TCP/IP API, the BSD sockets API, present on Unix-like systems and defined by the POSIX standard [5, 29]. An application is invoked in response to certain events and it is up to the application, that receives the events from uIP, to handle all the work with data to be transmitted - e.g. if the data packet is lost in the network, the application will be invoked and it has to resend the packet. This approach is based on the fact that it should be simple for the application to rebuild the same data. This way, the uIP stack does not need to use explicit dynamic memory allocation. Instead, it uses a single global buffer for holding the packets and it has a fixed table for holding the connection state. The global packet buffer is large enough to contain one packet of a maximum size [7].

When a packet arrives from the network, the device driver places it in the global buffer and calls the uIP stack. If the packet contains data, the uIP stack will notify the corresponding application. Because the data in the buffer will be overwritten by the next incoming packet, the application will either have to act immediately on the data or copy the data into its own buffer for later processing. The packet buffer will not be overwritten by new packets before the application has processed the data [7]. Packets arriving when the application is processing the data must be queued either by the network device or by the device driver. This means that uIP relies on the hardware when it comes to buffering. Most single-chip Ethernet controllers have on-chip buffers that are large enough to contain at least 4 maximum sized Ethernet frames [7]. This way, uIP does not have to have its own buffer structures and thus requires only a minimal memory amount. A possible packet loss is a trade-off for minimal memory requirements. It is not such a big problem for communication using TCP on the transport layer because of the acknowledgement scheme used in TCP to prevent data loss. However, data carried on UDP can be irrecoverably lost.

As was expected, measurements show that the uIP implementation provides very low throughput, particularly when it communicates with a PC host [4]. However, small systems that uIP is targeting, usually do not produce enough data to make the performance degradation a serious problem [4].

Despite being so small, uIP is not only RFC compliant, but also IPv6 Ready Phase 1 certified. uIP is written in the C programming language and it is fully integrated with the Contiki operating system. In uIP, there are even some more tricks to shrink the stack but its complete description is outside the scope of this thesis. Please refer to the Contiki documentation for more details [7].

2.4 Kernel and processes

The Contiki kernel is event-driven and provides a cooperative multitasking environment, but the system also supports preemptive multithreading that can be applied on a per-process basis [11]. The preemptive multithreading support is not implemented in the kernel, instead it is implemented as a library that is linked only with programs that explicitly require multithreading [11]. The kernel itself contains no platform specific code. It implements only CPU multiplexing and lets device drivers and applications communicate directly with hardware [11].

From high levels of abstraction, applications in Contiki OS are implemented and run as processes. Protothreads, the lightweight threads described in section 2.2, are used in

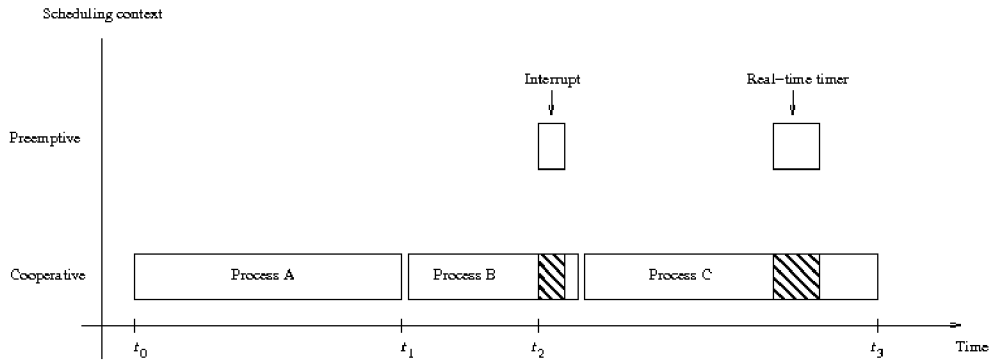


Figure 2.2: Contiki execution contexts (source: [9])

Contiki to implement processes. Both the Contiki kernel and applications use Protothreads extensively to achieve cooperative multitasking [12]. Every Contiki process consists of a process control block and a process thread [9]. The process control block contains run-time information about the process and the process thread contains the code of the process. Among other things, the process control block contains a textual name of the process, a pointer to the process thread and a state of the process. The process thread is implemented as a single Protothread, that is invoked from the process scheduler in the Contiki kernel [9].

From low levels of abstraction, every application is implemented as a simple C function and the process control block remembers the actual state of execution of this function in the same way as the local continuation works by Protothreads. Processes are therefore running quasi-parallel in Contiki.

The process control block is not declared or defined directly, but through the *PROCESS()* macro. This macro takes two parameters: a variable name of the process control block and a textual name of the process, which is used when debugging the system [9]. The process control block is shown in listing 2.1.

```

struct process {
    struct process *next;
    const char *name;
    int (* thread)(struct pt *, process_event_t, process_data_t);
    struct pt pt;
    unsigned char state, needspoll;
};

```

Listing 2.1: Process control block in Contiki OS (source [9])

All code execution is initiated by the Contiki kernel that acts like a simple dispatcher calling the functions [7]. Just like Protothreads, processes are also implemented using macros, making them standard C compatible.

In Contiki, code runs in either of two execution contexts: cooperative, in which code never preempts other code, and preemptive, which preempts the execution of cooperative code and returns control when the preemptive code is finished. Processes always run in the cooperative mode, whereas interrupt service routines and real-time timers run in the preemptive mode [9]. The code running in both execution contexts is illustrated in figure 2.2.

Interprocess communication is achieved by posting events in Contiki OS - processes communicate with each other by posting events to each other [11]. There are two types of events: synchronous and asynchronous. Synchronous events are directly delivered to the

receiving process when posted and can only be posted to a specific process [9]. Because synchronous events are delivered immediately, posting synchronous event is equivalent to a function call: the process to which an event is delivered is directly invoked, and the process that posted the event is blocked until the receiving process has finished processing the event [9].

Asynchronous events are delivered to the receiving process some time after they have been posted [9]. Before delivery, the asynchronous events are held on an event queue inside the Contiki kernel. The kernel loops through this event queue and delivers the event to the process by invoking the process. The receiver of an asynchronous event can be either a specific process or all running processes [9].

2.5 Timers

The Contiki kernel does not provide support for timed events, instead an application that wants to use timers needs to explicitly use a timer library. The timer library provides functions for setting, resetting and restarting timers, and for checking if a timer has expired. An application must manually check if its timers have expired - this is not done automatically [7].

Contiki has one clock library and a set of timer libraries: timer, stimer, ctimer, etimer, and rtimer [8]. The clock library provides functionality to handle the system time and to block the CPU for short time periods. It is the interface between Contiki and the platform-specific hardware clock [7]. The timer libraries are implemented with the functionality of the clock library as a base [8].

The timer and stimer libraries provide the simplest form of timers and are used to check whether a time period has passed. The difference between these two is the resolution of time - timers use system clock ticks, whose value is incremented when an interrupt from the hardware clock occurs, while stimers use seconds to offer much longer time periods [8]. The value representing seconds is also incremented in the interrupt service routine (ISR), but only when enough clock ticks since last increment occurred. The number of clock ticks within one second is represented by the *CLOCK_SECOND* macro provided by the clock library. That means there are *CLOCK_SECOND* interrupts from the hardware clock per second. The usage of the timer library and *CLOCK_SECOND* macro is shown in appendix A.

The simplest timer and stimer libraries are not able to post an event when a timer expires. Event timers should be used for this purpose. Event timers (etimer library) provide a way to generate timed events. An event timer will post an event to the process that set the timer when the event timer expires [7]. The etimer library is implemented as a Contiki process and uses the timer library as a base.

Callback timers (ctimer library) provide a timer mechanism that calls a specified C function when a ctimer expires [7]. Thus, they are especially useful in any code that does not have an explicit Contiki process [8].

The Real-time timers (rtimer library) handle the scheduling and execution of real-time tasks with predictable execution times [7]. The rtimer library provides real-time task support through callback functions - the rtimer immediately preempts any running Contiki process in order to let the real-time tasks execute at the scheduled time [8]. This behaviour is illustrated in figure 2.2. The rtimer library uses a separate hardware clock to allow a higher clock resolution [8]. The small part of the rtimer library is architecture-agnostic, but the particular implementation is platform-specific.

Chapter 3

Network Time Protocol

The Network Time Protocol provides a mechanism for synchronising systems' clocks over a variable-latency data network. NTP was introduced and is still developed by David Mills from the University of Delaware in Newark, United States of America [19]. NTP is arguably the longest running, continuously operating, ubiquitously available protocol in the Internet [20]. Despite being one of the oldest protocols in the Internet, it is not old-fashioned at all. NTP version 4, described in RFC 5905 [26], updates the older NTP version 3 to accommodate NTP to IPv6. Version 4 also includes improvements in mitigation and clock discipline algorithms that extend potential accuracy to the tens of microseconds with modern computers and fast LANs [26]. NTPv4 corrects some errors in NTPv3 design and includes an optional extension mechanism that can be used for adding more capabilities to NTP, e.g. the Autokey security protocol, described in RFC 5906, for authenticating servers to clients.

The Simple Network Time Protocol is a simplified NTP implementation, lacking complex synchronisation algorithms used by NTP [26]. SNTP is also described in RFC 5905 [26]. The packets of SNTP have the same structure and content as packets of NTP [26]. From observing the network communication, one can not tell, whether the client is a full blown NTP implementation or just SNTP. SNTP is a simplified sub-set of algorithms used by the NTP protocol, making the client implementation not only easier, but also suitable for resource constrained systems, such as embedded systems. Because NTP and SNTP servers and clients are completely interoperable and can be intermixed in NTP subnets [26], this thesis refers to an SNTP client for Contiki OS as NTP client.

3.1 Topology and hierarchy

NTP uses two different communication modes: one to one, referred as unicast mode, and one to many, referred as broadcast mode [26]. In unicast communication mode, an NTP client sends requests and an NTP server sends responses. In broadcast communication mode, the client sends no request and waits for a broadcast mode message from one or more servers [26].

NTP servers are rated with stratum (plural form strata) number representing their level in an NTP hierarchy and their possible accuracy [26]. Primary (stratum 1) servers synchronise to the reference clock directly traceable to UTC via radio, satellite or modem. The stratum 2 servers synchronise to stratum 1 servers via a hierarchical subnet. The stratum 3 servers synchronise to stratum 2 servers, and so on. The maximum stratum is 15,

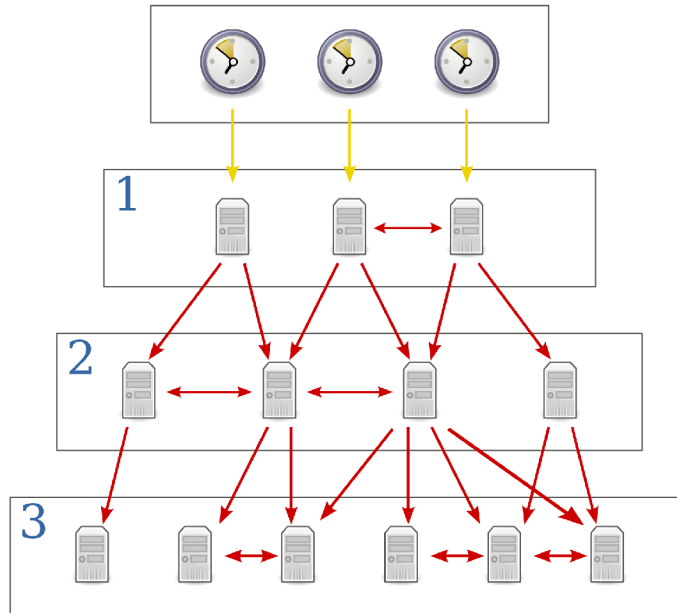


Figure 3.1: Topology and hierarchy of NTP (source: [14])

number 16 means unsynchronised server and higher numbers (up to 255) are reserved [26]. Synchronisation between servers in the same stratum level is also possible. Figure 3.1 shows a brief hierarchy of NTP. The subnet topology should be organised to avoid timing loops and to minimise synchronisation distances [26]. To achieve this in NTP, the subnet topology is determined using a variant of the Bellman-Ford distributed routing algorithm, which computes the shortest-distance spanning tree, rooted on the primary (stratum 1) servers [26]. As a result of this design, the algorithm automatically reorganises the subnet to produce the most accurate and reliable time, even when one or more primary or secondary servers or the network paths between them fail [26].

3.2 Time and timescales

To express the time, NTP always uses the Coordinated Universal Time (UTC) [26]. UTC is maintained by the International Bureau of Weights and Measures in Paris, France. It is the time scale that forms the basis for the coordinated dissemination of standard frequencies and time signals [28]. The time specified by UTC is the same for all timezones. Its calculation is practically the same as with Greenwich Mean Time (GMT), except the daylight savings are not accounted.

The UTC timescale represents mean solar time as disseminated by national standard laboratories [26]. This timescale is adjusted by the insertion of leap seconds to ensure approximate agreement with the time derived from the rotation of the Earth, which periodically speeds up and slows down due to the action of tides and changes within the Earth's core [28]. The goal of a leap second is to compensate UTC with these changes. A leap second is inserted or deleted on advice of the International Earth Rotation and Reference Systems Service [28]. NTP is well designed for a leap second occurrence - there is a Leap Indicator field in the structure of an NTP packet and there are also fields intended for the information about the leap second in structures that the NTP algorithm uses [26]. The

formal definition of UTC does not permit double leap seconds [29].

In a computer, the system time is represented by a system clock, maintained by hardware and the operating system. The goal of the NTP algorithms is to minimise both the time difference and frequency difference between UTC and the system clock. When these differences have been reduced below nominal tolerances, the system clock is said to be synchronised to UTC [26]. It has never been a goal of NTP to take care of local time, it is up to the operating system to provide users the correct local time [20].

The NTP and POSIX timescales are based on the UTC timescale, but not always coincident with it [23]. Both timescales reckon the time in standard (SI) seconds since the prime epoch, but the origin of the NTP timescale, the NTP prime epoch, is 00:00:00 UTC on 1 January 1900, while the prime epoch of the POSIX timescale is 00:00:00 UTC on 1 January 1970 [23]. So upon the first tick of the POSIX clock on 1 January 1970 the NTP clock read 2 208 988 800, representing the number of seconds since the NTP prime epoch.

3.3 Network and timestamps

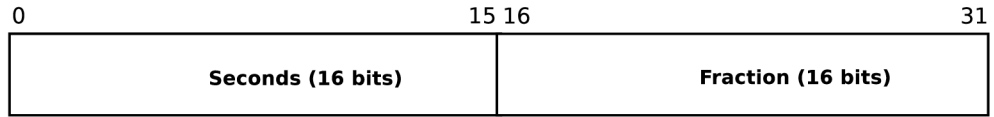
The network specification of NTP defines the protocol to use the User Datagram Protocol (UDP) on port number 123 [1, 26]. Reliable message delivery such as the Transmission Control Protocol (TCP) can actually make the delivery of NTP packets less reliable because retries would increase the delay value and other errors [26]. This is mostly because of the communication overhead.

NTP handles the time through timestamps - records of time. An NTP timestamp has two fields: a seconds field expressing the number of seconds and a fraction field expressing a fraction of a second [26]. All NTP timestamps are represented in 2's complement format, with bits numbered in big-endian fashion from zero starting at the left, or high-order position [26]. There are two timestamp formats in the NTP packet structure: a long 64-bit format and a short 32-bit format, as shown in figure 3.2. The 64-bit long timestamp consists of a 32-bit unsigned seconds field, spanning 2^{32} seconds (approx. 136 years from 1900 to 2036), and a 32-bit fraction field, resolving 2^{-32} seconds (approx. 232 picoseconds) [26]. The short 32-bit timestamp includes a 16-bit unsigned seconds field and a 16-bit fraction field.

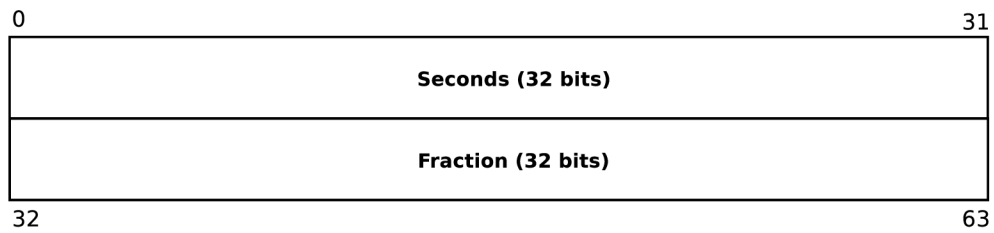
Besides these two formats, there is one more NTP timestamp format - the 128-bit NTP Date format. It includes a 64-bit signed seconds field and a 64-bit fraction field. For convenience in mapping between the formats, the seconds field of this format is divided into a 32-bit Era Number field and a 32-bit Era Offset field. This 128-bit NTP Date format is, however, not transmitted over the network and is only used where sufficient storage and word size is available [26]. There is practically no need of knowing about this format for embedded systems at least until the year 2036, when the 64-bit long timestamp wraps around and the Era Number will be incremented from zero to one. But strictly speaking, the NTP timestamp is a truncated NTP Date format [26].

The standard NTP packet structure without extension fields and Autokey security protocol is shown in figure 3.3. This structure is 48 bytes long and contains the following thirteen fields:

- Leap Indicator is a 2-bit integer, warning of an impending leap second to be inserted or deleted in the last minute of the current month [26].
- Version Number is a 3-bit integer, representing the NTP version number, currently 4.



NTP short 32-bit format



NTP long 64-bit format

Figure 3.2: Time formats used in NTP packet

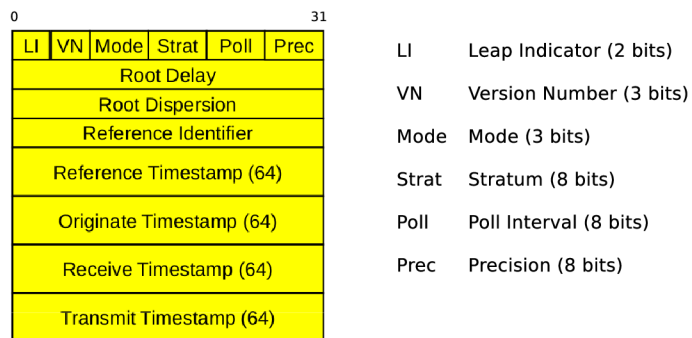


Figure 3.3: Basic NTP packet structure (source: [22])

- Mode is a 3-bit integer, representing the protocol mode. This is the only field distinguishing between servers and clients in NTP. In the client-server communication model, the client sets this field to value 3 (client) in the request, and the server sets it to value 4 (server) in the response. In the broadcast communication model, the server sets this field to value 5 (broadcast). Other modes are not used by SNTP servers and clients [26].
- Stratum is an 8-bit integer, representing the stratum as described in section 3.1. If the Stratum field is 0, which implies unspecified or invalid, the Reference Identifier field can be used to convey messages useful for status reporting and access control. These messages are called Kiss-o'-Death (KoD) packets and the ASCII messages they convey are called kiss codes [26].
- Poll is an 8-bit signed integer, representing the maximum interval between successive messages, in log2 seconds. Suggested default limits for minimum and maximum of the poll interval are 6 and 10, respectively [26].
- Precision is an 8-bit signed integer, representing the precision of the system clock, in log2 seconds. For instance, a value of -20 corresponds to a precision of about one microsecond (2^{-20} s) [26].
- Root Delay is a short 32-bit NTP timestamp, expressing the total round-trip delay to the reference clock [26].
- Root Dispersion is a short 32-bit NTP timestamp, expressing the total dispersion to the reference clock [26].
- Reference Identifier is a 32-bit code, identifying the particular server used for synchronisation or the reference clock. For packet stratum 0, this is a four-character ASCII string called kiss code. Kiss codes are particularly used by the server to tell the client to stop sending packets or to increase its polling interval. For stratum 1, this is a four-octet, left-justified, zero-padded ASCII string assigned to the reference clock (e.g. „GPS“ when synchronising against the Global Position System clock). Above stratum 1, this is the reference identifier of the server used for synchronisation and can be used by the client together with the stratum field to detect loops in the NTP hierarchy. If communicating over IPv4, the identifier is IPv4 address. If communicating over IPv6, it is the first four octets of the MD5 hash of the IPv6 address [26].
- Reference Timestamp is a long 64-bit NTP timestamp, expressing the time when the system clock was last synchronised against the reference clock [26].
- Originate Timestamp (or Origin Timestamp) is a long 64-bit NTP timestamp, expressing the time at the client when the request departed for the server [26]. The Originate Timestamp field is copied unchanged by the server from the Transmit Timestamp field of the client's request. It is important that the server copies this field intact, as the NTP client uses it to check the server's response.
- Receive Timestamp is a long 64-bit NTP timestamp, expressing the time at the server when the request arrived from the client [26].

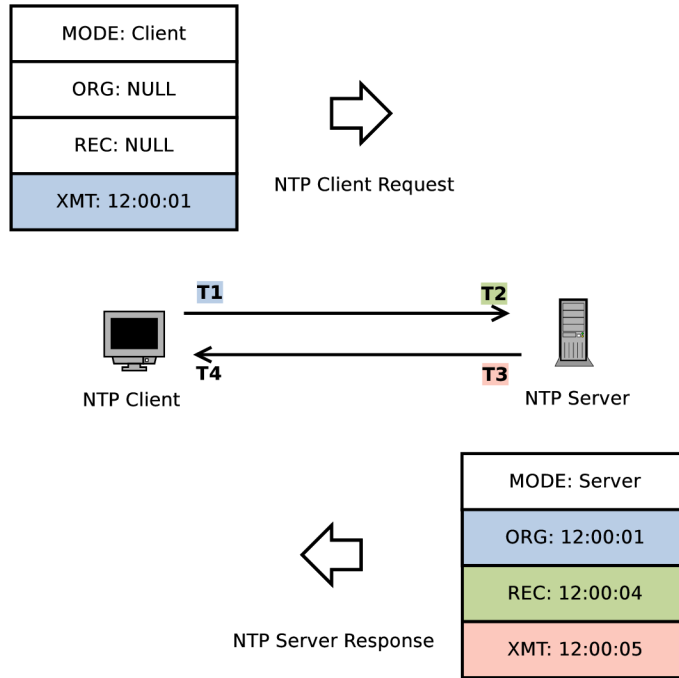


Figure 3.4: NTP unicast communication scenario

- Transmit Timestamp is a long 64-bit NTP timestamp, expressing the time at the server when the response left for the client [26].

The short 32-bit timestamp format is used for the Root Dispersion and Root Delay fields, because they do not need the scope and precision of the long 64-bit timestamp format. To better understand the NTP packet filling process and the meaning of Originate (org), Receive (rec) and Transmit (xmt) timestamps, a common unicast communication scenario is shown in figure 3.4. Please note that this figure shows only the timestamps used by NTP for the local clock offset and round-trip delay calculation. Algorithms used for the calculations are described in section 3.4.

3.4 Algorithms

Because of network latency, the received Transmit Timestamp will never be exactly corresponding to the current time. One of the main goals of NTP is to deal with the network latency [20].

As described in section 3.3, there are the following 64-bit long timestamps in the NTP packet: Origin, Receive and Transmit Timestamp. Upon NTP packet arrival, the client determines another timestamp called, Destination Timestamp [26]. This timestamp is represented as T4 in figure 3.4 and is not part of the NTP packet structure.

Using these four timestamps, the NTP client using unicast communication mode can compute the local clock offset which is given by $\theta = \frac{1}{2}[(t_2 - t_1) + (t_3 - t_4)]$, where t_1 is the time of the request packet transmission (Origin Timestamp), t_2 is the time of the request packet reception (Receive Timestamp), t_3 is the time of the response packet transmission (Transmit Timestamp) and t_4 is the time of the response packet reception (Destination Timestamp) [21, 26]. The implicit assumption in the above is that the one-way delay is

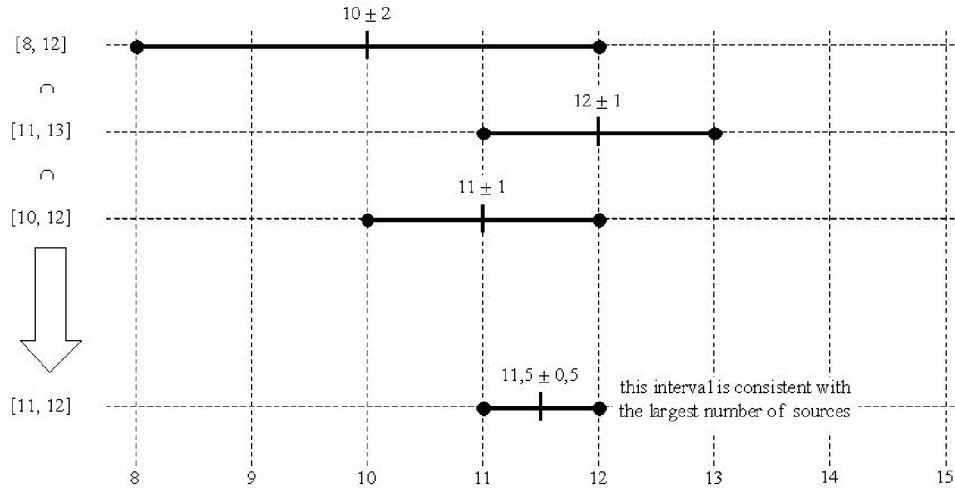


Figure 3.5: Intersection algorithm (source: [15])

statistically half of the round-trip delay [26], which is given by $\delta = (t_4 - t_1) - (t_3 - t_2)$.

In broadcast communication mode, Origin and Receive Timestamps are not accounted. The client computes its local clock offset which is given by $\theta = t_3 - t_4$. The implicit assumption in the above is that one-way delay from server to client is zero. Because this is never the case, it is useful to provide an initial volley where the client exchanges several packets with the server in order to calibrate the propagation delay [26].

When computing the result from more servers, the intersection algorithm is used for selecting the possible most exact timestamp received from various servers [18, 26]. Intersection algorithm is derived from Marzullo algorithm but the basic computation remains the same [19]. First of all, a selection of bad and good servers must be made. Bad servers are called Falsetickers and good are called Truechimers [26]. The division to these sets is based on their response. As one can assume, for a sensible result there must be more Truechimers than Falsetickers [26].

After selecting a set of reliable servers, the NTP algorithm compute the resulting timestamp. The resulting timestamp does not have to be the same as one of those provided by the servers. The NTP algorithm calculates using the clock accuracy estimates, determined by Root Dispersion, Root Delay and Precision fields of the server's response. These estimates are converted to intervals. Figure 3.5 shows the computation for the following example: If we have the estimates 10 ± 2 , 12 ± 1 and 11 ± 1 then these intervals are $\langle 8; 12 \rangle$, $\langle 11; 13 \rangle$ and $\langle 10; 12 \rangle$ which intersect to form $\langle 11; 12 \rangle$ or $11,5 \pm 0,5$ as consistent with all three values. The arithmetic mean is used as the result value. When querying servers again, the algorithm repeats but the new result computation also depends on the previous result [26, 19]. This eliminates possible jitter which can be caused by repeatedly querying the servers and getting slightly different answers from them.

Chapter 4

Analysis

To keep, measure and resolve the time, a computer needs a clock. A computer clock is an electronic device with a counter register counting oscillations in a quartz crystal oscillator with a particular frequency [27]. The operating system can keep and manipulate the system time using the hardware clock. Accessing the hardware clock is done through the operating system clock interface.

On top of the clock interface is an interface for providing the system time. To implement a reasonably useful NTP client, the operating system must be able to set, get and eventually adjust the system time through the time interface. Though not mandatory, adjusting the time is important in case the time shall be always a monotonically increasing function. Apart from that, an ability to communicate over UDP is also required for the NTP client. This is a task of the operating system network interface.

The NTP client is an application, that communicates with the NTP server through the network interface and uses the time interface of the operating system. It further calculates the local clock offset as described in section 3.4. In order to develop the NTP client for Contiki, the operating system Contiki and the hardware platform must provide the necessary components, as shown in figure 4.1.

To develop and test the Contiki NTP client, the AVR Raven platform with the 8-bit ATmega1284P CPU [3] will be used. This platform features IEEE 802.15.4 (Low-Rate Wireless Personal Area Network) link layer support. Together with an adaptation layer called 6LoWPAN (IPv6 over Low power Wireless Personal Area Networks), AVR Raven is able to communicate over IPv6.

4.1 Hardware clock

On AVR Raven, Contiki uses the 8-bit Timer/Counter 2 module, clocked from an asynchronous 32 768 Hz crystal oscillator, as the hardware clock by default. The oscillator is independent of any other clock, can only be used with Timer/Counter 2 and it enables the use of Timer/Counter 2 as a Real Time Counter [3]. The Timer/Counter 2 prescale value 8 is used in Contiki on the AVR Raven platform - the oscillator frequency of 32 768 Hz is effectively divided by 8 and the counter register is hence incremented with the frequency of $f_{T2} = \frac{f_{asy}}{prescaler} = \frac{32768}{8} = 4096$ Hz. Figure 4.2 shows the Timer/Counter 2 module used by Contiki.

The Timer/Counter 2 module is used in the Clear Timer on Compare Match (CTC) mode by Contiki. In this mode, the counter register *TCNT2* is incrementing and the

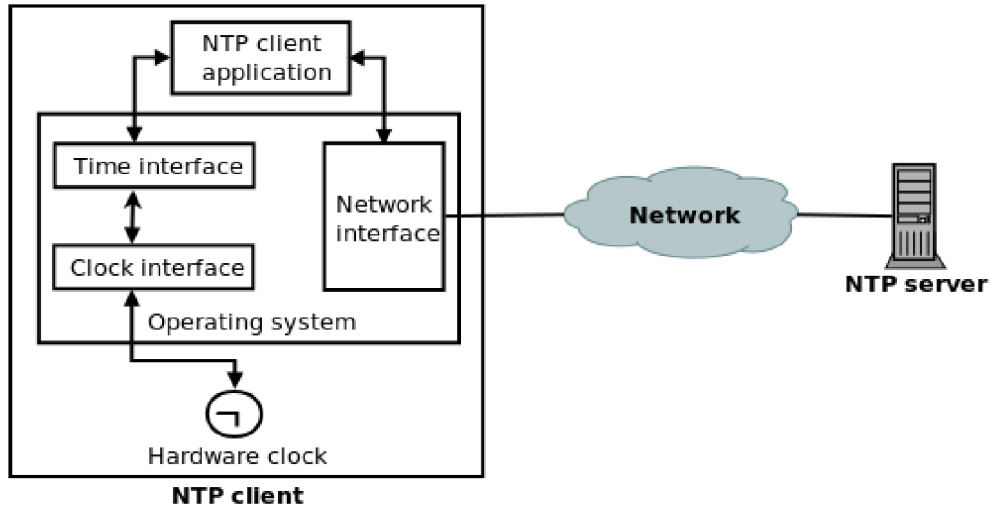


Figure 4.1: NTP client overview

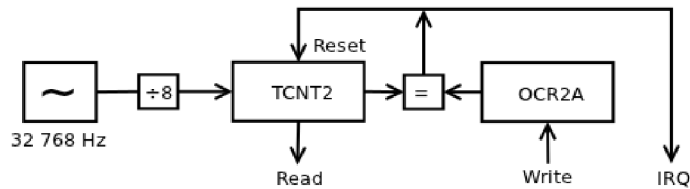


Figure 4.2: Timer/Counter 2 hardware clock module on AVR Raven

compare register *OCR2A* defines the maximum value of the counter register. A compare match between the counter register and the compare register sets the Output Compare Flag *OCF2A* and resets the counter register to zero [3]. This behaviour is illustrated in figure 4.3 - the *TOP* value is equal to the value in the compare register and the *BOTTOM* value is equal to zero.

Additionally, when the compare match occurs, an interrupt is raised and the interrupt service routine is executed. The flag indicating occurred match *OCF2A* is cleared automatically by hardware when executing the interrupt service routine in this case [3].

The interrupt service routine can be further used for updating the value in the *OCR2A* compare register. However, changing *OCR2A* to a value closer to zero while the counter is running must be done with care because the CTC mode does not have a double buffering feature. If the new value written to *OCR2A* is lower than the current value of *TCNT2*, the compare match will be missed [3].

4.2 Contiki clock interface

As described in section 2.5, Contiki provides a basic clock interface particularly for use of timers with a simple goal - measuring time. This interface is common for all supported hardware platforms, but the particular implementation is platform-specific. The definition of the common interface is located in the *core/sys/clock.h* file and the specific implementations can be found in the *clock.c* file in the *cpu/* directory of the Contiki source code.

The clock interface provides the *clock_init* call for initialising the hardware clock, that is

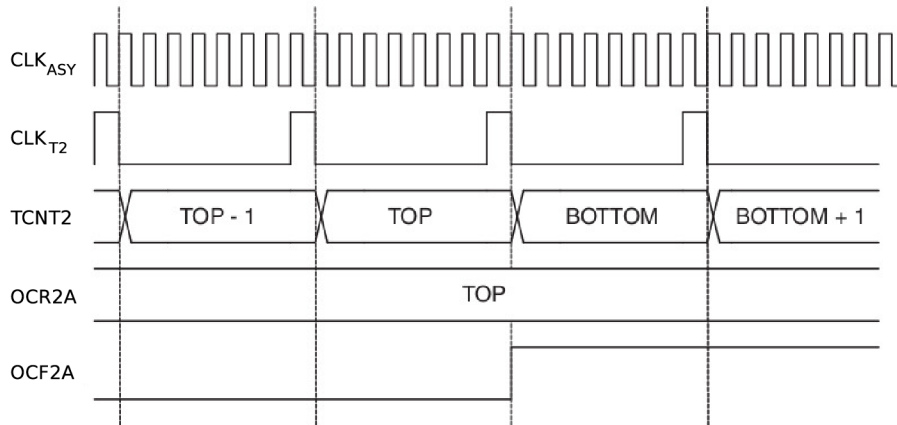


Figure 4.3: Timing diagram in CTC mode with prescaler 8 (source: [3])

automatically called during the boot sequence of Contiki. On AVR Raven, the *clock_init* call sets up appropriate registers and the interrupt service routine as described in section 4.1.

This call is implemented as a C macro and is defined for AVR CPUs in the file *cpu/avr/dev/clock-avr.h*. This macro evaluates to a specific setup code for each different AVR CPU during the compile time. The setup code is not common to all AVR CPUs because of differences among them - e.g. there are usually only three Timer/Counter modules, but AVR ATmega1284P has four Timer/Counter modules [3].

The hardware clock interrupt, described in section 4.1, is called *clock tick*, or *timer tick*. At each clock tick, the interrupt service routine increments a system clock value stored in the memory. On AVR CPUs, there is a variable counting these clock ticks, called *scount*, and a variable counting seconds, called *seconds*. These variables are defined together with the interrupt service routine in the *cpu/avr/dev/clock.c* file. As described in section 2.5, there are exactly *CLOCK_SECOND* ticks in one second. When the *scount* variable reaches the *CLOCK_SECOND* value, the *seconds* variable is incremented and the *scount* variable is reset. The *seconds* variable is used by the Contiki timers, discussed in section 2.5.

To obtain *CLOCK_SECOND* interrupts per second, there must be $\frac{f_{T2}}{CLOCK_SECOND}$ counter register increments between two successive interrupts. On compare match in CTC mode, the *TCNT2* counter register is reset to zero as shown in figure 4.3. The value zero is also included in the counting - the 0th count of the timer also takes one tick. Therefore the value of the *OCR2A* compare register must be $\frac{f_{T2}}{CLOCK_SECOND} - 1$ in CTC mode. The default value of *CLOCK_SECOND* for AVR Raven in Contiki is 128, which implies the default value of the compare register $\frac{4096}{128} - 1 = 31$. The *CLOCK_SECOND* value is defined in the *platform/avr-raven/contiki-conf.h* file. The value of the compare register is specified in the *clock_init* call and is computed during the compile time.

4.3 Time interface

The low-level clock interface described in the previous section is used by Contiki to provide the system time through the time interface. Because the value of the *seconds* variable is zero after the system booted, it actually represents the system uptime. The *seconds* variable is of the long data type and its value can be obtained using the *clock_seconds* call by the application. However, there is no call for setting this variable in Contiki 2.5. In the

current Git version at the time of writing, a new call `clock_set_seconds` can be used for this purpose. Because this call simply rewrites the `seconds` variable, it breaks the stimer library, and should therefore be avoided by the NTP client. Similarly, setting the `scout` variable would cause unbalanced increments of the `seconds` variable.

The precision of one second is also not adequate for the NTP client. Further precision can be acquired by reading the `scout` variable, as it provides a resolution of $\frac{1}{CLOCK_SECOND}$ seconds. Moreover, the hardware counter can be also queried, as it includes the time passed since the last update of the `scout` variable. Two read operations are needed then - read `scout` and read `TCNT2`. Because the `scout` variable depends on asynchronous interrupts produced by the clock module, the followed query of the counter register causes a race condition. The clock module runs asynchronously from the CPU clock and the result may be unpredictable if read while the module is running. To provide consistent time values, a proper solution must be designed.

If stimers should not be broken by setting the `seconds` or `scout` variable, and Contiki should be able to provide the current time in a higher precision, a new call interface must be designed. This call interface shall use the timescale and the time specification structure in compliance with the POSIX standard [29]. Such a structure for representing the time values is also not present in Contiki.

Similarly, there is no call for adjusting the time in Contiki. Because of memory constraints, software structures controlling the time adjustments are too heavyweight for use in an embedded operating system on 8-bit CPUs. Because of low CPU frequencies, the code of an interrupt service routine can not be complex and sophisticated clock discipline algorithms should be avoided. A call for adjusting the time should therefore use abilities provided by the hardware clock as much as possible.

Updating the value in the `OCR2A` compare register can be used to adjust the time, because decrementing the compare register value causes a faster increment of the `scout` variable, which in turn causes a faster increment of the `seconds` variable and vice versa. Such changes would influence the system time and the dependent Contiki timers. However, applications requiring uninfluenced timers could use the Contiki rtimers, described in section 2.5, because they use a separate hardware clock unaffected by these changes (Timer/Counter 3 in case of the AVR Raven platform).

4.4 NTP client application

Apart from the time interface, the NTP client application also needs to use the operating system network interface. Thanks to the uIP stack, described in section 2.3, the UDP network communication is not an issue for Contiki OS. The NTP client needs server associations in the NTP unicast communication mode. However, too many server associations complicate the client design. In fact, in the most common case, there can be only a single NTP master server in the whole network [26]. A single server association requires only a simple calculation of the local clock offset θ , whereas more server associations require the intersection algorithm described in section 3.4. Implementation of such an algorithm, requiring advanced data structures, should be avoided in a memory constrained environment.

The NTP broadcast communication mode, on the other hand, requires no server associations and no packet filling process on the client side. Moreover, because the client does not have to actively send any NTP packets, an energy consumption of the client is reduced. Contiki supports broadcast packets as well as sending multicast packets [7]. An implementation of NTP broadcast mode is therefore also possible. Joining multicast groups

through Internet Group Management Protocol (IGMP) and receiving non-local multicast packets was not supported at the time of writing [7].

The NTP client should be able to communicate over both IPv4 and IPv6. Thanks to the uIP stack, this is not an issue for Contiki. The only constraint is that both IP versions can not be used simultaneously and the decision must be made during the compilation [7]. Although the `UIP_CONF_IPV6` macro can be used to determine what IP version support is being compiled, the NTP client application should be written IP-version agnostic. Contiki is also able to use the Domain Name System for the resolution of IPv4 addresses. DNS resolution of IPv6 addresses was not implemented in Contiki OS at the time of writing [7].

A problem might be a possible packet loss when communication uses UDP on the transport layer. The reason, why this can happen often in Contiki, is explained in section 2.3. In NTP unicast mode, the packet loss might occur either for the client's query to the server or for the server's response to the client. If the client's query loss occurs, no server response will be sent. Similarly, if the server's response loss occurs, no message will be received by the client. Not to block the whole system till the response arrives is therefore a desired behaviour of the client.

The NTP client will further calculate the local clock offset using the NTP timestamps, as described in section 3.4. As mentioned in section 3.2, the NTP timescale is not coincident with the POSIX timescale. If the new calls in the time interface use the standard POSIX timescale, conversion between the NTP and POSIX timestamps will have to be calculated.

The client can set the Transmit Timestamp in its query to any arbitrary value. This is in compliance with the NTPv4 specification [26]. It is important for the client to store the sent timestamp, because it is later used by the client to check the server response. That practically means, that the conversion from the POSIX timestamp to the 64-bit long NTP timestamp is not needed when the client sends the request. However, the conversion vice versa is needed when the client calculates the local clock offset from the received timestamps.

Since both timescales reckon the time in seconds, the conversion between the seconds fields of the timestamps is simple. However, the conversion from the NTP fraction field value (2^{-32}) to the POSIX fraction field value (microseconds or nanoseconds) is problematic. The relation between the POSIX fraction field and the NTP fraction field is given by $POSIX.frac = NTP.frac \times POSIX.res \div 2^{32}$, where $POSIX.frac$ is the POSIX fraction field value, $NTP.frac$ is the NTP fraction field value and $POSIX.res$ is the POSIX timestamp resolution (10^6 or 10^9). The problem is that there is no portable solution for the operation of type $int_{64} := int_{32} \times int_{32}$ [17]. Therefore, the conversion requires either floating point operations or operations including 64-bit numbers. These operations can be memory expensive, especially on 8-bit microcontrollers, and their use must be considered carefully or another suitable solution must be designed.

Chapter 5

Design

The analysis showed, that just an implementation of the NTP client application is not sufficient, because of missing calls in the time interface. Analysis further described the necessary components of the NTP client.

The uIP stack, described in section 2.3, provides a feature-rich communication interface for the NTP client application. The communication interface is therefore not an issue. However, the time interface must be extended with new calls, that must be further implemented in the clock library. Figure 5.1 shows the overview of the NTP client on AVR Raven.

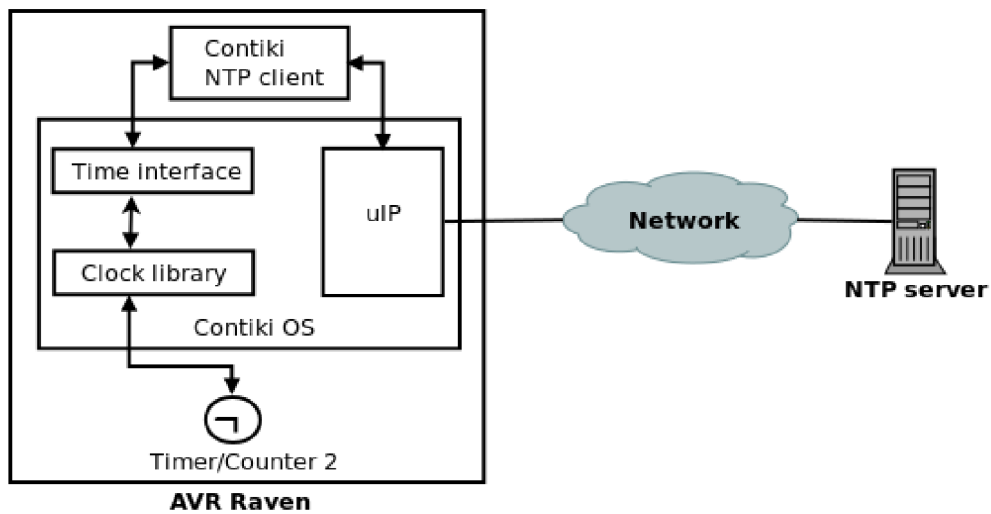


Figure 5.1: NTP client design overview

5.1 Time interface extension

Section 4.3 described, that there is no proper way of setting, getting and adjusting the time for an NTP client in Contiki OS. A new interface for setting, getting and eventually adjusting the time must therefore be developed.

Setting the time should not cause the misbehaviour of the Contiki timers, described in section 4.3. A modification of the *scount* and the *seconds* variable must be therefore avoided. This can be achieved using an additional variable, containing the system boot time, and modifying only this variable by a call for setting the time. This way, the *seconds*

variable will be further representing the system uptime and the current real time can be obtained by $boottime + seconds$. Since the *scout* also can not be changed, setting the time is only possible within a resolution of one second. The call for adjusting the time takes one parameter - the time to set, that is, the current real time in seconds since the POSIX prime epoch (1 January 1970). Because the current real time can not be negative, the parameter is of the unsigned data type. Listing 5.1 shows the call interface for setting the time.

```
void clock_set_time(unsigned long sec);
```

Listing 5.1: Call interface for setting the time

By contrast, a call for getting the current time must be able to provide a higher precision. The one second resolution is also not adequate for adjusting the time. Therefore, a new time specification structure must be introduced. To conform to the POSIX standard [29], this structure consists of two parts. The structure definition is shown in listing 5.2. The structure name was chosen *time_spec* to avoid collisions with existing POSIX-compliant systems. This structure consists of two signed long values representing seconds and nanoseconds. The nanosecond precision was chosen because modern systems also aim towards this precision and the microsecond precision would require at least the same data width [29, 24]. The value 0 seconds and 0 nanoseconds is equal to the POSIX prime epoch (1 January 1970). In case of the seconds part, the long data type was chosen because the already present value *seconds* is also of the long type in Contiki. To conveniently represent real-time values as well as local clock adjustment values, which may also be negative, the type was chosen to be signed. In case of the nanoseconds part, the signed long data type was chosen to conform to the POSIX standard [29] and to be able to represent positive as well as negative values for local clock adjustments.

```
struct time_spec {
    long sec;
    long nsec;
};
```

Listing 5.2: Time specification structure

Listing 5.3 shows the call interface for getting the time. The call for getting the time fills the time specification structure pointed to by the *ts* parameter. The part representing seconds is simply filled with the value of $boottime + seconds$, while the part representing nanoseconds should be filled with the maximum precision the clock model allows. As described in section 4.3, this can be achieved by reading the *scout* variable and by querying the hardware counter that is used for interrupt generation and includes the time passed since the last update of the *scout* variable. This way, a resolution of $\frac{1}{CLOCK_SECOND \times counts} = \frac{1}{128 \times 32} = 0.000244140625$ seconds can be acquired, where *counts* is the number of counter register increments between two successive interrupts, which is 32 by default on AVR Raven, as explained in section 4.2.

```
void clock_get_time(struct time_spec *ts);
```

Listing 5.3: Call interface for getting the time

Since setting the current time is possible only within one second precision, finer time setting must be made using the time adjustments. Section 4.3 explained, that adjusting the time should use the hardware clock as much as possible. Therefore, adjusting the time changes the value in *OCR2A* compare register to delay or shorten the clock tick interval,

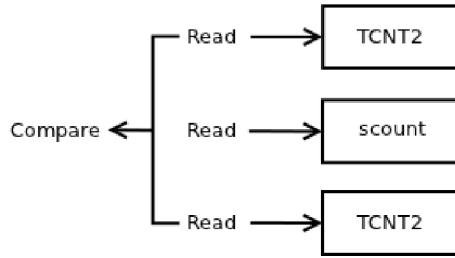


Figure 5.2: Multiple read and result comparison

which in turn speeds up or slows down the system time. To comply with other operating systems, the amount of required adjustments is specified by the time specification structure. If the amount of required adjustments is positive, then the system clock is speeded up until the adjustment has been completed. If the amount of required adjustments is negative, then the clock is slowed down in a similar fashion. Because the application specifies the amount of adjustments by the time specification structure, a new call must be introduced to determinate how many clock ticks will be delayed or shortened, respectively. Listing 5.4 shows the call interface for adjusting the time.

```
void clock_adjust_time(struct time_spec *delta);
```

Listing 5.4: Call interface for adjusting the time

As a result of this design, a leap second occurrence will be handled like an unexpected change of time - the operating system will continue with the wrong system time for some time, but the NTP client will set or adjust the system time. This will effectively cause the leap second correction to be applied too late [31], which is a trade-off for smaller memory requirements.

5.2 Clock library extension

The previous section described how the call for getting the time acquires the maximum precision the clock model allows. Section 4.3 showed that there can be unpredictable results caused by a race condition when reading the *scount* variable and the counter register *TCNT2* afterwards. Although the read could be wrapped with an interrupt disable, the common solution on AVR CPUs in Contiki is to perform more read operations, compare the results and perform read operations again if the results are not consistent. Figure 5.2 shows such a solution.

The call for adjusting the time computes the number of clock ticks with a longer or shorter tick interval. To store the result, a new variable must be introduced. If the value of this variable is zero, no adjustments are in progress. Otherwise, the default value of the compare register is incremented or decremented by 1 and written to *OCR2A*. This effectively causes adjusting the time, as described in section 4.3. When the time was adjusted enough, the default value of the compare register is written back to *OCR2A*.

Section 4.2 explained, that the default value of the compare register for AVR Raven in Contiki is 31, but the number of counter register increments between two successive interrupts is 32. Incrementing the compare register value causes one real second to be experienced as $\frac{f_{T2}}{CLOCK_SECOND \times (32+1)} = \frac{4096}{128 \times 33} = 0.96$ seconds by the operating system. Decrementing the compare register value causes one real second to be experienced as

$$\frac{f_{T2}}{CLOCK_SECOND \times (32-1)} = \frac{4096}{128 \times 31} \doteq 1.032258 \text{ seconds by the operating system.}$$

Each increment of the counter register *TCNT2* takes $\frac{1}{128 \times 32} = 0.000244140625$ seconds. This is also the minimal possible clock adjustment, that can be achieved by changing the compare register value just for one clock tick. The fastest possible adjustment is approximately $0.03 \frac{s}{s}$. If faster adjustments were needed, the compare register would have to be updated with different values. However, updating the compare register with more different values would require a more complicated software design and lower values could cause missing of the compare match as described in section 4.1.

5.3 Contiki NTP client

The client application itself is a Contiki process, which uses the designed operating system interface from the previous sections and the uIP communication stack.

The client is able to use both NTP communication modes, the NTP broadcast mode and the NTP unicast mode. If the client will use only the broadcast mode, the structures and code related to the unicast mode should not be included in the resulting program. NTP broadcast mode packets can be received and processed from any NTP server in the network. To avoid a complicated design when the NTP unicast mode is used, the client is able to communicate only with one specified NTP server.

The NTP client fills and checks only the seconds part of the NTP timestamp, because the conversion to the NTP format would increase the interval between the timestamp determination and the dispatch of the filled packet. After the filled NTP packet is sent, the client schedules sending the next NTP packet in 2^τ seconds by using the event timer library. In NTPv4, τ ranges from 4, resulting in NTP poll interval of 16 seconds, to 17, resulting in NTP poll interval of 36 hours. However, the event timer library imposes a limit on the scheduled time. This limit is platform specific and depends on the *CLOCK_SECOND* value, e.g. the τ value can not be greater than 8 on AVR Raven assuming 128 interrupts per second. Upon scheduling the event timer, the client process yields and another process could be run. The client process is later invoked either by an event announcing the server response or by the event timer in case no server response arrived.

For the lack of a simple solution for IPv4 communication over IEEE 802.15.4 link layer, only IPv6 communication will be tested on the AVR Raven platform. DNS resolution is not supported by the NTP client for this reason and the remote server must be specified by address.

The packet loss problem was described in section 4.4. However, a packet loss is not an issue if the client process uses the event timer library. In broadcast mode, a lost server packet causes no setting or adjusting the system time. The client simply waits without disruption for the next NTP broadcast message. If the client needs to figure out its local clock offset at the moment, it can query the server by using the NTP unicast mode. In unicast mode, the client process is invoked in response to the expired event timer and queries the server again.

When the server response arrives, the destination timestamp determination is one of the first tasks the client does. After that, the client makes packet sanity tests, such as checking whether the response is from a synchronised server. In unicast mode, the Originate Timestamp is compared with the stored sent timestamp. The received packet is considered bogus in case of mismatch and further processing is stopped. Otherwise, the NTP timestamps are converted to the local timestamp format and the local clock offset is computed as described

in section 3.4. After the local clock offset is computed, the stored transmitted timestamp is immediately set to zero to protect against a replay of the last transmitted packet. In broadcast mode, the received packet is always considered correct and the local clock offset is computed as the difference between the local stored timestamp and the received Transmit Timestamp. The local clock offset determined from the broadcast mode is influenced by the network propagation delay and therefore less accurate. The NTP client could exchange several packets with the server to calibrate the propagation delay. But since local variables can not be reused in the Contiki process when the process yields, this would cause either an extra memory overhead or a complicated client design.

Dynamic increasing or decreasing the client’s Poll interval in response to Kiss-o’-Death packets, described in section 3.3, would also require a complicated design. The client instead assumes, that an exhausted NTP server rather drops the incoming client’s query than sending a response with the KoD code.

Section 5.1 described that the client uses the POSIX timescale, whereas NTP uses the NTP timescale. Since both timescales reckon the time in seconds, the number of seconds between the NTP epoch and the POSIX epoch can be simply subtracted from the seconds part of the NTP timestamp. But the conversion from the fraction part of the long 64-bit NTP timestamp to nanoseconds, used in the local timestamp structure, is one of the most problematic tasks for memory constrained systems. An accurate conversion requires either floating point operations or operations including 64-bit numbers [17]. The conversion is given by $nsec = fractionl \times 10^9 \div 2^{32}$, where $nsec$ is the nanoseconds part of the local timestamp and $fractionl$ is the fraction part of the long 64-bit NTP timestamp.

Because there is no native hardware support for floating point nor 64-bit arithmetic operations, the compiler supplies these operations in form of a library, called *libgcc* in case of the GCC compiler. This causes a significantly bigger resulting binary file (kilobytes in case of floating point operations and hundreds of bytes in case of 64-bit operations). The greatest common divisor of 10^9 and 2^{32} is 2^9 , so in fact, a relatively simple multiplication of $fractionl$ by $\frac{5^9}{2^{23}}$ must be computed. This could be computed on 32 bits using sequential divisions by the power of 2 and multiplications by the power of 5. In the standard C programming language, the bitwise right shift operator divides the unsigned data type by the power of 2 and the bitwise left shift operator multiplies the unsigned data type by the power of 2 [17]. Therefore, the multiplication by 5 can be done using two left shifts and adding the original value ($5x = 4x + x$). The only constraint is that the overall coefficient of these operations must not be greater than 1, that is, the value must be in the range from 0 to $2^{32} - 1$ in every step. Otherwise, the value could overflow and the result would be incorrect. Division can not cause such a situation but multiplication could. The original value could be divided by a greater divisor, but this would lead to a greater inaccuracy because of loosing the least significant bits. Because of this, the multiplication done as soon as possible provides more accurate results. The ideal conversion sequence is therefore given by formula 5.1.

$$nsec = fractionl \times \frac{5}{2^3} \times \frac{5}{2^3} \times \frac{5}{2^3} \times \frac{5^2}{2^3} \times \frac{5}{2^3} \times \frac{5}{2^3} \times \frac{5^2}{2^3} \times \frac{1}{2^2} \quad (5.1)$$

It must be noted, that the above presented conversion is not exactly accurate, because the least significant bits are lost because of right shifting. The accuracy can be determined by iterating over all the possible values of $fractionl$ and comparing the results with the reference algorithm that uses the floating point operations. Such a measurement reports the maximum error of 5 nanoseconds, which is totally adequate for most platforms without

the floating point unit or for platforms where 64-bit multiplications are expensive. The implementation of the above as well as the program used for the error determination can be found on the CD attached to this thesis. The table of CD contents is listed in appendix E.

After the timestamps were converted, the local clock offset is computed as given in section 3.4. Depending on the absolute value of the local clock offset, the system time is either set or adjusted using the *clock_set_time* or *clock_adjst_time* call, respectively. The clock is set if the time difference is greater than or equal to the offset threshold value. The NTP specification suggests 0.125 seconds as the default offset threshold value [26]. Because the designed call for setting the time, described in section 5.1, can set the time only within a resolution of one second, the threshold value must be at least one second.

Chapter 6

Implementation

This chapter describes the implementation of the designed interface extensions and the implementation of the designed NTP client application. The following software was used for the development: Ubuntu 10.10, GCC 4.3.5, Binutils 2.20.1, AVR Libc 1.6.8 and Contiki 2.5. The AVR Dragon programmer and AVRDUDE 5.10 were used for flashing the AVR Raven hardware.

How to get a working setup with Contiki on the AVR Raven platform is described in the documents on the CD enclosed to this thesis. The table of CD contents is listed in appendix [E](#).

6.1 Time interface extension

Since there is no proper way of setting, getting and adjusting the time in Contiki OS, the interface for setting, getting and adjusting the time must have been developed. The implementation uses the designed call interface and time specification structure from section [5.1](#).

6.1.1 Time specification structure

A new structure for expressing time values was implemented. This structure is shown in listing [6.1](#). Section [5.1](#) described the reasons for the chosen data types in this structure. The implicit assumption is that the compiler chooses at least 32-bit data width for the long data type. According to ISO C99 standard [[17](#)], the maximum value for an object of type signed long shall be greater or equal $2^{31}-1$ (2 147 483 647) [[17](#)]. This in fact results in at least a 32-bit variable unless the compilation setting is changed. Such a time representation will wrap around in the year 2038.

```
struct time_spec {
    long sec;
    long nsec;
};
```

Listing 6.1: Time specification structure

6.1.2 Setting the time

Setting the time is only possible within one second precision - finer time setting must be made using the time adjustments. The implemented *clock_set_time* function computes

when the system booted in seconds since the POSIX epoch and saves the result in the newly implemented *boottime* variable. This avoids the misbehaviour of Contiki timers, described in section 5.1.

Thanks to the implemented *clock_set_time* function and *boottime* variable, the running Contiki system is able to tell the uptime, the current real time and the time when the system was booted. Listing 6.2 shows the function for setting the time.

```
volatile unsigned long boottime;

void clock_set_time(unsigned long sec)
{
    boottime = sec - seconds;
}
```

Listing 6.2: Function for setting the time

6.1.3 Getting the time

Getting the correct current real time is only possible if it was set using the *clock_set_time* function before. The implemented *clock_get_time* function is then able to tell the current time in seconds since the POSIX epoch by simply adding *boottime* and *seconds*.

The nanoseconds part is filled by reading the *scout* variable and the hardware counter register. The comparison avoids the need to disable clock interrupts to prevent unexpected results. This is a common practice on the AVR CPUs in Contiki, as described in 5.2. In order to minimise the possible comparison mismatch, the consistent values are obtained first and used for computation afterwards. The seconds part is compared in a similar manner - if the seconds part is not consistent, the other values might not be consistent as well. Listing 6.3 shows the function for getting the time, where *CLOCK_CTC_MODE* equals 1 and *CLOCK_COMPARE_DEFAULT_VALUE* is equal to the value of the compare register when no time adjustments are in progress.

```
void clock_get_time(struct time_spec *ts)
{
    uint8_t counter, tmp_scount;
    do {
        ts->sec = boottime + seconds;
        do {
            counter = CLOCK_COUNTER_REGISTER;
            tmp_scount = scout;
        } while (counter != CLOCK_COUNTER_REGISTER);

        ts->nsec = tmp_scount * (1000000000 / CLOCK_SECOND) +
            counter * (1000000000 / (CLOCK_SECOND *
                (CLOCK_COMPARE_DEFAULT_VALUE + CLOCK_CTC_MODE)));
    } while (ts->sec != (boottime + seconds));
}
```

Listing 6.3: Function for getting the time

Because both *1000000000* and *CLOCK_SECOND* are constants, the compiler is able to calculate the result of the division during the compile time. Furthermore, as both numbers are integers, the result is integer as well [17]. Most of the CPU time is therefore spent on the multiplications where the variables *counter* and *tmp_scount* are involved. If the code is compiled using GCC version 4.3.5, one such a multiplication of two 32-bit variables

takes 33 instructions including the *call* and *ret* instructions for entering and returning from the *_mulsi3* routine, which computes the result. This results in 48 clock cycles of overhead, which takes 6 000 nanoseconds with an 8 MHz CPU clock, according to the AVR Instruction Set manual [2]. The timestamp provided is therefore not exact. Because the time consumed strongly depends on the architecture and compiler specifications, no correction was implemented to remove this inaccuracy. The application must be instead aware that the timestamp is not exactly accurate.

6.1.4 Adjusting the time

A new function computing the amount of required adjusted ticks was implemented. The *clock_adjust_time* function stores the computed result in a new variable called *adjcompare*, which is further discussed in section 6.2. If the amount of required adjustments is positive, then the system time is speeded up until the adjustment has been completed and vice versa. If the amount of required adjustments is 0 seconds and 0 nanoseconds, then are the eventual adjustments in progress stopped, but any already completed part is not undone. The time values that are between two successive multiples of the clock resolution are truncated. Listing 6.4 shows the function for adjusting the time.

```

void clock_adjust_time(struct time_spec *delta)
{
    if (delta->sec == 0L) {
        if (delta->nsec == 0L) {
            adjcompare = 0; // stop adjustments
            return;
        } else {
            adjcompare = -delta->nsec / (1000000000 / (CLOCK_SECOND *
                (CLOCK_COMPAREDEFAULT.VALUE + CLOCK_CTC_MODE)));
        }
    } else {
        adjcompare = -delta->sec * (CLOCK_SECOND *
            (CLOCK_COMPAREDEFAULT.VALUE + CLOCK_CTC_MODE)) +
            -delta->nsec / (1000000000 / (CLOCK_SECOND *
                (CLOCK_COMPAREDEFAULT.VALUE + CLOCK_CTC_MODE)));
    }
}

```

Listing 6.4: Function for adjusting the time

6.2 Clock library extension

Section 4.2 described, that the *clock_init* call evaluates to a specific setup code for each different AVR CPU during the compilation. Other AVR CPUs can be using a different hardware clock. However, the time interface is common for all AVR CPUs. Therefore, new general names for each part of the hardware clock must be defined in the *clock_init* call.

The compare register *OCR2A* is defined by macro as *CLOCK_COMPARE_REGISTER*, the counter register *TCNT2* is defined as *CLOCK_COUNTER_REGISTER*, the default value of the clock compare register, when no time adjustments are in progress, is defined as *CLOCK_COMPARE_DEFAULT_VALUE* and *CLOCK_CTC_MODE* is defined as 1, because the hardware clock is used in CTC mode, as described in section 4.1. So in fact, the code presented in the previous section is not a pseudocode. Such an extension also makes porting the code to other platforms simple.

The *clock_adjust_time* function uses the new *adjcompare* variable, as described in the previous section. The data type of this variable was chosen to be of the signed 16-bit type. The limit imposed on time adjustments is therefore 2^{15} counter register increments for slowing down the clock and $2^{15} - 1$ for speeding up the clock. This equals to $2^{15} \times 0.000244140625 = 8$ seconds and $(2^{15} - 1) \times 0.000244140625 \doteq 7.999756$ seconds, respectively. A wider data type of the *adjcompare* variable would cause greater scope, but also an additional memory overhead. Since it has to be possible to adjust the time at least within one second, a smaller data width of *adjcompare* would cause too small scope. The *volatile* modifier must be used in conjunction with this variable, because the variable may be updated in the interrupt service routine. Listing 6.5 shows the variable definition and its use in the interrupt service routine for adjusting the time.

```
volatile int16_t adjcompare;

ISR(AVR_OUTPUT_COMPARE_INT)
{
    ...
    if (adjcompare == 0) { // if not adjusting
        CLOCK_COMPARE_REGISTER = CLOCK_COMPARE_DEFAULT_VALUE;
    } else if (adjcompare > 0) { // if slowing down
        adjcompare--;
        CLOCK_COMPARE_REGISTER = CLOCK_COMPARE_DEFAULT_VALUE + 1;
    } else { // if speeding up
        adjcompare++;
        CLOCK_COMPARE_REGISTER = CLOCK_COMPARE_DEFAULT_VALUE - 1;
    }
    ...
}
```

Listing 6.5: Pseudocode of adjustments in interrupt service routine

6.3 Contiki NTP client

The structure representing an NTP message was borrowed from the OpenNTPD daemon and the Dragonfly NTP daemon. This structure is not using the GCC extension for representing a bit field, instead it uses a single 8-bit integer called *status* for Leap Indicator, Version Number and Mode fields of the NTP packet structure, described in section 3.3. Accessing each field of the *status* byte is done using the bitmasks. Unlike using the bit field extension, this is compliant with the standard C language [17].

Parameters such as the remote NTP server address, the offset threshold value or the τ exponent of the NTP poll interval can be configured by a standard C *define* macro in the source code or in the Makefile. The default offset threshold value was chosen 3 seconds. Approx. 1% of this amount can be adjusted in 1 second. The unicast mode can be turned off by specifying no remote host. In this case, all of the code related to the unicast mode will not be compiled.

The client is IP-version agnostic and the *UIP_CONF_IPV6* macro is used only when printing the remote server address for debugging purposes. The remote NTP server address can be either IPv4 or IPv6 address, but can not be specified by a domain name. Communication over IPv4 was not tested though, for the lack of a simple solution for IPv4 communication over IEEE 802.15.4.

The NTP client is written as an event-driven process, that never exits. Upon sending the NTP query, the NTP client process sets the event timer, yields and waits for the next event by using the *PROCESS_WAIT_EVENT* statement. This is comparable to the daemon mode in other operating systems. The NTP process is later invoked either in response to the incoming packet event or in response to the timer expiration event. The type of the event is determined by the *if* statement. This way, no active waiting blocks the whole system. Before entering the main loop, the client sends its first NTP query after 6 seconds of uptime to set the system time. Otherwise, the first set of the system time would happen after the event timer expires (2^τ seconds). The value of 6 seconds should provide enough time for Contiki to configure the network interface. If the network interface is not configured at that time, the client enters the main loop and schedules sending the next packet as described above.

If any other application wants the NTP client to query the server, it can send the *PROCESS_EVENT_MSG* event to the NTP process at any time. However, no event is sent to that application when the server response arrives or when the system time is changed. The application can instead experience a change of the system time using the *clock_get_time* call. Listing 6.6 shows the Contiki NTP client pseudocode.

```

PROCESS.THREAD(event)
{
    for(;;) { // main loop
        PROCESS_WAIT_EVENT();
        if(event == tcpip_event) {
            tcpip_handler();
        }
#ifdef REMOTEHOST // unicast mode support
        else if(etimer_expired()) {
            timeout_handler();
            etimer_restart(); // set etimer to 2^TAU seconds
        } else if(event == PROCESS_EVENT_MSG) // event from another application
        {
            timeout_handler();
        }
#endif
    }
}

tcpip_handler(void) // process incoming server packet
{
    clock_get_time();
    offset = compute_offset();
    if(abs(offset) >= ADJUST_THRESHOLD) {
        clock_set_time();
    } else {
        clock_adjust_time();
    }
}

#ifdef REMOTEHOST
timeout_handler(void) // send query to REMOTEHOST
{
    clock_get_time();
    fill_and_send_ntp_packet();
}
#endif REMOTEHOST

```

Listing 6.6: NTP client pseudocode

Listing 6.7 shows the conversion from the NTP timestamp to the POSIX timestamp, as presented in section 5.3. The conversion uses only shifts and additions, which makes the resulting binary file significantly smaller. Because the current NTP Era ends in 2036, the conversion between the seconds fields has to be changed in the future [25].

According to the output from the *avr-size* tool, the use of 64-bit arithmetic operations for the conversion takes 728 bytes more in the resulting binary file (GCC supplies routines for multiplication and shifting 64-bit integers) and the use of floating point operations takes 3 358 bytes more than the developed algorithm. Besides significantly smaller memory requirements, it was observed that this algorithm provides on AVR Raven more accurate results than the libgcc floating point library supplied by GCC.

```

#define JAN_1970 2208988800UL /* 1970 - 1900 in seconds */

void ntp_to_ts(const struct l_fixedpt *ntp, struct time_spec *ts)
{
    ts->sec = ntp->int_partl - JAN_1970;
    ts->nsec = fractionl_to_nsec(ntp->fractionl);
}

unsigned long fractionl_to_nsec(uint32_t fractionl)
{
    unsigned long nsec;
    nsec = fractionl;
    nsec = (nsec >> 1) + (nsec >> 3); // nsec = nsec/2 + nsec/8 = (5*nsec)/8
    nsec = (nsec >> 1) + (nsec >> 3); // nsec = (5*nsec)/8 = (25*fractionl)/64
    nsec = (nsec >> 1) + (nsec >> 3); // nsec = fractionl * 5^3/2^9
    /* Now we can multiply by 5^2 because then the total
     * multiplication coefficient of the original number fractionl
     * will be: fractionl * (5^5)/((2^3)^4) = fractionl * 0.762939453,
     * which is less than 1, so it can not overflow.
     */
    nsec = (nsec << 1) + nsec + (nsec >> 3); // nsec*3 + nsec/8 = (25*nsec)/8

    nsec = (nsec >> 1) + (nsec >> 3);
    nsec = (nsec >> 1) + (nsec >> 3);

    /* Again we can multiply by 5^2. The total coefficient will be
     * fractionl * (5^9)/((2^3)^7) = fractionl * 0.931322575
     */
    nsec = (nsec << 1) + nsec + (nsec >> 3); // nsec*3 + nsec/8 = (25*nsec)/8

    /* Last shift to agree with division by 2^23 can not be done earlier
     * because the total coefficient would always be greater than 1.
     */
    nsec = nsec >> 2;
    return nsec;
}

```

Listing 6.7: Conversion from NTP timestamp to POSIX timestamp

6.4 Code metrics

The developed NTP client and patches for Contiki version 2.5 as well as the actual Contiki Git version (at the time of writing) are provided on the CD attached to this thesis. The NTP client application is common for both Contiki versions. The code uses the Contiki indentation style.

The patch extending the Contiki operating system version 2.5 inserts 72 new lines of code and modifies 1 line of code. The modified line is a backport from the actual Contiki Git version to prevent missing the compare match between the *scount* variable and the *CLOCK_SECOND* value.

The patch extending the actual Contiki Git version (committed on 27 June 2012) inserts 78 new lines of code and modifies 1 line of code. The modified line fixes a reported bug, which was not fixed at the time of writing. This bug causes a wrong decision of the C preprocessor whether the *CLOCK_SECOND* value is a power of two, which in turn may cause a division operation in the clock interrupt service routine.

The NTP client application has 2 code files and 1 header file. The code file containing the definition of the NTP process has 198 lines of code. The second code file contains various conversions from NTP fraction part to nanoseconds. The user may choose by using the C *define* macro which conversion will be used by the NTP client application. This file has 52 lines of code.

Various code metrics of the NTP client application code shows listing 6.8. These metrics were acquired using the *cloc* program.

Language	files	blank	comment	code
C	2	52	122	198
C/C++ Header	1	19	54	52
SUM:	3	71	176	250

Listing 6.8: NTP client application code metrics

Chapter 7

Measurements

There are several factors that can be measured. The clock interrupt frequency measurements show the influence of the clock adjustments on the number of clock ticks (interrupts) per second. The clock offset measurements show the time difference between the reference clock and the local clock. The clock phase measurements show the phase difference between the reference clock and the local clock, that is, when each second is accounted. All of the presented plots and their respective source values can be found on the CD attached to this thesis. The table of CD contents is listed in appendix E.

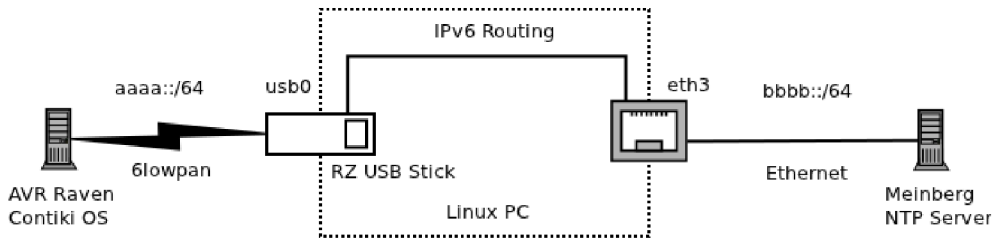


Figure 7.1: Network topology for measurements

Figure 7.1 shows the network topology used for the measurements. The reference clock for all of the presented measurements was the Meinberg M600 NTP stratum 1 server, which was synchronised using the GPS. The NTP client was running on the AVR Raven platform in a room at 23 °C.

7.1 Clock interrupt frequency

The AVR Raven’s bit 7 of Port D and the ground pin were connected to the UNI-T 2025CEL digital oscilloscope to measure the clock interrupt frequency. At the beginning of the interrupt service routine a logic 1 was written to the bit 7 of Port D, what caused a high level of voltage. At the end of the interrupt service routine a logic 0 was written to the bit 7 of Port D, what caused a low level of voltage.

When there are no clock adjustments in progress, the value of the output compare register is 31 by default. The clock interrupt frequency is supposed to be equal to the value of the *CLOCK_SECOND* macro, which is 128 by default on AVR Raven and is given by

formula 7.1. Figure B.1 in appendix B shows the oscilloscope output for this case.

$$\frac{\frac{f_{asy}}{prescaler}}{counts} = \frac{\frac{32768}{8}}{32} = 128 \quad (7.1)$$

Figure B.2 in appendix B shows the oscilloscope output when slowing down the clock. The clock interrupt frequency is supposed to be equal to $124.\overline{12}$ Hz and is given by formula 7.2.

$$\frac{\frac{f_{asy}}{prescaler}}{counts + 1} = \frac{\frac{32768}{8}}{32 + 1} = 124.\overline{12} \quad (7.2)$$

Figure B.3 in appendix B shows the oscilloscope output when speeding up the clock. The clock interrupt frequency is supposed to be approximately equal to 132.129 Hz and is given by formula 7.3.

$$\frac{\frac{f_{asy}}{prescaler}}{counts - 1} = \frac{\frac{32768}{8}}{32 - 1} \doteq 132.129 \quad (7.3)$$

The measured values are not exactly equal to those expected. This is mostly because of a room temperature influence on the clock source (32 768 Hz quartz crystal oscillator), but it could also be air pressure or magnetic fields, etc.

7.2 Clock offset

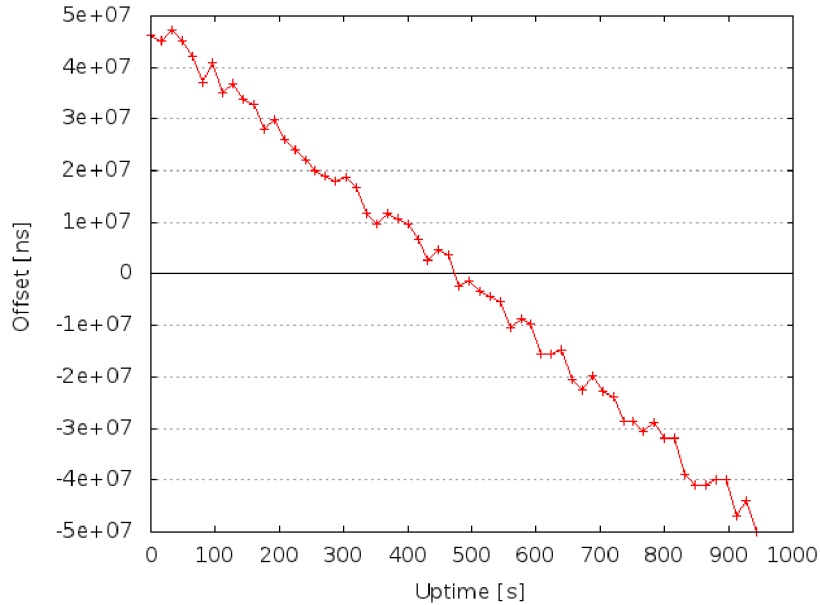


Figure 7.2: Local clock offset without NTP client

Figure 7.2 shows the local clock offset in case no NTP client runs on the device. The time is set with the initial offset of about 45 milliseconds. However, the clock progresses faster because of frequency errors. This compensates the initial clock offset at first, but then it causes the offset increase. The clock is running faster with approximately 100 PPM, where 1 PPM is equal to $10^{-6} \frac{s}{s}$ (0.0001%). That is, the clock error is about 9 seconds a day in

this case. The offset increase is not exactly linear because of the frequency jitter, which can be also observed.

Figure 7.3 shows the local clock offset acquired from the serial output in case the Contiki NTP client runs on the device. When the developed NTP client receives a response from the server, it calculates the local clock offset and prints its value to the serial output. The NTP poll interval was set to 16 seconds, that means, the local clock offset is calculated and eventually corrected every 16 seconds.

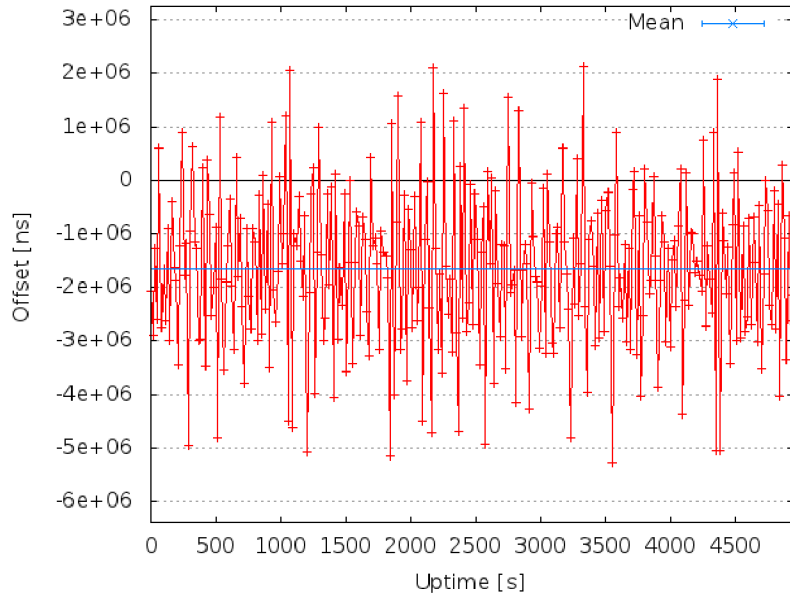


Figure 7.3: Local clock offset with adjustments and NTP poll interval 16s

The blue line shows the mean local clock offset value, which should be equal to zero in a perfect case. This is however not the case, because of the oscillator frequency error shown in figure 7.2. More figures showing the local clock offset measurements can be found in appendix C.

7.3 Clock phase

The GPS based clock Meinberg GPS 167 and digital oscilloscope UNI-T 2025CEL were used for measuring the clock phase difference. Meinberg GPS clock rises an impulse when each second is accounted. Contiki on AVR Raven was configured to write a logic 1 to bit 7 of Port D when each second is accounted and to write a logic 0 to the same bit after 25 clock ticks.

When the NTP client uses the *clock_adjust_time* call, the local clock offset as well as the phase is being adjusted. Figure 7.4 shows the phase while adjusting the clock. The yellow line is the output signal from Meinberg GPS clock and the blue line is the output signal from AVR Raven.

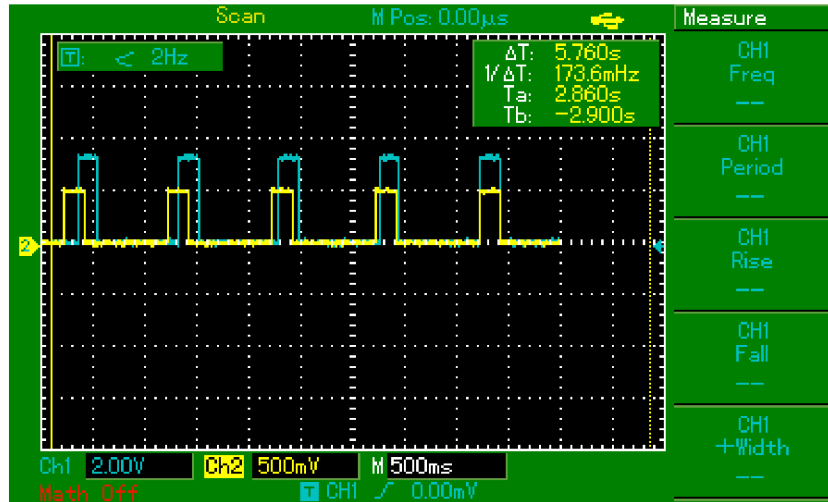


Figure 7.4: Second impulses when clock adjustments are in progress

Figures showing the clock out of phase and in phase with the reference clock can be found in appendix D.

Chapter 8

Conclusion

As of early 2012, the Network Time Protocol has been operating for over 30 years and remains the longest running, continuously operating application protocol in the Internet [25]. This thesis demonstrates, that its use can be further extended to a new platform of constrained devices, such as embedded systems.

The developed time interface extends Contiki OS towards the real-time support, while requiring minimal memory amounts. This interface provides new calls to get, set and adjust the system time without any modification of the existing code. The clock library was extended to provide a platform-agnostic hardware clock interface with a minimal memory overhead. The time interface could therefore be ported to different hardware platforms easily. The call for getting the time provides the maximum precision the hardware clock allows. Setting the time is only possible within the precision of one second in order to avoid the misbehaviour of the existing timer libraries. The time adjustments, that require sophisticated clock discipline algorithms, were not implemented because of complicated design and memory constraints.

The developed NTP client for Contiki OS uses the developed time interface to keep an accurate time on the device. The Contiki NTP client is able to use the NTP unicast and broadcast modes. The unique timestamp conversion provides a portable solution to avoid memory expensive floating point or 64-bit arithmetic operations in a constrained environment. The measurements show that the Contiki NTP client provides an accurate time synchronisation mechanism to the Contiki operating system. The Contiki NTP client is a Simple Network Time Protocol client, that can send requests to only one specified NTP server. NTP algorithms for the time determination from more server responses were therefore not implemented.

The Network Time Protocol is suitable for embedded systems and constrained devices, forming the modern Internet and it will arguably find its place among them in the near future.

Bibliography

- [1] Internet Assigned Numbers Authority. Service Name and Transport Protocol Port Number Registry [online]. <http://www.iana.org/assignments/port-numbers>, 2012 [cit. 2012-01-23].
- [2] Atmel Corporation. AVR Instruction Set manual. www.atmel.com/Images/doc0856.pdf, 2010.
- [3] Atmel Corporation. Datasheet ATmega164A/PA/324A/PA/644A/PA/1284/P Complete. www.atmel.com/Images/doc8272.pdf, 2011.
- [4] A. Dunkels. Towards TCP/IP for Wireless Sensor Networks. Master's thesis, Swedish Institute of Computer Science, Stockholm, Sweden, 2005. <http://www.sics.se/~adam/dunkels05towards.pdf>.
- [5] A. Dunkels. *Programming Memory-Constrained Networked Embedded Systems*. PhD thesis, Swedish Institute of Computer Science, Stockholm, Sweden, 2007. <http://www.sics.se/~adam/dunkels07programming.pdf>.
- [6] A. Dunkels. Poster Abstract Rime A Lightweight Layered Communication Stack for Sensor Networks [online]. <http://www.sics.se/~adam/dunkels07rime.pdf>, 2007 [cit. 2012-01-24].
- [7] A. Dunkels. Contiki OS Documentation, 2011. Documentation can be generated from the Contiki source code.
- [8] A. Dunkels. Contiki OS Wiki - Timers [online]. <http://www.sics.se/contiki/wiki/index.php/Timers>, 2011 [cit. 2012-04-13].
- [9] A. Dunkels. Contiki OS Wiki - Processes [online]. <http://www.sics.se/contiki/wiki/index.php/Processes>, 2011 [cit. 2012-04-23].
- [10] A. Dunkels. Protothreads - Lightweight, Stackless Threads in C [online]. <http://dunkels.com/adam/pt/>, 2011 [cit. 2012-06-04].
- [11] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors [online]. <http://www.sics.se/~adam/dunkels04contiki.pdf>, 2004 [cit. 2012-01-28].
- [12] A. Dunkels, D. Kopf, and R. Wohlers. Contiki OS Wiki - FAQ [online]. <http://www.sics.se/contiki/wiki/index.php/FAQ>, 2011 [cit. 2012-01-20].

- [13] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems [online]. <http://www.sics.se/~adam/dunkels06protothreads.pdf>, 2006 [cit. 2012-01-19].
- [14] B. Esham. Network Time Protocol servers and clients. http://commons.wikimedia.org/wiki/File:Network_Time_Protocol_servers_and_clients.svg, 2007 [cit. 2012-01-31].
- [15] D. Exb. Marzullo example 1. http://en.wikipedia.org/wiki/File:Marzullo_example-1.jpg, 2007 [cit. 2012-02-02].
- [16] IPv6 Forum. IPv6 Ready Logo Program Approved List [online]. <https://www.ipv6ready.org/db/index.php/public>, 2012 [cit. 2012-01-27].
- [17] British Standards Institute. *The C Standard: Incorporating Technical Corrigendum 1*. John Wiley & Sons Publishers, 2002. ISBN 978-0-47-084573-8.
- [18] D. Mills. Improved Algorithms for Synchronizing Computer Network Clocks. *ACM SIGCOMM Computer Communication Review*, 24, October 1994. <http://dl.acm.org/citation.cfm?id=190314.190343>.
- [19] D. Mills. A Brief History of NTP Time: Confessions of an Internet Timekeeper. Technical report, Electrical and Computer Engineering Department, University of Delaware, Newark, USA, 2003. <http://www.eecis.udel.edu/~mills/database/papers/history.pdf>.
- [20] D. Mills. Service Name and Transport Protocol Port Number Registry [online]. <http://www.eecis.udel.edu/~mills/database/brief/overview/overview.pdf>, 2004 [cit. 2012-01-29].
- [21] D. Mills. NTP Clock Discipline Modelling and Analysis [online]. <http://www.eecis.udel.edu/~mills/database/brief/algor/algor.pdf>, 2005 [cit. 2012-02-02].
- [22] D. Mills. NTP Architecture, Protocol and Algorithms [online]. <http://www.eecis.udel.edu/~mills/database/brief/arch/arch.pdf>, 2007 [cit. 2012-02-03].
- [23] D. Mills. The NTP Timescale and Leap Seconds [online]. <http://www.eecis.udel.edu/~mills/leap.html>, 2012 [cit. 2012-02-01].
- [24] D. Mills. Precision Time Synchronization [online]. <http://www.eecis.udel.edu/~mills/precision.html>, 2012 [cit. 2012-05-10].
- [25] D. Mills. The NTP Era and Era Numbering [online]. <http://www.eecis.udel.edu/~mills/y2k.html>, 2012 [cit. 2012-26-06].
- [26] D. Mills, J. Martin, J. Burbank, and W. Kasch. RFC 5905: Network Time Protocol Version 4: Protocol and Algorithms Specification. <http://www.rfc-editor.org/rfc/rfc5905.txt>, 2010.

- [27] F. Moradi and A. Javaheri. Clock Synchronization in Sensor Networks for Civil Security. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 2009. <http://publications.lib.chalmers.se/records/fulltext/112022.pdf>.
- [28] International Bureau of Weights and Measures. Coordinated Universal Time [online]. http://www.bipm.org/en/scientific/tai/time_server.html, 2011 [cit. 2012-02-01].
- [29] IEEE Computer Society and Open Group. IEEE Std 1003.1-2008 (POSIX.1-2008) [online]. <http://pubs.opengroup.org/onlinepubs/9699919799/>, 2008 [cit. 2012-02-01].
- [30] J. Vasseur and A. Dunkels. *Interconnecting Smarts Objects with IP: The Next Internet*. Morgan Kaufmann Publishers, 2010. ISBN 978-0-12-375165-2.
- [31] U. Windl, D. Dalton, M. Martinec, D. Worley, et al. The NTP FAQ and HOWTO [online]. <http://www.ntp.org/ntpfaq/>, 2006 [cit. 2012-06-16].

Appendix A

Protothreads Example

Listing A.1 shows an example of delaying text on an LCD panel using Protothreads. It is taken from Adam Dunkels' Protothreads website [10] and slightly modified. Protothreads can be used to introduce delays inside a function without using a full threading model. The example shows a function writing characters to an LCD panel. Suppose that each character is shown for one second, then the next character replaces the previous.

```
1 #include "pt.h"
2 #include "timer.h"
3 #include <string.h>
4
5 typedef unsigned short lc_t;
6 struct pt {
7     lc_t lc; /* Local continuation */
8 };
9
10 struct pt state;
11 struct timer timer;
12
13 PT_THREAD(display_text(struct pt *pt, const char *msg))
14 {
15     PT_BEGIN(pt);
16     for (int i = 0; i < strlen(msg); i++) {
17         lcd_display_char(msg[i]);
18         timer_set(&timer, CLOCKSECOND); /* Wait for one second. */
19         PT_WAIT_UNTIL(pt, timer_expired(&timer));
20     }
21     PT_END(pt);
22 }
23
24 int main(void)
25 {
26     PT_INIT(&state);
27     for (;;) {
28         display_text(&state, "Hello world");
29         /* Here could be another thread run */
30     }
31     return 0;
32 }
```

Listing A.1: Example using Protothreads

The `PT_WAIT_UNTIL` macro actually causes the function to return. While the function is waiting for the timer to expire another function could be called and run. When the function

is entered again, the execution continues with the `PT_WAIT_UNTIL` macro which causes the function to check the condition it is waiting for (timer expired). If the condition is met, the function resumes, and it returns again if not. Strictly speaking, the amount of time between showing each character can be more than one second. This is because Protothreads are not running simultaneously: if the timer expired and another Protothread was running, this Protothread would have to wait until it is entered again. When the condition specified in `PT_WAIT_UNTIL` is met, the next iteration of the *for* loop (line 16) is started and the next character is displayed.

How does it work? The macro `PT_BEGIN` is expanded to a *switch* statement by the preprocessor. The `PT_WAIT_UNTIL` macro expands to *case* and setting the local continuation to the value, so that the next time this function is run, it jumps to this *case*. The structure holding the state is defined outside of the function, so its context is not lost when the function returns. The simplest state structure would hold just the local continuation variable. Listing A.2 shows the same example after a simplified preprocessing.

```

1 #include "pt.h"
2 #include "timer.h"
3 #include <string.h>
4
5 typedef unsigned short lc_t;
6 struct pt {
7     lc_t lc;                               /* Local continuation */
8 };
9
10 struct pt state;
11 struct timer timer;
12
13 int display_text(struct pt *pt, const char *msg) /* Expanded PT_THREAD */
14 {
15     switch(pt->lc) { case 0:                /* Expanded PT_BEGIN(pt); */
16         for (int i = 0; i < strlen(msg); i++) {
17             lcd_display_char(msg[i]);
18             timer_set(&timer, CLOCK_SECOND); /* Wait for one second. */
19
20             /* The following two lines are expanded */
21             pt->lc = 31; case 31: /* PT_WAIT_UNTIL(pt, timer_expired(&timer)); */
22             if(!(timer_expired(&timer))) { return PT_WAITING; } /* macro */
23
24         }
25     pt->lc = 0; return PT_ENDED; } /* Expanded PT_END */
26 }
27
28 int main(void)
29 {
30     state->lc = 0; /* Expanded PT_INIT */
31     for (;;) {
32         display_text(&state, "Hello world");
33         /* Here could be another thread run */
34     }
35     return 0;
36 }

```

Listing A.2: Preprocessed example using Protothreads

Appendix B

Clock Interrupt Frequency Measurements

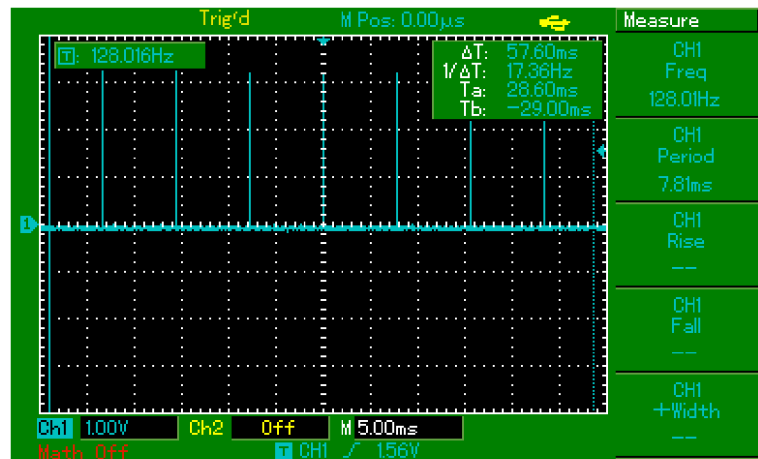


Figure B.1: Interrupt frequency without clock adjustment

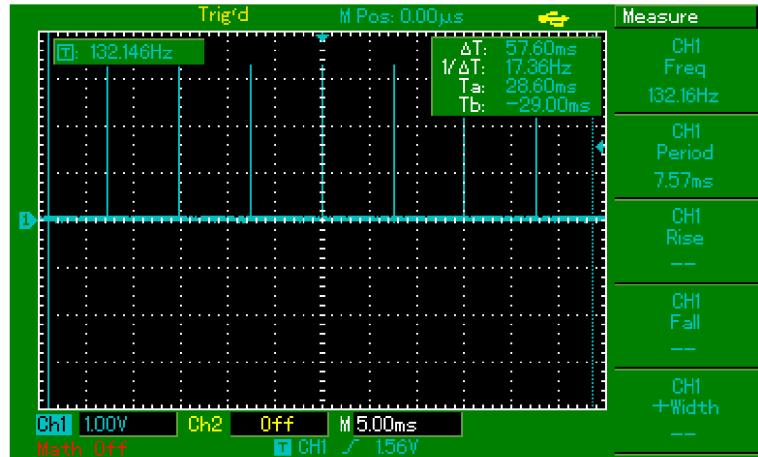


Figure B.2: Interrupt frequency when speeding up the clock

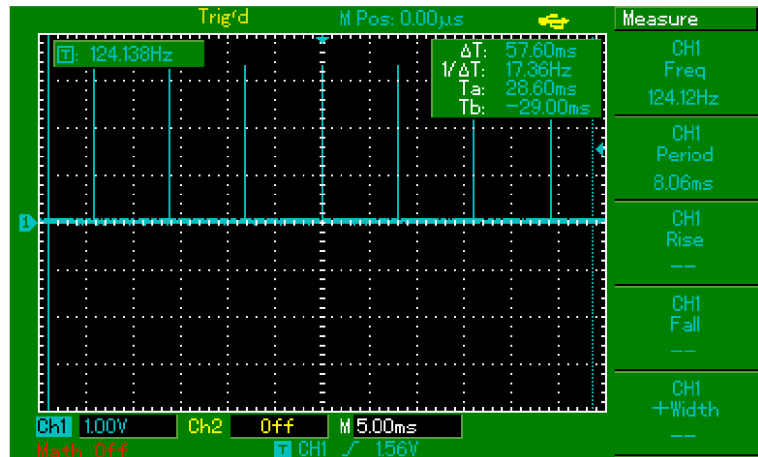


Figure B.3: Interrupt frequency when slowing down the clock

Appendix C

Clock Offset Measurements

The serial output from AVR Raven, the TechTools DigiView DV3100 logic analyser and the Meinberg GPS 167 clock were used for measuring the local clock offset. Meinberg GPS 167 rises an impulse when each second is accounted. AVR Raven was configured to write a logic 1 to bit 7 of Port D when each second is accounted, and to write a logic 0 to the same bit after 25 clock ticks.

Figure C.1 shows the local clock offset from long-term uptime observation when NTP client sets the time, but no adjustments are applied. A linear offset increase can be observed, until the threshold value for setting the time is reached. The offset threshold value was 3 seconds.

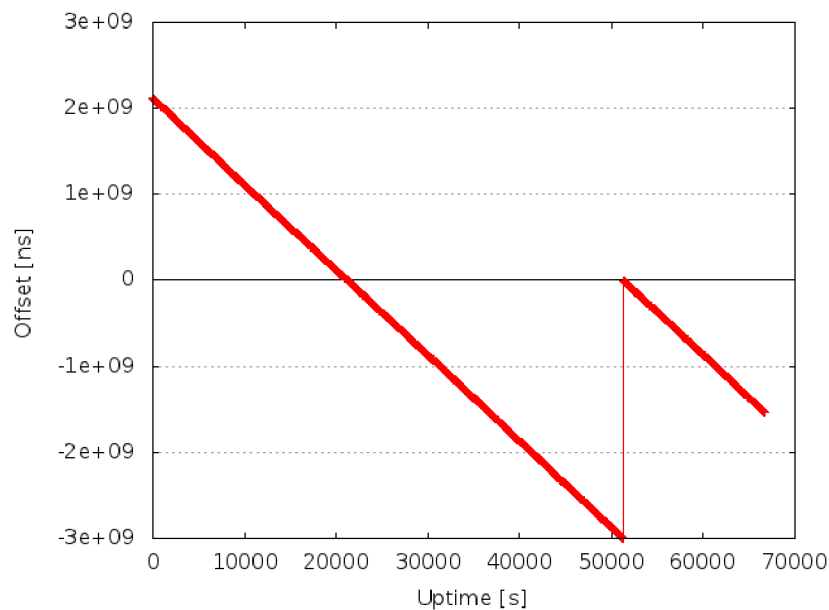


Figure C.1: Local clock offset with NTP client setting the time but without adjustments

Figure C.2 shows the local clock offset acquired from logic analyser when the Contiki NTP Client runs on the device. The logic analyser captures the rising and the falling edge from the Meinberg GPS clock and from AVR Raven. The difference between the rising edge from Meinberg GPS clock and the rising edge from AVR Raven gives the local clock offset.

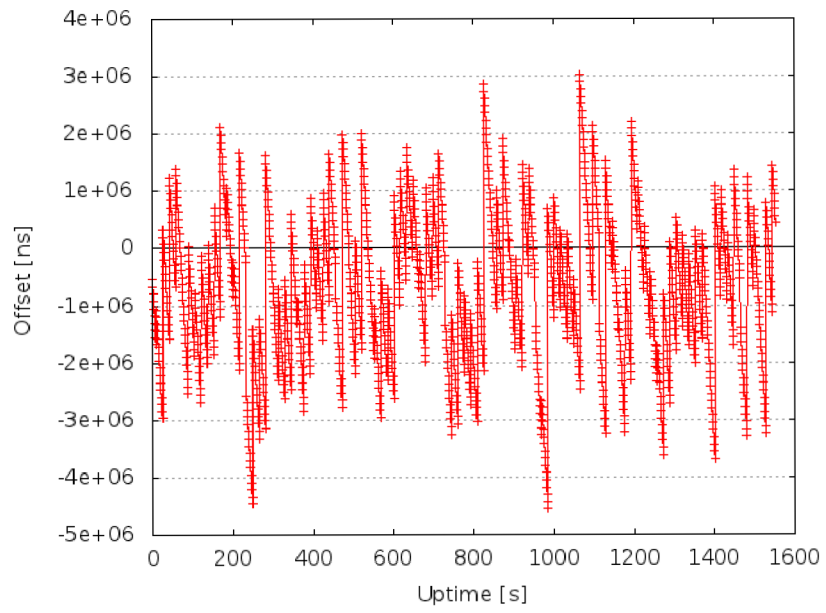


Figure C.2: Local clock offset with adjustments and NTP poll interval 16s

Appendix D

Clock Phase Measurements

The following figures show the phase difference between the reference clock (Meinberg GPS 167) and AVR Raven. The figures were acquired from the UNI-T 2025CEL digital oscilloscope. The yellow line on channel 2 shows the impulses from the GPS synchronised clock Meinberg GPS 167. The blue line on channel 1 shows the impulses from AVR Raven running Contiki OS with the developed NTP client. The rising edge occurs when each second is being accounted. Figure D.1 shows the clock out of phase and figure D.2 shows the clock in phase.

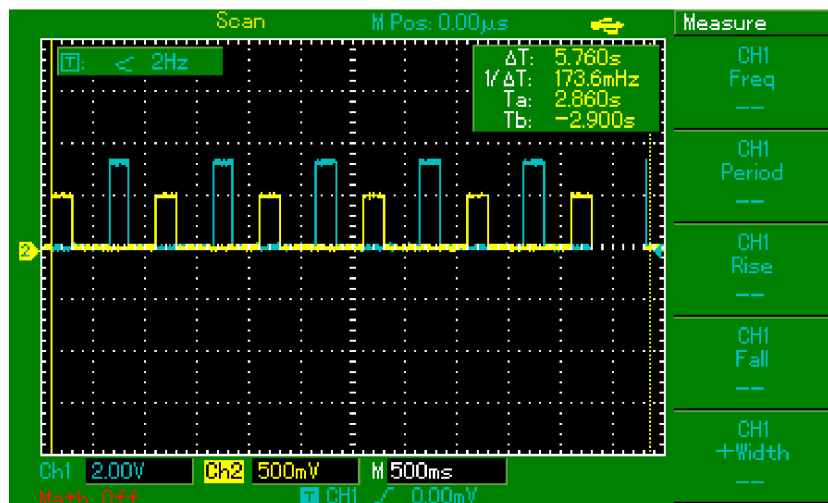


Figure D.1: Second impulses when the clock is out of phase

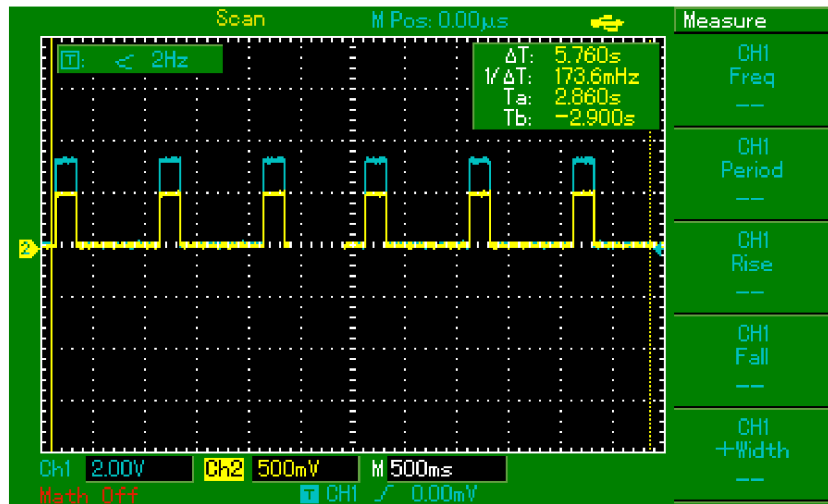


Figure D.2: Second impulses when the clock is in phase

Appendix E

CD Contents

Directory	Contents
docs/	Setup guides for running and debugging Contiki OS on AVR Raven
docs/datasheets/	Datasheets for used hardware
docs/img/	Images from Dopsy group laboratory at RheinMain University
sw/	Software used in this thesis including Contiki source code and patches
sw/ntpd/	Contiki NTP Client source code
sw/utills/plots/	Scripts and measured results for plots
sw/utills/tests/	Programs used for testing Contiki NTP Client
text/	LaTeX source files of this thesis
text/fig/	Figures used in this thesis