Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Andreas Moser

Linux Kernel für hochpräzise Uhrensynchronisation

Student Thesis SA-2002.29
Summer Term 2002

Tutor: Philipp Blum

Supervisor:
Prof. Dr. Lothar Thiele

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Inhaltsverzeichnis

Abbildungsverzeichnis

Vorwort

Steckt man sich zum Ziel, mehrere verteilte Uhren in einem Rechnernetz sehr präzise zu synchronisieren, stösst man schnell an seine Grenzen. Die unterschiedlichen Laufzeiten und ganz allgemein das reale physikalische Medium sind Dinge, die man nur begrenzt in den Griff bekommen kann.

In Zusammenarbeit mit der Firma BridgeCo AG in Dübendorf[1] hat das Institut für Technische Informatik und Kommunikationsnetze der ETH Zürich (TIK) eine Arbeit ausgeschrieben, die zum Ziel hat, verteilte Uhren über Wireless LAN (WLAN) hochpräzise zu synchronisieren. Im Vorfeld wurde bereits ein Algorithmus entwickelt, der in diesem Rahmen implementiert werden sollte. Daniel Sigg und Eric Schreiber nahmen diese Aufgabe als Diplomarbeit in Angriff. Das Ziel meiner Semesterarbeit war es, für diese Uhrensynchronisation die Infrastruktur unter dem Betriebssystem Linux zur Verfügung zu stellen. Dazu gehörte das Design, die Implementierung und die Validierung eines Uhrenobjekts, das im Linux-Kernel platziert wird, um diesen hohen Anforderungen gerecht werden zu können.

Da diese beiden Arbeiten gleichzeitig stattfanden, konnte der gesamte Entwicklungsprozess gemeinsam abgesprochen und laufend angepasst werden.

[1] www.bridgeco.net

Aufgabenstellung

Kapitel 1

Theoretische Grundlagen

In diesem Kapitel werden verschiedene theoretische Grundlagen vermittelt, welche nötig sind für das Verständnis dieser Semesterarbeit. Der erste Abschnitt (1.1) beantwortet die Frage, wofür hochpräzise Uhren überhaupt nötig sind. Der zweite Abschnitt (1.2) macht eine kurze Einführung in das freie Betriebssystem Linux. Im dritten (1.3) geht es dann um die zur Verfügung stehenden Hardware-Ressourcen bezüglich Zeitmessung, während der vierte und fünfte Abschnitt wesentliche Prinzipien der Betriebssystem-Architektur, wie Kernel- und Userspace (1.4) und System Calls (1.2) thematisieren. Der letzte Abschnitt (1.6) zeigt die Möglichkeit auf, dass ein Stück Code auf verschiedene Arten in den Kernel integriert werden kann.

1.1 Warum hochpräzise Uhren?

Nachdem im Vorwort der Hintergrund dieser Semesterarbeit kurz dargelegt wurde, ist die Frage bislang noch nicht geklärt worden, wofür denn derart präzise Uhren überhaupt nötig seien?

Bei verteilten Systemen ist es in vielen Anwendungsfällen nötig, dass deren Knoten über eine synchronisierte Zeit verfügen, da sie in der Lage sein müssen, Abläufe koordiniert auszulösen oder zu erfassen. Insbesondere im Multimedia-Bereich muss diese Synchronisation speziell hohen Anforderungen genügen. Als Beispiel ist das menschliche Ohr zu nennen, das schon auf kleinste Zeitverschiebungen zwischen Signalen sehr empfindlich reagieren kann. Es liegt also auf der Hand, dass ein Synchronisationsalgorithmus diese Aufgabe übernehmen muss und somit auch über entsprechende Uhren verfügen muss.

1.2 Linux

Linux ist mehr als ein Betriebssystem. Es ist eine Weltanschauung. Hinter Linux steckt eine Philosophie, die vielen sehr fremd ist. Die Idee, dass Linux nicht kom-

merziell vermarktet werden darf und der gesamte Quellcode[1] völlig offen einsehbar ist, ist schon revolutionär. Linus Torvalds, der Vater von Linux, hatte sich nämlich anfangs der 90er entschieden, seine stark an Minix[2] anlehnende Errungenschaft unter die GNU[3] Public License[4] zu stellen. Diese verbietet es, kommerziellen Nutzen aus dem Verkauf des ihm unterstellten Produktes zu ziehen. Und obwohl ihm damals von namhaften Stellen nicht die geringste Chance prophezeit wurde, konnte sich Linux immer mehr durchsetzen, bis es schliesslich mit dem GNU Projekt zusammenwuchs. Heute ist GNU/Linux eine echte Alternative zu den omnipräsenten Microsoft-Produkten. Dies hat viele Gründe und soll an dieser Stelle nicht weiter diskutiert werden. Eine detaillierte Analyse über die Open-Source Bewegung ist zu finden unter www.rs3.ch/~roman/projects/opensource. Es handelt sich dabei um eine Arbeit, die von einem meiner Kommilitonen im Rahmen der ergänzenden Vorlesung *Sozioligie* an der ETH geschrieben wurde.

1.3 TSC und PIT

Der Linux Kernel hat Zugriff auf drei Uhren der Hardware: die Real Time Clock, kurz *RTC*, den Time Stamp Counter oder nur *TSC*, und den Programmable Interval Timer, der mit *PIT* abgekürzt wird. Davon benützt der Kernel die ersten zwei, um die aktuelle Tageszeit nachzuführen. Der PIT wird vom Kernel so programmiert, dass er mit vorgegebener, konstanter Frequenz Interrupts generiert. Diese periodischen Interrupts sind von grosser Bedeutung für die Implementierung von Timern, sowohl vom Kernel- als auch vom Userspace. Für diese Arbeit stand aber der TSC im Vordergrund, weshalb dieser nun etwas genauer erläutert werden soll.

Alle Intel 80x86 Prozessoren besitzen einen CLK[5] Input Pin, der das periodisch schwingende Signal von einem externen Oszillator bezieht. Mit speziellem Augenmerk auf die Pentium-Prozessoren ist zu sagen, dass die meisten neuen Modelle auf dem Markt ein 64-bit breites TSC-Register haben, dessen Wert mit jedem Clock-Zyklus um 1 inkrementiert wird. Ist also ein Prozessor mit einer Frequenz von 800 MHz getaktet, so wird der Wert des TSC Registers alle 1.25 Nanosekunden um 1 erhöht. Im Linux-Kernel steht die Assembler-Instruktion `rdtsc` zur Verfügung, um diesen auszulesen. Linux nutzt dies aus, um eine viel höhere Genauigkeit von Timer-Intervallen zu erreichen als es mit dem PIT alleine möglich wäre. Dazu muss aber der Kernel zuerst einmal die Clock Frequenz genau bestimmen, da dies zur Zeit seiner Kompilierung[6] noch nicht bekannt ist. Er tut dies beim Booten des Systems, indem die Anzahl Clock-Signale während einer Zeit von 50.00077 Millisekunden gezählt werden.

[1] Im folgenden auch oft nur kurz “Code” genannt
[2] Eine Minimalversion von UNIX
[3] steht für GNU is Not UNIX
[4] kurz GPL
[5] clock
[6] *Kompilieren* nennt man den Vorgang des Übersetzens von menschlich lesbarem Code zu binärem Code, der vom Prozessor ausgeführt werden kann.

Für die vorliegende Arbeit entpuppte sich das Auslesen des TSC-Registerwertes mittels obig erwähnter Instruktion *rdtsc* als sehr nützlich. Stellt diese doch ein Mittel zur Verfügung, mithilfe dessen die Dauer von kurzen Abläufen fast auf die Nanosekunde genau gemessen werden können.

1.4 Kernel- und Userspace

Eine Charakteristik von UNIX, Linux und ähnlichen Betriebssystemen ist die klare Unterscheidung zwischen dem Kernel- und dem Userspace. *Kernel* ist eine Bezeichung für den eigentlichen Betriebssystemkern. Dieser stellt quasi ein Allerheiligstes dar, weil dort die zeit-, hardware-, I/O- und speicherkritischen Abläufe geschehen. Also müssen diese heiklen Angelegenheiten vor unbedachtem oder unsachgemässem Zugriff geschützt werden. Dies geschieht, indem alle Prozesse[7] im sogenannten *Userspace* ablaufen, wo sie keine betriebssystemrelevanten Privilegien geniessen und somit auch keinen Schaden anrichten können. Eine grobe Übersicht über diese Struktur ist Abb. 1.1 zu entnehmen.

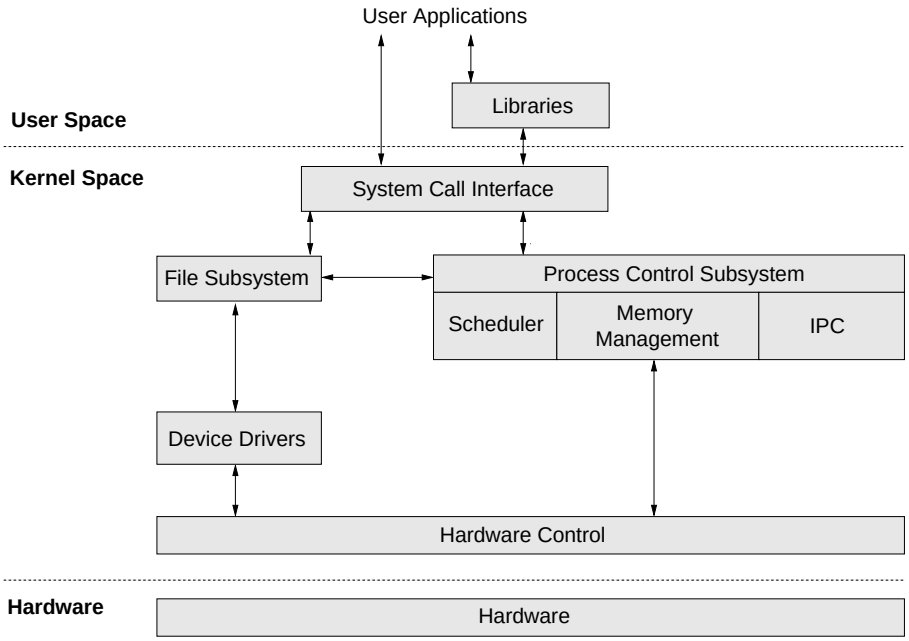

Abbildung 1.1: Struktur des Linux-Kernels

Aus dieser Architektur folgen einige wichtige Dinge. So muss Linux fähig sein,

[7]Prozesse sind programmierte Codestücke, die vom Prozessor sequentiell abgearbeitet werden. Somit sind alle Programme wie z.B. ein Browser, ein Texteditor oder das Helloworld-Programm Prozesse aus Betriebssystemsicht.

mehrere Prozesse "gleichzeitig"[8] im Userspace auszuführen. Dies macht ein ausgeklügeltes Prozess-Management erforderlich. Für diese Aufgabe wird ein sogenannter *Scheduler*[9] eingesetzt, der grundsätzlich nach verschiedenen Prinzipien die wertvolle Rechenzeit verteilen kann. Vereinfacht gesagt geschieht dies, indem er in regelmässigen Abständen alle momentan ablaufenden Prozesse überschaut und ihnen nach bestimmten Kriterien eine bestimmte Rechenzeit zuteilt. Bereits an dieser Stelle wird klar, dass ein Prozess im Userspace immer beschränkte Ressourcen zur Verfügung hat und deshalb auch mehr oder weniger grossen zeitlichen Verzögerungen unterlegen ist. Wohingegen im Kernelspace keine Prozesse existieren und somit auch der Scheduler hinfällig ist. Läuft also ein bestimmtes Stück Code im Kernelmode[10] ab, so kann es höchstens durch Timer-Interrupts und ähnliches unterbrochen werden. Und dies ist genau das, was für eine hochpräzise Uhr dringend erforderlich ist. Es ist unmöglich, eine Softwareuhr mit extrem hohen Anforderungen im Userspace zu implementieren, wenn nur schon ein einfacher Lesezugriff durch diverse Mechanismen verzögert werden kann. Dazu kommt, dass die Uhr ja Teil eines *verteilten* Dienstes ist, dessen Kommunikation über die *wireless* Schnittstelle läuft. Somit gehen sowohl beim Server als auch beim Client ein- und ausgehende Pakete über das entsprechende *wireless Netzwerk-Interface*, was natürlich sehr Hardware-nah ist und somit direkt vom Kernel ausgeführt werden muss.

Es ist also aus diesen Gründen leicht ersichtlich, dass eine Softwareuhr, die extremen Anforderungen genügen soll, unbedingt im Kernelspace platziert werden muss. Wie dies im Detail geschieht und welche weiteren Designentscheide noch getroffen werden mussten, entnehme man dem Kapitel 2.

1.5 System Calls

Dem geneigten Leser wird schon aufgefallen sein, dass ein im Userspace ablaufender Prozess sicherlich auf dem einen oder anderen Weg Zugriff auf z.B. Hardware-Ressourcen erhalten muss. Wie sonst könnte ein Texteditor eine bearbeitete Datei auf der Harddisk abspeichern? Das Prinzip vom klar getrennten Kernel und Userspace verlangt nun aber, dass nur der erstere unmittelbare Interaktionen mit der besagten Harddisk unternehmen darf. Also muss es einen Mechanismus geben, der den Prozessen über den Weg des Kernelspaces einen entsprechenden Dienst zur Verfügung stellt. Dies geschieht mittels der sogenannten *System Calls*. Sie stellen ein Interface dar zwischen User- und Kernelspace. Das Prinzip der System Calls hat drei grosse Vorteile: erstens macht es die Programmierarbeit leichter und bewahrt den Benutzer davor, tiefe Systemkenntnisse erarbeiten zu müssen für einen "einfachen" Lese- oder Schreibzugriff. Zweitens macht es die Zugriffe sicher, indem der Kernel jeden System Call Aufruf erst auf Gültigkeit und Sicherheit überprüft.

[8] selbstverständlich führt ein einzelner Prozessor zu einer bestimmten Zeit tatsächlich nur eine Codesequenz aus.

[9] engl.: to schedule=einteilen, disponieren

[10] Die Begriffe Kernelmode und Kernelspace entsprechen sich genauso, wie Usermode und Userspace.

Als dritter und letzter Vorteil ist zu erwähnen, dass Programme, die System Calls verwenden, sehr einfach auf andere Systeme portierbar sind, da sie auf irgendeinem Kernel kompiliert und ausgeführt werden können, die denselben Dienst anbieten.

Wie also funktioniert ein System Call nun? Abbildung 1.2 bietet dafür eine gute Übersicht. Der Programmierer ruft in seiner Userspace-Applikation nicht etwa direkt eine Kernelfunktion auf, sondern eine *wrapper routine*[11] der Standard C-Bibliothek, der *libc*. Diese überprüft, wie oben erwähnt, den Aufruf auf Gültigkeit und Sicherheit. Ausserdem bereitet sie die Übergabeparameter an den System Call so auf, dass sie für den Kernelspace auch tatsächlich zur Verfügung stehen. Nachdem nun alles bereit ist, kann ein sogenannter *trap* ausgelöst werden, womit die Grenze zwischen User- und Kernelmode überschritten wird. Da es viele verschiedene System Calls gibt, muss eine sogenannte *System Call Number* mitgegeben werden, um den entsprechenden System Call zu identifizieren. Als erstes wird im Kernelmode nun der *System Call Handler* ausgeführt. Dieser funktioniert im wesentlichen wie alle Exception Handler und speichert als erstes den Inhalt der meisten Register im Kernelmode Stack. Danach kommt der wichtigste Teil: die *System Call Service Routine* wird aufgerufen. Hier wird nun endlich das getan, was der Benutzer mit seinem System Call Aufruf tatsächlich beabsichtigte. Der Name der Service Routine, die zum System Call *xyz* gehört, ist normalerweise *sys_xyz*. Es gibt nur einige wenige Ausnahmen zu dieser Regel.

Nachdem nun *sys_xyz* die gewünschten Aktionen durchgeführt hat, beginnt der Rückweg in den Userspace. Dieser kann in groben Zügen der Abbildung 1.2 entnommen werden und soll hier nicht detailliert behandelt werden. Wichtig ist nur zu wissen, dass der Rückgabewert, den der Programmierer sieht, immer nur eine `integer` Variable ist, die etwas über den Erfolg der System Call Ausführung aussagt. Mögliche Resultatwerte werden also nicht auf diesem Weg zurückgegeben. Stattdessen wird dem System Call ein oder mehrere Pointer mitgegeben, die auf den Speicherbereich zeigen, wo die Resultatwerte vom Kernelspace hingeschrieben werden können.

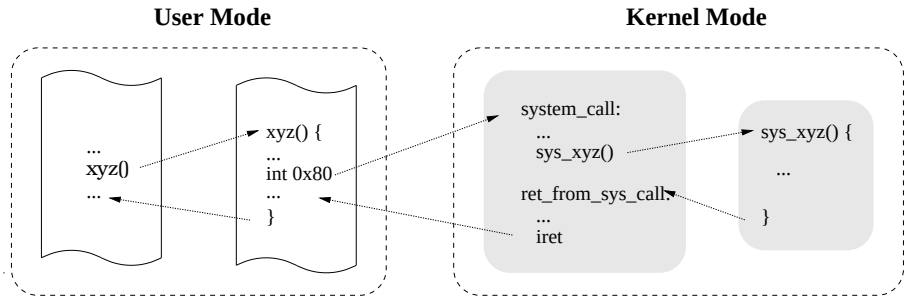

Abbildung 1.2: Aufruf eines System Calls

[11] von engl. to wrap=einwickeln, einpacken

1.6 Kernel-Modul oder kein Kernel-Modul?

Jeder System-Programmierer, der eigenen Code in den Kernel integrieren möchte, kann dies auf zwei verschiedene Arten tun: entweder wird der Code so geschrieben, dass er als sogenanntes *Kernel-Modul* kompiliert wird, oder er wird statisch in den Kernel gelinkt. Dies kommt daher, dass in der Programmiersprache C, in welcher der Kernel nämlich geschrieben ist, verschiedene Programmstücke einzeln kompiliert werden können. Die daraus resultierenden, sogenannten *Object Files* mit der File Extension "*.o*" können sodann durch den *Link-Vorgang* zu einem einzigen, ausführbaren Programm zusammengefügt werden. Erstaunlicherweise ist es möglich, solche Object Files in einen *laufenden* Kernel einzufügen, also zur Laufzeit dynamisch zu linken. Die Entscheidung des Programmierers besteht also darin, ob er den kompilierten Code als Object File zur Verfügung haben will, den er dann in den Kernel einfügen kann, wann es erforderlich ist, oder ob der Code statisch in den Kernel gelinkt wird.

Kapitel 2

Implementation

2.1 Anforderungen an das Uhrenobjekt

Die Anforderungen an das Uhrenobjekt sehen wie folgt aus:

- Es soll mehrere Uhren zur Verfügung stellen, welche die Hardware nicht manipulieren und die einfach verwaltet werden können.
- Es soll den Synchronisationsalgorithmus sauber von den Applikationen, die auf die Uhren zugreifen, trennen.
- Es sollen sowohl zum User- als auch zum Kernelspace flexible Interfaces implementiert werden.

2.2 Designentscheid: Uhrenobjekt als Kernel Modul

In Abschnitt 1.4 wurde bereits gezeigt, dass die Software-Uhr unbedingt im Kernelspace platziert werden muss. Und aus Abschnitt 1.6 wird deutlich, dass eine Entscheidung getroffen werden musste, *wie* das Uhrenobjekt in den Kernel integriert werden soll. Die Uhr könnte fest in den Kernel gelinkt werden oder als Modul kompiliert werden.

Diese Entscheidung ist uns relativ einfach gefallen. Die Vorteile eines Kernel-Moduls überwiegen für eine solche Aufgabe mit Abstand. Als erstes muss dazu festgehalten werden, dass es für die Ausführungsgeschwindigkeit keine Rolle spielt, ob der Code als Modul vorliegt oder nicht. Wichtig ist allein, dass er im Kernelmode ausgeführt wird. Für die Entwicklung hingegen ist die Variante des Moduls massiv besser. Dies aus zwei Gründen: zum einen dauert es nur einen Sekundenbruchteil, um ein Kernel-Modul zu kompilieren, während das Kompilieren des gesamten Kernels ca. 3-20 Minuten in Anspruch nimmt. Dies kann durchaus ein schwerwiegendes Kriterium sein, wenn man bedenkt, dass ziemlich häufig kompiliert werden muss. Zum anderen kann ein Kernel-Modul sehr einfach mit dem Shell Command[1] `insmod`

[1] Als *shell* wird das Eingabefenster von Linux bezeichnet

in den Kernel eingefügt und mit `rmmod` wieder entfernt werden. Um hingegen einen neukompilierten Kernel zu testen, muss, von ein paar vorbereitenden abgesehen, das ganze System neu gebootet werden, was wiederum etwa eine Minute kostet. Die enorme Zeitersparnis im Fall des Kernel-Moduls ist also offensichtlich.

Es gibt noch einen Punkt, der gegen das statische Linken in den Kernel spricht: Unser Uhrenobjekt hat ein verhältnismässig kleines Zielpublikum und es macht somit keinen Sinn, dass es in jeden Kernel standardmässig integriert wird. Man stelle sich nur vor, wie ein Standard-Kernel aussehen würde, wenn jede Sonderfunktionalität darin aufgenommen würde. Er wäre unnötig aufgeblasen mit Dingen, die der durchschnittliche Benutzer nie braucht. Deshalb ist es eine sehr elegante Lösung, die Uhr als Modul zu implementieren, das dann gezielt von denjenigen Personen eingesetzt werden kann, die Verwendung dafür finden.

Zusammenfassend die Vorteile des Kernel-Moduls:

- Hat auf Ausführungsgeschwindigkeit keinen Einfluss
- Enorme Flexibilität
- Schnelleres Kompilieren
- Schnelleres Testen
- Wird der verhältnismässig kleinen Zielgruppe gerecht

2.3 Die Software-Uhr

Die Idee der Software-Uhr besteht zu einem Teil darin, dass jegliche Hardware-Ressourcen unangetastet bleiben und das Betriebssystem somit nicht beeinflusst wird. Das TSC-Register wird in jedem Fall nur gelesen. Um die Anforderungen in Abschnitt 2.1 zu erfüllen, kann das Uhrenobjekt auf vorerst abstrakter Ebene etwa so konzipiert werden wie in Abbildung 2.1 dargestellt. Daraus ist ersichtlich, dass es intern über eine ganze Reihe von Uhren verfügt, welche sich in *Referenzuhren* und *Applikationsuhren* unterteilen. Erstere verfügen sowohl über lesende als auch über schreibende Interfaces, während den Applikationen keine schreibenden Funktionen zur Verfügung stehen. Die saubere Trennung rührt daher, dass Applikationen, die auf die Uhren zugreifen, gewisse Anforderungen an diese haben. So hat eine Anwendung die Möglichkeit, die von ihr initialisierte Applikationsuhr so zu konfigurieren, dass diese niemals eine bestimmte minimale und maximale Laufgeschwindigkeit unter- resp. überschreitet. Eine der häufigsten, praktischen Forderungen ist wohl die, dass eine Uhr niemals rückwärts laufen darf. Also werden vom Synchronisationsalgorithmus gewünschte sprunghafte Änderungen für die Applikation in ein Zeitintervall transformiert, in welchem die Uhr schneller oder langsamer läuft. Ein qualitatives Beispiel zeigt Abbildung 2.2.

Rot dargestellt ist ein möglicher Verlauf einer Referenzuhr. Sie springt an einem Punkt plötzlich rückwarts und fährt dann mit gleicher Steigung fort. Unter Umständen kann die Applikationsuhr ihrer Anwendung diesen Sprung nicht so weitergeben.

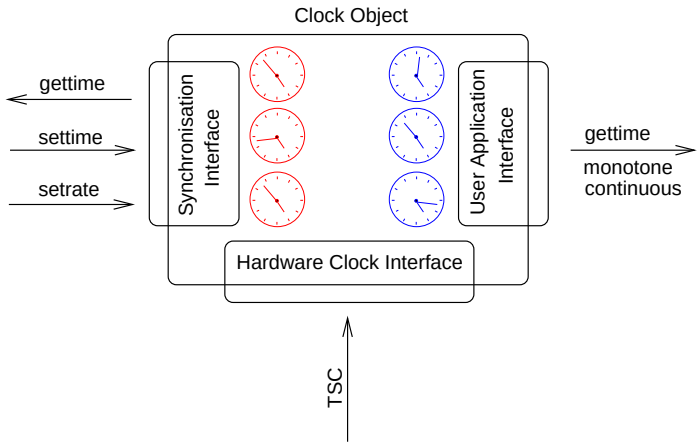

Abbildung 2.1: Uhrenobjekt

Deshalb sinkt diese mit der minimal ihr erlaubten Steigung, bis die Zeitwerte der beiden Uhren wieder gleich sind. Danach stimmt die Applikationsuhr und die Referenzuhr wieder überein.

Es stellt sich nun immer mehr die Frage, wie die Zeit überhaupt bestimmt und wiedergegeben wird. An dieser Stelle kommt nun der in Abschnitt 1.3 erläuterte TSC zum Zuge. Die Zeit wird nämlich als Funktion des momentanen TSC-Registerwertes wiedergegeben:

$$C = \alpha + (1 + \beta) \cdot TSC \quad [ticks] \tag{2.1}$$

C steht dabei für Clock. Abbildung 2.3 veranschaulicht diese Formel auf leicht einsichtige Weise. Der Parameter α sorgt für den nötigen Offset, während β die Laufgeschwindigkeit reguliert. Aus der Formel ist ersichtlich, dass die Laufgeschwindigkeit der Uhr genau der des TSC-Registers entspricht, wenn β Null ist (blaue und grüne Gerade).

Es gilt also festzuhalten, dass die Zeit nur durch ein gespeichertes Variablenpaar, nämlich α und β, repräsentiert wird. Möchte nun eine Anwendung oder ein Kernel-Modul die Zeit lesen, so wird diese gemäss Formel 2.1 berechnet und zurückgegeben. Es ist also nicht nötig, dass ein Prozess im Hintergrund ununterbrochen etwas ausführen müsste, wie das etwa der Fall ist beim Nachführen der Jiffies für die Systemzeit.

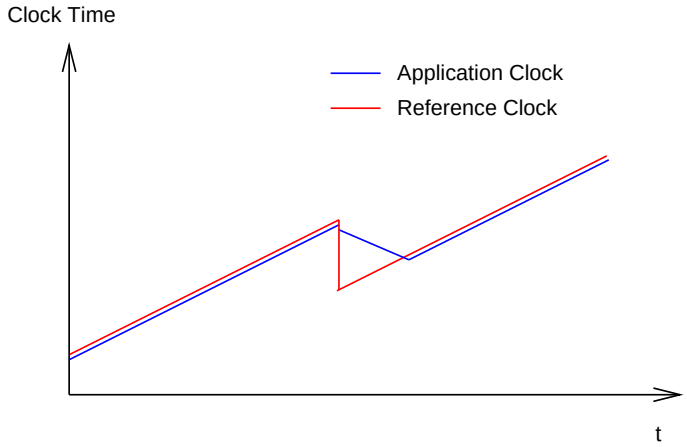

Abbildung 2.2: Qualitativer Verlauf einer Referenz- und einer Applikationsuhr

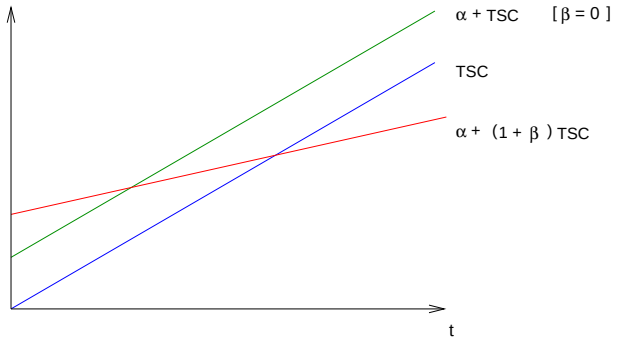

Abbildung 2.3: Illustration der Zeitformel

2.4 Formate und Kommunikationsprotokoll

Erinnern wir uns, dass das zu implementierende Uhrenobjekt als Infrastruktur für eine Client/Server-Struktur über WLAN dient. Für die Kommunikation zwischen den verschiedenen Knoten muss also ein Protokoll vereinbart werden, mittels dessen die Daten ausgetauscht werden können. Das beinhaltet, dass für Variablen wie α, β, TSC und einige weitere ein einheitliches Format gewählt werden muss. In Tabelle 2.1 sind die gewählten C-Formate mit ihren jeweiligen Längen aufgelistet. Die Variable TSC muss die vollen 64-bit des TSC-Registers übernehmen, da sie mit 32-bit und einer CPU-Taktrate von beispielsweise 870 MHz bereits nach 5 Sekunden überlaufen würde. Aus Formel 2.1 wird sofort klar, dass die Variable α dieselbe Länge haben muss wie TSC. Allein β kommt mit 32-bit aus und ist zudem vorzeichenbehaftet.

Variable	C-Format	Länge
α	unsigned long long	64 bit = 8 bytes
β	signed integer	32 bit = 4 bytes
TSC	unsigned long long	64 bit = 8 bytes

Tabelle 2.1: Formate der Software-Uhr

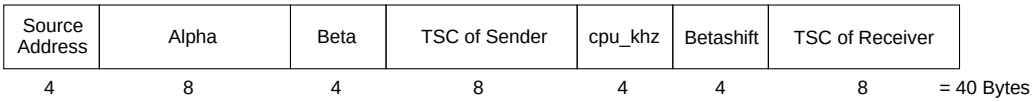

Abbildung 2.4: Kommunikationsprotokoll

Das Protokoll, zu dem wir uns entschieden haben, ist in Abbildung 2.4 dargestellt. Es beinhaltet zu einem Teil die obigen Variablen mit der entsprechenden Länge. Die restlichen Felder haben folgende Bedeutung:

- Die *Source Address* ist die IP-Adresse des jeweiligen Senders.
- *cpu_khz* meint die CPU-Taktrate des Senderrechners in kHz.
- *Betashift* sagt aus, um wieviele Bits β nach links geshiftet wurde. Dieses Format wurde gewählt, weil β eine sehr kleine Zahl ist und *float Variablen* innerhalb des Kernels nicht ohne weiteres gebraucht werden können.

2.5 Logische Struktur

Abbildung 2.5 zeigt, wie das Uhrenobjekt auf logischer Ebene in die Kernel-Umgebung eingebettet ist. Es ist erkennbar, dass der Synchronisationsalgorithmus, sehr ver-

einfacht dargestellt, ebenfalls im Kernelspace implementiert ist und die vom Uhrenobjekt zur Verfügung gestellten Interfaces benutzt.

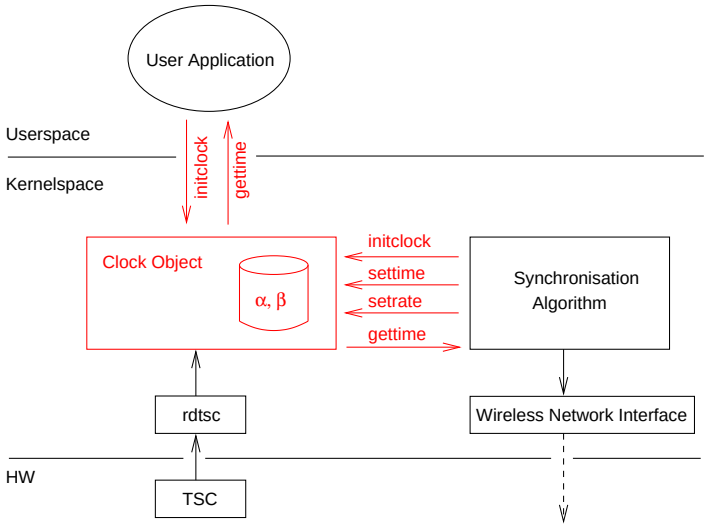

Abbildung 2.5: Struktur der Realisierung

Ausgehend von dieser logischen Struktur gehen wir nun etwas weiter ins Detail. Abbildung 2.6 soll illustrieren, wie das Uhrenobjekt seine internen Funktionen einerseits gegenüber dem Kernelspace, andererseits gegenüber dem Userspace, zur Verfügung stellt. Im Falle der Kernelspace-Interfaces, rot dargestellt, werden die vorhandenen Uhr-internen Funktionen einfach als Kernel-Symbole exportiert. Damit werden sie für den gesamten Kernel bekannt und verfügbar. Das Modul, das den Synchronisationsalgorithmus implementiert, muss sich also nicht darum kümmern, woher diese Funktionen zur Verfügung gestellt werden, sondern kann sie einfach benutzen.

Nicht ganz so einfach sieht es bei den Interfaces für den Userspace aus (blau). Im Prinzip geht es auch hier nur darum, dass Uhr-interne Funktionen von Applikationen aufgerufen werden können. Das könnte grundsätzlich mit verschiedenen Methoden gelöst werden. Das */proc filesystem*[2] wäre eine davon gewesen. Wie sich jedoch schnell zeigte, wäre diese Variante zu langsam, unflexibel und nicht genügend beeinflussbar gewesen.

Ganz anders mit System Calls, welche einige Vorteile haben:

- Genügend beeinflussbar bezüglich Geschwindigkeit
- Einfache Lösung für die Anwendungen

[2]Das */proc filesystem* ist ein virtuelles Filesystem, das nicht Speicher verwaltet, sondern dem Benutzer den Zugang zu Inhalten von Datenstrukturen im Kernel erlaubt.

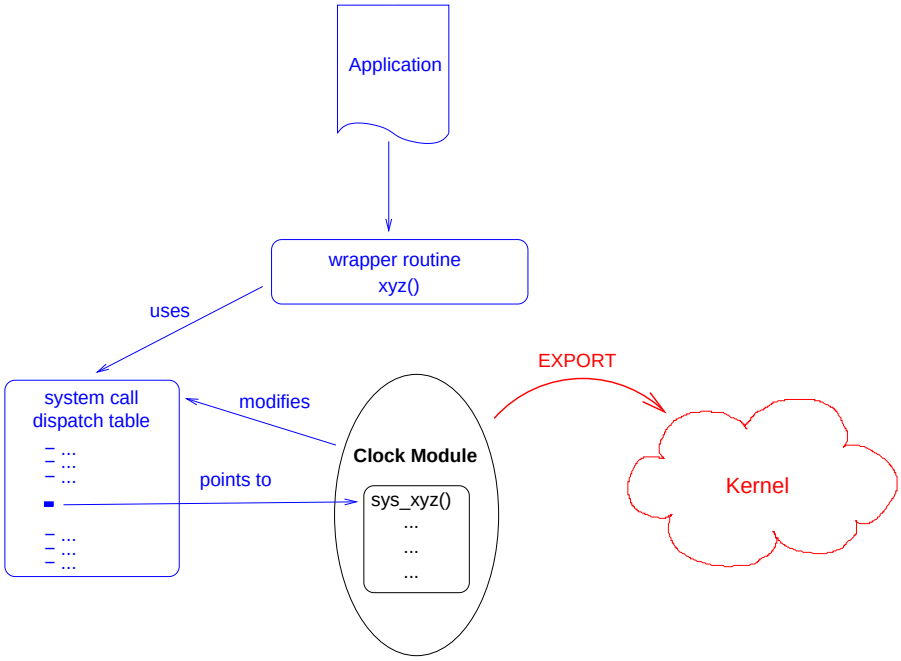

Abbildung 2.6: Interfaces für Kernel- und Userspace

- Flexibel, insbesondere für Portierung

Wir haben uns also für diese Variante entschieden. In Abschnitt 1.2 wurde gezeigt, wie diese funktionieren. Die in Abbildung 1.2 als *xyz()* dargestellte *wrapper routine* ist hier ebenfalls zu finden. Diese ruft bekanntlich eine *system call service routine* auf. Das Spezielle ist, dass es sich in unserem Fall nicht um eine "konventionelle" handelt, sondern um eine vom Uhrenobjekt implementierte Funktion. Damit dies möglich ist, muss die *system call dispatch table* [3] so modifiziert werden, dass ein ausgewählter Eintrag entsprechend auf unsere Methode umgeleitet wird. In Abbildung 2.6 ist dieser Sachverhalt mit dem Pfeil dargestellt, der die Beschriftung *modifies* trägt. So gelingt es schliesslich, ein sauberes und flexibles Interface des Uhrenobjekts für den Userspace zu implementieren.

2.6 C-Struktur

Nun soll noch der letzte Abstraktionsgrad beiseite gelassen werden. Wie sieht also die konkrete Implementierung in C aus? In Abbildung 2.7 wird gezeigt, wie die einzelnen C-Dateien kompiliert, gelinkt und benutzt werden.

[3] Diese Tabelle ist gespeichert im `sys_call_table` array. Sie hat `NR_syscalls` Einträge (meist 256) und ist in der Datei linux/arch/i386/kernel/entry.S definiert.

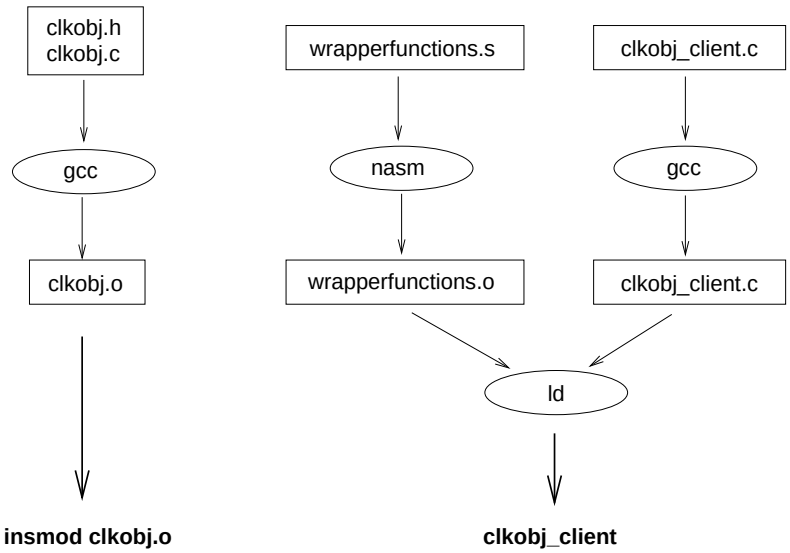

Abbildung 2.7: C-Dateien

Das Uhrenobjekt selbst besteht aus einer *.h*- und einer *.c*-Datei mit den Namen *clkobj.h* und *clkobj.c*. Sie werden zum Modul *clkobj.o* kompiliert, welches mittels `insmod`-Befehl[4] in den laufenden Kernel eingefügt werden kann. Dieses Einfügen muss natürlich als erstes geschehen, da es sich dabei um das Kernstück handelt, von dem alle Funktionalitäten abhängen.

Um die ausführbare Userspace-Applikation *clkobj_client* zu generieren, werden die beiden *.o*-Dateien miteinander gelinkt[5], welche aus den Files *wrapperfunctions.s* und *clkobj_client.c* durch Kompilieren entstehen. Wie an der Endung erkennbar ist, handelt es sich bei der Datei mit den *wrapper routines* um Assembler-Code, der mit *nasm* kompiliert wird. *clkobj_client* ist so, wie sie im Anhang aufgeführt ist (A.2.3), eine Beispielapplikation, welche zwei Referenz- und eine Applikationsuhr initialisiert und dann einen bestimmten Verlauf der Referenzuhren vorgibt.

2.7 Implementation des Uhrenobjekts

In diesem Abschnitt sollen erwähnenswerte Teile vom Quellcode des Uhrenobjekts diskutiert werden. Im Anhang sind alle beteiligten Dateien aufgeführt (A), insbesondere auch das Makefile (A.1), auf das hier nicht eingegangen wird.

[4] von “insert module”
[5] Der Linker von GNU heisst *ld*

2.7.1 clkobj.h

Wir beginnen mit dem Headerfile *clkobj.h* (A.2.1). Das Vorgehen wird bei allen Unterabschnitten dasselbe sein: Die Datei wird von vorne her durchgegangen, wobei jeweils nur das wichtige zur Sprache kommt. Das entsprechende Code-Segment ist dann jeweils nochmals angegeben, um die Lesbarkeit zu verbessern.

Gleich am Anfang des Files finden sich zwei Strukturen:

```
struct clkval
{
    unsigned long long alpha;
    int beta;
    unsigned long long tsc;
    unsigned int cpu_khz;
};

struct clk_desc
{
    int desc;
    unsigned int cpu_khz;
    int max_clk_drift_ppm;
    unsigned long ip;
};
```

Die erste, *struct clkval*, dient dazu, die Uhren-internen Daten an die Applikationen zurückzugeben. Dies kommt daher, dass System Calls grundsätzlich nur einen Erfolgsstatus zurückgeben (siehe 1.2). Die Struktur wird also vom Uhrenobjekt mit den entsprechenden Daten gefüllt, wo sie dann von der Applikation ausgelesen werden kann. Neben den bekannten Werten α, β und TSC kommt noch die Member-Variable `cpu_khz` dazu, welches die CPU-Taktrate des lokalen Rechners in kHz angibt. Diese Grösse wird benötigt, um eine Zeit, die in der Einheit `[ticks]` vorliegt, umwandeln zu können in `[ns]`.

Die zweite Struktur, *struct clk_desc*, dient dem Client[6], der die Referenzuhr[7] initialisiert, als Beschreibung. Die Member-Variablen bedeuten (ohne `cpu_khz`, da bereits diskutiert):

- `desc` ist der Deskriptor, mit welchem die Uhr bei allen weiteren Interaktionen identifiziert wird.
- `max_clk_drift_ppm` gibt den Drift in *ppm* an, den die Uhr maximal hat.
- `ip` ist die IP-Adresse des Rechners, auf dessen Uhr sich die lokale Referenzuhr synchronisiert (?)

Das nächste erwähnenswerte, das uns begegnet in dieser Datei, ist die eile

[6] also dem Synchronisationsalgorithmus

[7] Im Code ist mit dem Kürzel *clk* immer eine Referenzuhr gemeint. Applikationsuhren werden mit *appclk* bezeichnet

```
#define EXPORT_SYMTAB
```

Diese Definition muss sowohl in der *.h*- als auch der *.c*-Datei gemacht werden, damit das Exportieren der Uhren-internen Funktionen in den Kernel funktioniert (illustriert in Abbildung 2.6).

Gleich darauf folgen weitere Definitionen:

```
#define __NR_initclk 230
#define __NR_setclk 231
#define __NR_settime 232
#define __NR_setrate 233
      ...
```

Diese Zahlen geben an, welcher Eintrag der *system call dispatch table* auf die entsprechende Funktion umgeleitet wird. Die Definition `#define __NR_initclk 230` bedeutet also, dass der Eintrag mit dem Index 230 auf die Funktion `sys_initclk()` umgeleitet wird. Zur Erinnerung: dies geschieht beim Einfügen des Uhrenobjekts in den Kernel (`insmod`). Beim Entfernen (`rmmod`) wird der Eingriff in diese wichtige Tabelle natürlich rückgängig gemacht.

Es ist wichtig, sich bewusst zu sein, dass diese Manipulation sehr gefährlich sein kann, da der System Call, der ersetzt wird, nicht mehr zur Verfügung steht. Deshalb ist es unumgänglich, dass auf jedem System, auf dem das Uhrenobjekt benutzt werden soll, nur Einträge ersetzt werden, die auf die "Dummy-Funktion" `sys_ni_syscall` zeigen. Diese müssen manuell gesucht werden in der Kernel-Datei `linux/arch/i386/kernel/entry.S`.

Die nächste Definition,

```
#define MAXNUMBEROFCLOCKS 10
```

gibt an, wieviele Uhren das Uhrenobjekt instanzieren[8] kann. Entsprechend dieser Zahl wird beim Einfügen des Moduls Speicher reserviert.

Damit sind wir bei der letzten Definition angelangt:

```
#define BETASHIFT 24
```

Im Abschnitt 2.4 wurde in Zusammenhang mit dem Protokoll erklärt, dass β eine sehr kleine Zahl ist und somit um einige Stellen nach links geshiftet übertragen wird. Die vorliegende Konstante `BETASHIFT` sagt nun, um wieviele bits geshiftet wird. In diesem Falle von einem shift um 24 Stellen handelt es sich somit um eine Multiplikation mit der Zahl 2^{24}.

Wie der Name der Variable vermuten lässt, handelt es sich bei

```
extern long sys_call_table[];
```

um das oben erwähnte Array, das die *system call dispatch table* enthält. Da es sich dabei um ein Kernel-weit bekanntes Symbol handelt, reicht es, eine Deklaration mit dem Schlüsselwort `extern` vorzunehmen. Danach steht sie im Modul zur Verfügung.

[8] Das Wort *Instanz* ist hier, wie im ganzen Dokument, nicht im Sinne der Programmiersprache C/C++ gemeint, sondern steht einfach für die *Initialisierung* einer Uhr

```
static int (*old_call_initclk)(int);
static int (*old_call_setclk)(int);
static int (*old_call_settime)(int);
      ...
```

Diese Pointer werden gebraucht, um die bestehenden Einträge in der *system call dispatch table* zwischenzuspeichern. Damit wird sichergestellt, dass die Umleitung auf die Methoden des Uhrenobjekts rückgängig gemacht werden kann.

```
struct clk
{
    unsigned long long alpha;
    int beta;
    unsigned long ip;
};

struct appclk
{
    unsigned long long alpha;
    int beta;
    int max_slope;
    int number_of_parents;
    int *parents;
};
```

Obige zwei Strukturen dienen dazu, die eigentlichen Werte der Uhren abzulegen. Erstere enthält nur bekannte Variablen, während die Struktur für die Applikationsuhren, `struct appclk`, noch folgendes erwähnenswertes enthält:

- `max_slope` ist derjenige Drift, den eine Applikation maximal zulässt. Das Format ist dasselbe wie dasjenige von β.
- `number_of_parents` ist die Anzahl Referenzuhren, auf die sich die Applikationsuhr bezieht oder, anders ausgedrückt, "als Eltern hat".
- `parents` ist das Array, das die Deskriptoren der `number_of_parents` "Eltern"-Referenzuhren enthält.

Die Funktionsdeklarationen, die nun im Headerfile folgen, werden an dieser Stelle nicht besprochen. Eine Diskussion der Methoden findet sich im Unterabschnitt 2.7.2. Was nun folgt, wurde oben bereits thematisch angeschnitten:

```
EXPORT_SYMBOL_NOVERS(sys_initclk);
EXPORT_SYMBOL_NOVERS(sys_setclk);
EXPORT_SYMBOL_NOVERS(sys_settime);
      ...
```

Es handelt sich dabei um das Exportieren der Funktionen als Kernel-Symbole. Dies wird mit dem Kernel-Makro `EXPORT_SYMBOL_NOVERS` gemacht, welches eine strikte Kontrolle der Kernelversionen unterlässt. Das sonst verbreitete Makro `EXPORT_SYMBOL` hat bei uns nicht funktioniert. Zur Illustration dieses Vorgangs kann die Abbildung 2.6 eine Hilfe sein.

```
struct clk *clk_array;
struct appclk *appclk_array;
```

In diesen Arrays werden die Strukturen, welche oben erläutert wurden, abgelegt. Der dafür nötige Speicherplatz wird beim Einfügen des Moduls gemäss `MAXNUMBEROFCLOCKS` statisch alloziert. Es stehen gleich viele Referenz- und Applikationsuhren zur Verfügung. Der Array-Index einer bestimmten Uhr entspricht *Deskriptor der Uhr - 1*. Anschaulich heisst das, dass die erste Instanz einer Uhr den *Deskriptor 1* hat, währenddem der *Index 0* ist. Ersteres entspricht eher der menschlichen Intuition, während letzteres von der Programmiersprache C herrührt. Die beiden Variablen

```
int count_clk;
int count_appclk;
```

zählen, wie die Nomenklatur deutlich macht, die Referenz- resp. Applikationsuhren.

Der Rest des Headerfiles sind intern benötigte Variablen, wovon nur noch zu erwähnen bleibt, dass die CPU-Taktrate im Modul nur einmal bestimmt wird und dann in der Variable

```
unsigned long cpu_khz;
```

abgelegt wird.

2.7.2 clkobj.c

Im folgenden wird zu jeder Methode des Uhrenobjekts das Essentiellste in aufzählender Form erklärt. Die Implementation derselben befindet sich in der Datei *clkobj.c* (A.2.2). Um die Deklarationen etwas lesbarer zu machen, werden jeweils alle vorangestellten Schlüsselwörter wie z.B. `asmlinkage` weggelassen.

```
int sys_initclk(unsigned long ip, struct clk_desc *rd);
```

- Initialisiert eine Referenzuhr.
- Es wird überprüft, ob `MAXNUMBEROFCLOCKS` noch nicht überschritten wird.
- Die Zeit der Uhr wird mittels `do_gettimeofday()` auf einen plausiblen Initialwert gesetzt.
- $\alpha \neq 0$, $\beta = 0$
- Den Membervariablen der Deskriptorstruktur `struct clk_desc` werden die richtigen Werte zugewiesen.

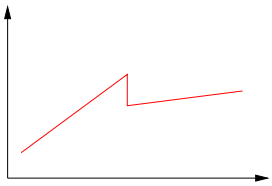

Abbildung 2.8: sys_setclk()

```
int sys_setclk(int clk_desc, unsigned long long *new_time, int *new_beta);
```

- Überprüft, ob der Deskriptor gültig ist. Dies machen alle Funktionen mit einem übergebenen Deskriptor und wird im folgenden nicht mehr erwähnt werden.
- Setzt die Zeit der Uhr auf `new_time` und die Driftgeschwindigkeit auf `new_beta`. Erstere wird in `[ticks]` angegeben.
- α und β werden modifiziert.
- $\alpha = new\_time - TSC - TSC * new\_beta$
- Die darin vorkommende Multiplikation "$new\_beta * TSC$" ist nicht ganz einfach zu berechnen, wie bereits weiter oben festgehalten wurde. Das Problem wird dadurch gelöst, dass der 64-bit breite TSC-Registerwert aufgeteilt wird in zwei 32-bit breite, welche jeweils in den 4 high-bytes einer 64-bit Variable platziert werden. Siehe dazu Abbildung 2.9. So steht im rechten Teil beider Hilfsvariablen genug freier Platz zur Verfügung, um zuerst den `BETASHIFT` nach rechts und dann die Multiplikation mit `new_beta` durchzuführen.

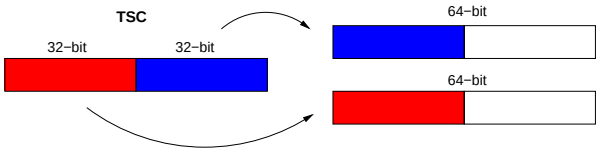

Abbildung 2.9: Aufteilung in zwei 32-bit Werte

```
int sys_settime(int clk_desc, unsigned long long *new_time);
```

- Setzt die Zeit der Uhr auf `new_time`, ohne dass die Driftgeschwindigkeit geändert wird.
- Nur α wird modifiziert.

- $\alpha = new\_time - TSC - TSC * new\_beta$

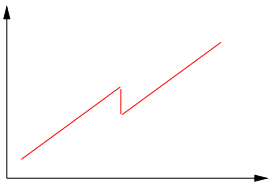

Abbildung 2.10: sys_settime()

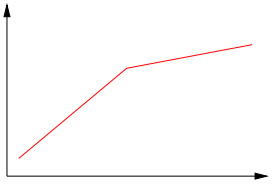

Abbildung 2.11: sys_setrate()

```
int sys_setrate(int clk_desc, int *new_beta);
```

- Ändert die Driftgeschwindigkeit, indem β auf `new_time` gesetzt wird.
- α muss jedoch auch neu gesetzt werden, damit die Zeit der Uhr *nicht springt*.
- $new\_alpha = old\_alpha + TSC * (old\_beta - new\_beta)$

```
int sys_getrawtime(int clk_desc, struct clkval *cv);
```

- Gibt die Zeit der Uhr in roher Form, also als α, β und TSC, zurück. Sie werden in der Resultatstruktur `cv` abgelegt.

```
int sys_readtime(int clk_desc, unsigned long long *now);
```

- Gibt die Zeit der Uhr in Form von Ticks zurück, berechnet diese also selbst aus α, β und TSC.
- Formel für Zeit: $now = \alpha + TSC + TSC * \beta$
- Das Resultat wird in der Variable `now` abgelegt.

```
int sys_init_appclk(int *appclk_desc, int max_slope, int number_of_parents,
```

- Diese Funktion dient der Initialisierung einer Applikationsuhr.
- Es wird überprüft, ob `MAXNUMBEROFCLOCKS` noch nicht überschritten wird.
- Stellt sicher, dass `max_slope` > 0 ist. Nötigenfalls wird das Vorzeichen gewechselt.
- Alloziert Speicher für das "Eltern-Array" gemäss `number_of_parents`.
- Die Zeit der Uhr wird mittels `do_gettimeofday()` auf einen plausiblen Initialwert gesetzt.
- Weise den Deskriptor der Variable `appclk_desc` zu.

```
int sys_gettime_appclk(int appclk_desc, unsigned long long *nsec);
```

- Gibt die Zeit der Applikatioinsuhr in der Einheit `[ns]` zurück, berechnet diese also selbst aus α, β und TSC. Die Umwandlung von `ticks` in `ns` mittels der Member-Variable `cpu_khz` schliesst die Berechnung ab.
- Der zurückgegebene Zeitwert ist die geglättete Zeit der genauesten Elternuhr.

- Der Algorithmus beginnt damit, dass die genaueste Elternuhr gefunden werden muss. Dazu muss die Zeit von jeder solchen berechnet werden. Gemäss dem LS-Algorithmus wird diejenige Uhr mit dem grössten Zeitwert ausgewählt.
- Beim ersten Aufruf dieser Funktion wird die Zeit der Referenzuhr ungeprüft zurückgegeben.
- Die Applikationsuhr übernimmt die Werte α und β der ausgewählten Elternuhr.
- Wurde die Funktion schon mehrmals aufgerufen, wird nun geprüft, ob der exakte Zeitwert innerhalb der geforderten Grenzen liegt. Diese Grenzen werden berechnet aus der vom Benutzer spezifizierten `max_slope`.
- Würde der maximal und minimal erlaubte Drift verletzt, wird derjenige Extremwert zurückgegeben, der gerade noch den Anforderungen genügt.
- Der Wert, der beim letzten Mal zurückgegeben wurde, muss also gespeichert werden.
- Achtung: Diese Methode funktioniert im jetzigen Status nur mit einer einzigen Applikationsuhr, da für die früher zurückgegebenen Zeitwerte kein Array existiert, sondern nur eine einzige Variable!

- Ursprünglich geplant wäre gewesen, das Resultat als `struct timeval` zurückgeben, was kompatibel wäre mit dem System Call `gettimeofday()`.

```
int sys_nsec2ticks(unsigned long long *nsec, unsigned long long *ticks,
                   unsigned int cpu_khz);
```

- Wandelt den 64-bit breiten Wert `nsec` in `ticks` um.
- Benutzt die Funktion `div_mult()`.

```
int sys_ticks2nsec(unsigned long long *ticks, unsigned long long *nsec,
                   unsigned int cpu_khz);
```

- Wandelt den 64-bit breiten Wert `ticks` in `nsec` um.
- Benutzt die Funktion `div_mult()`.

```
int sys_calctime(unsigned long long *alpha, unsigned long long *tsc,
                 int *beta, unsigned long long *now);
```

- Dies ist eine Hilfsfunktion, die eine Zeit in Form von `ticks` berechnet. Die dafür erforderlichen Werte α, β und TSC stammen nicht aus dem Uhrenobjekt selbst und dem momentanen TSC-Registerwert, sondern werden als Parameter übergeben.

```
int sys_getclkpntr(int clk_desc, struct clk **c);
```

- Gibt einen Pointer zu der Referenzuhr mit dem Deskriptor `clk_desc` zurück.
- Ist implementiert, jedoch bis zum jetzigen Zeitpunkt kaum gebraucht worden.

```
int div_mult(unsigned long long *dividend, unsigned int divisor,
             unsigned int mult, unsigned long long *result);
```

- Löst das schon mehrfach angetönte Problem, dass im Kernel die Floating Point Unit (FPU) nicht einfach benutzt werden kann.
- Der Algorithmus basiert auf der Datei `linux/include/asm_i386/div64.h`, wurde jedoch erweitert, damit eine Division *und* eine Multiplikation ausgeführt wird.
- Für das Verständnis dieser Erläuterungen sollte der Quellcode betrachtet werden.
- Die 64-Bitbreite wird aufgeteilt in zweimal 32-bit.

- Da *nach* der Division noch multipliziert wird und sich somit kleine Fehler einer Rundung vervielfachen würden, muss der Rest der Division zwischengespeichert werden (__mod). Dieser wird separat zuerst multipliziert (Resultat: `temp_L1`) und dann dividiert (Resultat: `temp_L2`).
- Somit kann `temp_L2` zum Schluss einfach mitaddiert werden.

```
unsigned long calibrate_tsc(void);
```

- Aus dem Linux-Kernel[9] fast unverändert übernommen.
- Die Membervariable `cpu_khz`, also die CPU-Taktfrequenz, wird bestimmt, indem genau gleich wie beim Systemstart, die Anzahl Clock-Signale während einer Zeit von 50.00077 Millisekunden gezählt werden.

2.7.3 clkobj_client.c

Diese Datei (A.2.3) macht Gebrauch von den verschiedenen System Calls, die das Uhrenobjekt zur Verfügung stellt. Es handelt sich bloss um ein willkürliches Beispiel, wie eine Applikation die Uhr benützen kann. Deshalb seien an dieser Stelle nur zwei Dinge herausgestrichen:

```
#include <clkobj.h>
```

Das Headerfile `clkobj.h` muss eingebunden werden, da sonst die nötigen Deklarationen der Strukturen fehlen.

```
extern int initclk(unsigned long ip, struct clk_desc *rd);
```

Die System Calls, die benutzt werden sollen, müssen mit dem Schlüsselwort `extern` deklariert werden. Die Funktion trägt hier im Userspace nicht mehr die Vorsilbe "`sys_`", sondern heisst so, wie sie in der Datei `wrapperfunctions.s` benannt wurde. Wir haben uns an die Konvention gehalten, dass es sich dabei um denselben Namen handelt wie im Kernelspace, jedoch mit weggelassener Vorsilbe "`sys_`".

2.7.4 wrapperfunctions.s

Bei dieser Datei handelt es sich um Assembler-Code, der die Aufgabe hat, einen System Call für den Userspace zur Verfügung zu stellen, indem er dafür sorgt, dass die Argumente an den Kernelspace weitergegeben werden und der `$0x80` *system call trap* ausgeführt wird. Damit der *richtige* System Call aufgerufen wird, ist es unumgänglich, dass die System Call Nummer im `eax`-Register mit der vom Uhrenobjekt dafür bestimmten übereinstimmt.

Für weitere Details sei auf die Quellcode-Dokumentation verwiesen.

[9] aus linux/arch/i386/kernel/time.c

2.8 Anwendnung und Praktisches

In diesem Abschnitt wird in knapper Form gezeigt, wie die vorhandenen Objekte und Applikationen praktisch benutzt werden.

Sind alle Quellcodekomponenten fehlerfrei vorhanden, müssen diese als erstes kompiliert werden. Dies geschieht, vorausgesetzt wir befinden uns im entsprechenden Verzeichnis, mittels des Shell-Kommandos

```
make.
```

Auf das dabei ausgeführte `Makefile` wurde bereits verwiesen (A.1).

Nun muss das Uhrenobjekt als Modul in den Kernel eingefügt werden. Dazu wird als Superuser der Command

```
insmod clkobj.o
```

ausgeführt, wonach das Modul in der Liste des `lsmod` Commands als `clkobj` erscheinen muss.

Nun steht die Infrastruktur des Uhrenobjekts zur Verfügung und der Userspace-Client kann ausgeführt werden. Um das eingefügte Kernel-Modul wieder zu entfernen, bedient man sich (wiederum als Superuser) des Shell Commands

```
rmmod.
```

Allfällige Kernel-Debug-Meldungen, die mittels `printk()` im Quellcode gemacht werden, sieht man sich am besten mit dem Kommando

```
dmesg
```

oder meist besser

```
dmesg | tail -x
```

an. Die zweite Möglichkeit gibt nur die letzten x Zeilen aus. Es muss davon abgeraten werden, sich auf die in der `xconsole` erscheinenden Meldungen zu verlassen, weil diese nicht immer zuverlässig nachgeführt werden. Vor allem, wenn man eine Ausgabe vermisst, sollte man immer auf den `dmesg` Command zurückgreifen.

Kapitel 3

Validierung

3.1 Prinzip

Das Prinzip der Validierung ist in Abbildung 3.1 dargestellt. Anstelle des Synchronisationsalgorithmus wird das Uhrenobjekt von einer Userspace-Applikation modifiziert. In unserem Fall heisst das, dass diese Applikation *zwei Referenzuhren* initialisiert und danach verschiedene, willkürliche Modifikationen mittels System Calls daran vornimmt. Eine weitere Anwendung initialisiert *eine Applikationsuhr*, die auf die beiden vorhandenen Referenzuhren zugreift. Gemäss LS-Algorithmus richtet sich die Applikationsuhr immer nach derjenigen Referenzuhr mit dem grösseren Zeitwert. Wichtig ist jedoch, dass der Applikationsuhr je eine obere und untere Driftgrenze vorgegeben wird, die sie der Anwendung zurückgeben darf. Bei der Validierung wird darauf geachtet, dass den Referenzuhren z.T. Werte ausserhalb dieser Grenzen gesetzt werden, damit dieser Mechanismus sichtbar wird.

Um das ganze auswerten und visualieren zu können, werden die von der Applikationsuhr zurückgegebenen Werte jeweils in eine Datei gespeichert, von wo sie anschliessend (*offline*) von Matlab gelesen und verarbeitet werden können.

3.2 Resultate

In Abbildung 3.2 ist ein Beispiel eines solchen Verlaufs (siehe 3.1) dargestellt. Es zeigt deutlich, dass die Applikationsuhr (grün) sowohl im ersten als auch im zweiten Teil derjenigen Uhr mit der späteren Zeit "nacheilt". Zuerst ist es die Referenzuhr 2 (rot), dann die andere (lachsrot).

Ebenfalls deutlich wird die Tatsache, dass die Applikationsuhr eine Driftgeschwindigkeit, die ihre Vorgaben übersteigt, nicht mehr mitmacht und mit ihrer maximal erlaubten Geschwindigkeit weiterläuft. Sobald die Referenzuhr jedoch langsamer wird und sich im erlaubten Bereich der Applikationsuhr bewegt, stimmen deren Zeiten wieder exakt überein.

Es ist noch zu sagen, dass die Werte beider Achsen so skaliert wurden, dass sie bei Null beginnen. Das ist ohne weiteres erlaubt, weil es ja keine Rolle spielt, *wann*

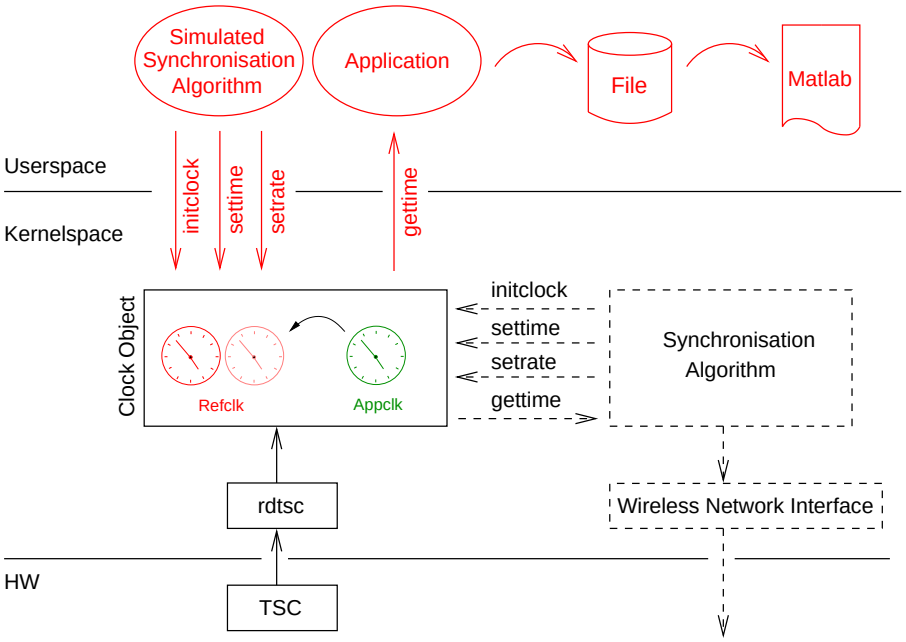

Abbildung 3.1: Prinzip der Validierung

in absoluter Zeit dieser Testlauf gestartet wird. Auf der x-Achse ist die absolute Zeit in der Einheit von `[ticks]`, während die y-Achse die Uhrzeiten in derselben Einheit zeigt.

3.3 Validierung der Uhrensynchronisation über WLAN

Die Validierung des Gesamtsystems gehörte nicht zu meinen Aufgaben. Dass sie trotzdem hier thematisiert wird, hat den Grund, dass ich dafür eine kleine elektronische Schaltung entwickelt habe, die ein wichtiger Bestandteil davon war. In den folgenden Unterabschnitten wird deshalb kurz die Idee für die Validierung und die elektronische Schaltung beschrieben.

3.3.1 Prinzip und Realisierung

Um die Uhrensynchronisation über wireless LAN, also das Gesamtsystem, zu validieren, ist es nötig, dass man die Zeit je eines Servers und eines Clients zur exakt gleichen Zeit liest und sie vergleicht. Tut man dies mehrere Mal hintereinander, kann überprüft werden, ob sich der Client an den Server annähert und der Synchronisationsalgorithmus somit funktioniert. Die interessanteste Frage dabei ist natürlich, *mit welcher Genauigkeit* dies möglich ist.

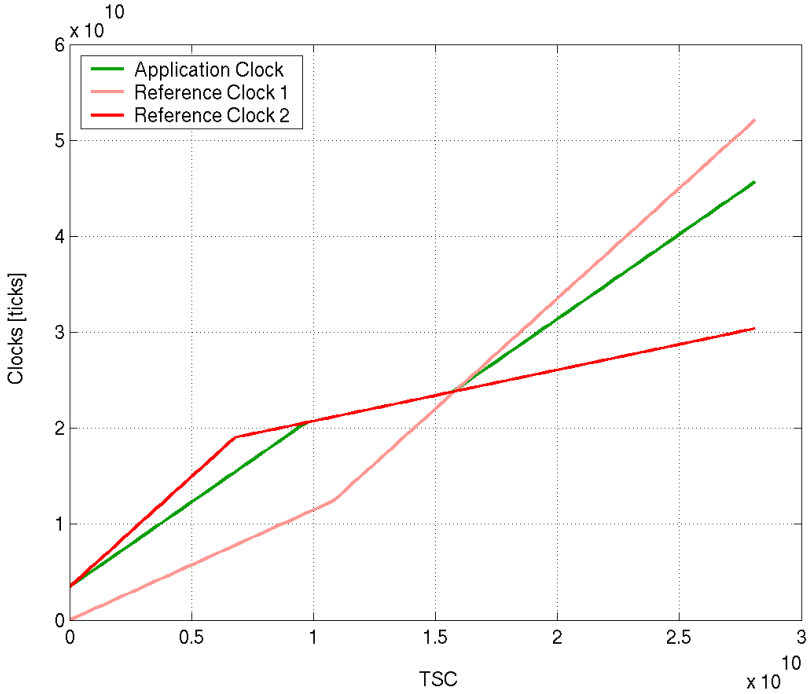

Abbildung 3.2: Trace eines Testlaufs

Das Problem, das sich bei der Realisierung diese idealen Prinzips stellt, ist die Frage, wie man zwei Uhren möglichst *gleichzeitig* ausliest. Dabei verfolgten wir folgende Idee: ein externes "Gerät" wird gleichzeitig an zwei Computern, nämlich den Server und den Client, über den Parallelport angeschlossen. Es generiert sodann zur exakt gleichen Zeit eine positive Spannungsflanke an einem bestimmten Pin. Dieser wird vom Kernel als Interrupt interpretiert. Ein Interrupt-Handler wird aufgerufen, der so modifiziert wurde, dass er vom Uhrenobjekt einen Zeitstempel nimmt und diesen in eine Datei ablegt. Da dies sowohl beim Server- als auch beim Client-PC geschieht, liegen Zeitstempel vor, die miteinander verglichen werden dürfen.

Für mehr Details sei hier auf die Dokumentation der Diplomarbeit von Daniel Sigg und Eric Schreiber verwiesen.

3.3.2 Elektronik

Für das obig beschriebene Prinzip ist also ein externes “Gerät” nötig, das solche Spannungsflanken generiert. Es handelt sich dabei um eine einfache elektronische Schaltung auf einer Platine, die über zwei Ausgänge in Form je eines Parallelportanschlusses verfügt. Diese Schaltung wurde von mir entworfen und implementiert. Ich möchte dabei jedoch betonen, dass ich dazu nur dank der grosszügigen Unterstützung meines Kommilitonen Matthias Auf der Maur in der Lage war.

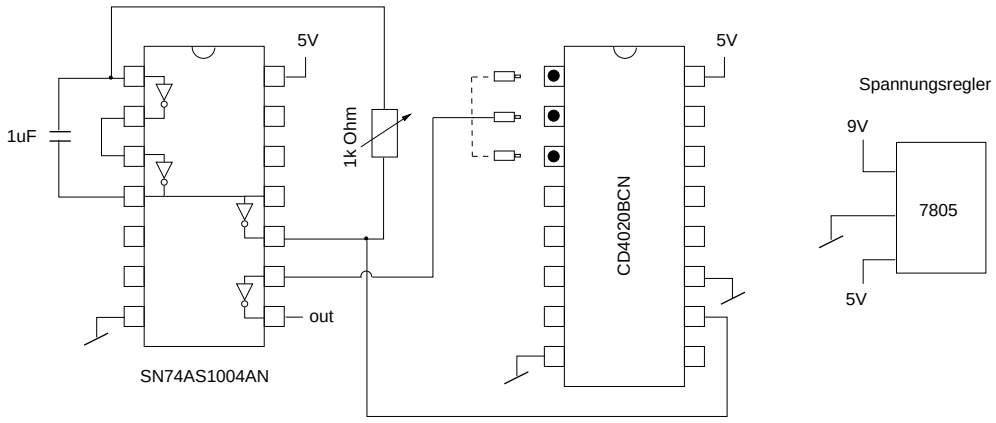

Abbildung 3.3: Schema der Elektronik

Die Schaltung ist in Abbildung 3.3 gezeigt und funktioniert sehr einfach: drei in Serie geschaltete Inverter haben an zwei verschiedenen Stellen je eine Rückkopplung mit einem Kondensator und einem Potentiometer. Die Frequenz der dadurch verursachten Viereckspannung wäre nun für unsere Zwecke viel zu schnell und muss somit mit einem Zähler heruntergeteilt werden. Damit man die resultierende langsame Frequenz innerhalb gewisser Schranken wählen kann, besteht die Möglichkeit, das Ausgangssignal an verschiedenen Orten abzugreifen. Das wurde realisiert durch einen kleinen Stecker, der jeweils verschiedene Ausgangspins des Zählers abgreifen kann. Die Periodendauer hängt stark vom Widerstandswert des Potentiometers ab und liegt je nach Einstellung ungefähr zwischen einer Sekunde und einer halben Minute.

Kapitel 4

Schlussfolgerungen

Zusammenfassend kann folgendes festgehalten werden:

- Das Uhrenobjekt ist erfolgreich entwickelt, implementiert und validiert.
- Die elektronische Schaltung für die Validierung der Uhrensynchronisation über WLAN ist implementiert und funktioniert einwandfrei.
- Was das Gebiet meiner Semesterarbeit anbetrifft, ist alles wichtige abgeschlossen und bedarf keiner Fortsetzung.
- Wie weiter oben erwähnt wurde, kann die Funktion `sys_gettime_appclk()` nur für eine einzige Applikationsuhr benutzt werden.
- Die Speicherallokation für die Uhren im Kernel-Modul könnte eventuell noch dynamsich mittels `kmalloc()` geschehen. Das würde das Memory-Management verschönern und zudem eine obere Grenze an maximal möglichen Uhren hinfällig machen.

Kapitel 5

Persönliche Erfahrungen und Dank

Als ich die Ausschreibung für diese Semesterarbeit zum ersten Mal gelesen hatte, war ich geneigt, sie als zu schwierig auf die Seite zu legen. Dennoch interessierte ich mich sehr für eine Arbeit, die mit UNIX/Linux und insbesondere mit dessen Kernel zu tun hat. Ich meldete mich dann doch beim zuständigen Assistenten und freute mich schliesslich sehr auf die Herausforderung, die genau in dem gesuchten Bereich angesiedelt war.

Ich wurde nicht enttäuscht. Mit viel Freude machte ich mich an die überwiegend neue Materie heran und erlebte schon früh Erfolgserlebnisse. Schon bald hatte ich mein erstes eigenes Kernelmodul geschrieben. Oft bedauerte ich es, dass ich durch Vorlesungen und ähnliches bei der interessanten Arbeit unterbrochen wurde.

Gegen Schluss des Semesters geriet ich dann ziemlich stark unter Zeitdruck, da nicht auf die Validierungsarbeiten verzichtet werden konnte und sich gleichzeitig der Vortrag und die Dokumentation nicht selbst erledigen wollten. Ich habe dies als sehr belastend erlebt, versuchte es jedoch als Vorbereitung für den allgegenwärtigen Termindruck in der Berufswelt zu sehen. Im Nachhinein bin ich mit den Ergebnissen der Validierung und der Präsentation meiner Arbeit sehr zufrieden.

Es bleiben noch einige Worte zu sagen zu der Teamarbeit mit Daniel Sigg und Eric Schreiber. Um vorne zu beginnen, war es ein sehr glücklicher Umstand, dass wir unseren Arbeitsplatz im gleichen Raum hatten. Dadurch wurde uns eine permanente und unkomplizierte Kommunikation ermöglicht. Leider machte ich die Erfahrung, dass in meiner Abwesenheit, die im Rahmen einer Semesterarbeit unumgänglich war, z.T. gemeinsam besprochene Entscheide abgeändert wurden und ich dies erst eine gewisse Zeit später erfuhr. Dabei handelte es sich jedoch eher um kleine Dinge, die ich mit kleinem Aufwand anpassen konnte. Im grossen und ganzen habe ich die Zusammenarbeit jedoch als sehr positiv erlebt und hätte auf keinen Fall tauschen wollen mit einer Arbeit im Elfenbeinturm.

Speziellen Dank möchte ich aussprechen:

- *Philipp Blum*, meinem Betreuer, für seine sehr engagierte und kompetente

Unterstützung und Zusammenarbeit.

- *Roman Hoog Antink*, meinem Kommilitonen, der mir immer wieder hilfreiche Tipps in Zusammehang mit dem Linux-Kernel geben konnte.
- *Matthias Auf der Maur*, meinem Kommilitonen, der trotz eigener Semesterarbeit bereit war, mir tatkräftige Hilfe zu leisten bei der Entwicklung der elektronischen Schaltung.
- *Yvonne Schnetzler*, meiner Kollegin, der ich das beste Titelblatt verdanke, das je eine Arbeit von mir gekürt hat.
- *Jérémie Moser-Guéneau*, meiner lieben Frau, die auch in den belastenden Zeiten Geduld mit mir hatte und mich ermutigte.
- *Gott*, dem Schöpfer von Himmel und Erde, der meinem Leben Sinn und Halt gibt.

Literaturverzeichnis

[1] Rüdiger Reischuk und Ed Wimmers Danny Dolve, Ray Strong. *A Decentralized High Performance Time Service Architecture.* 1995.

[2] Daniel Sigg und Eric Schreiber. *Hochpräzise Uhrensynchronisation für Wireless LAN.* Diplomarbeit, ETH Zurich, 2002.

[3] Frank Schmuck und Flavin Cristian. *Continuous Clock Amortization Need Not Affect the Precision of a Clock Synchronization Algorithm.* Proceedings of the Nineth ACM Symposion on Principles of Distributed Computing, pages 133-143, 1990.

[4] Daniel P.Bovet und Marco Cesati. *Understanding the LINUX Kernel.* O'Reilly, 2001.

Kapitel 6

Weblinks

URL	Beschreibung
www.opensource.org	Geburtsstätte der GPL
www.linuxdoc.org	Linux Documentation Project
www.freshmeat.net	Software-Archiv
www.kernel.org	Linux Kernelarchiv
www.linux.it/kerneldocs/ksys/ksys.html	Kernel System Calls
www.suse.de	Deutscher Linux-Distributor
www.weinelt.de/latex/	LATEX-Index

Anhang A

Sourcen

A.1 Makefile

```
KERNELFLAGS = -Wall -I/lib/modules/‘uname -r‘/build/include -I.
-D__KERNEL__ -DMODULE -Wstrict-prototypes -O2 -c

all: clkobj.o clkobj_client

clkobj.o: clkobj.c
        gcc $(KERNELFLAGS) clkobj.c

clkobj_client: clkobj_client.o wrapperfunctions.o

clkobj_client.o: clkobj_client.c
        gcc -I. -O2 -c clkobj_client.c

%.o: %.s
        nasm -g -f elf -o $@ $<

clean:
        rm -f *.o clkobj_client *~
```

A.2 Das Uhrenobjekt

A.2.1 clkobj.h

```
/*
  clkobj.h

  This kernel-module stores logical software-clocks and offers some system
  calls to operate on them

  Written by Andreas Moser (c) 2002, GPL
*/

/* Argument struct for communication with applications */
struct clkval
{
    unsigned long long alpha;
    int beta;
    unsigned long long tsc;
    unsigned int cpu_khz;
};

/* Clock descriptor */
struct clk_desc
{
    int desc;
    unsigned int cpu_khz;
    int max_clk_drift_ppm;
    unsigned long ip;
};

#ifdef __KERNEL__

#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <asm/msr.h>
#include <asm/uaccess.h>
#include <linux/mm.h>
#include <linux/slab.h>
#include <asm/io.h>

#define EXPORT_SYMTAB
#define __NR_initclk 230
#define __NR_setclk 231
#define __NR_settime 232
#define __NR_setrate 233
#define __NR_getrawtime 234
#define __NR_readtime 235
#define __NR_init_appclk 236
```

```
#define __NR_gettime_appclk 237
#define __NR_nsec2ticks 238
#define __NR_ticks2nsec 239
#define __NR_calctime 240
#define MAXNUMBEROFCLOCKS 10
#define BETASHIFT 24
#define MAX_CLK_DRIFT_PPM 50
#define CALIBRATE_LATCH (5 * LATCH)
#define CALIBRATE_TIME  (5 * 1000020/HZ)

/*
  System call table. Defined as external. Will be
  filled up by the kernel when insmod'ed
*/
extern long sys_call_table[];

/* Pointer to the old system calls */
#ifndef _ORINOCO_H
static int (*old_call_initclk)(int);
static int (*old_call_setclk)(int);
static int (*old_call_settime)(int);
static int (*old_call_setrate) (int);
static int (*old_call_getrawtime)(int);
static int (*old_call_readtime)(int);
static int (*old_call_init_appclk)(int);
static int (*old_call_gettime_appclk)(int);
static int (*old_call_nsec2ticks)(int);
static int (*old_call_ticks2nsec)(int);
static int (*old_call_calctime)(int);
#endif

/* Clock for internal use */
struct clk
{
    unsigned long long alpha;
    int beta;
    unsigned long ip;
};

/* Application clock for internal use */
struct appclk
{
    unsigned long long alpha;
    int beta;
    int max_slope;  /* positive or negative, maximal slope
                       -> upper and lower boundary */
    int number_of_parents;
    int *parents;
};
```

```
/* Initializes a clock */
extern asmlinkage int sys_initclk(unsigned long ip, struct clk_desc *rd);

/* Set clock by new time [ticks] and beta (changes alpha and beta) */
extern asmlinkage int sys_setclk(int clk_desc,
unsigned long long *new_time, int *new_beta);

/* Set time by new time [ticks] (changes alpha only) */
extern asmlinkage int sys_settime(int clk_desc,
unsigned long long *new_time);

/* Set new rate and don't let the time jump (changes alpha and beta) */
extern asmlinkage int sys_setrate(int clk_desc, int *new_beta);

/* Get time of clock clk_desc in terms of raw alpha, beta and tsc */
extern asmlinkage int sys_getrawtime(int clk_desc, struct clkval *cv);

/* Calculates time [ticks] of clk clk_desc */
extern asmlinkage int sys_readtime(int clk_desc,
unsigned long long *now);

/* ticks <-> nanoseconds conversions */
extern asmlinkage int sys_nsec2ticks(unsigned long long *nsec,
unsigned long long *ticks, unsigned int cpu_khz);

extern asmlinkage int sys_ticks2nsec(unsigned long long *ticks,
unsigned long long *nsec, unsigned int cpu_khz);

int div_mult(unsigned long long *dividend, unsigned int divisor,
unsigned int mult, unsigned long long *result);

extern asmlinkage int sys_calctime(unsigned long long *alpha,
unsigned long long *tsc, int *beta, unsigned long long *now);

/* Returns pointer to clock-struct clk_desc */
extern asmlinkage int sys_getclkpntr(int clk_desc, struct clk **c);

/*
  Initializes an application clock
  Not EXPORTED (user space available only)
*/
asmlinkage int sys_init_appclk(int *appclk_desc, int max_slope,
int number_of_parents, int *parents);
```

```
/*
  Get time of application clock appclk_desc
  Not EXPORTED (user space available only)
*/
asmlinkage int sys_gettime_appclk(int appclk_desc,
unsigned long long *nsec);

/*
  Elicit frequency of cpu-clock doing the same as at boot-time
*/
unsigned long calibrate_tsc(void);

EXPORT_SYMBOL_NOVERS(sys_initclk);
EXPORT_SYMBOL_NOVERS(sys_setclk);
EXPORT_SYMBOL_NOVERS(sys_settime);
EXPORT_SYMBOL_NOVERS(sys_setrate);
EXPORT_SYMBOL_NOVERS(sys_getrawtime);
EXPORT_SYMBOL_NOVERS(sys_readtime);
EXPORT_SYMBOL_NOVERS(sys_getclkpntr);
EXPORT_SYMBOL_NOVERS(sys_nsec2ticks);
EXPORT_SYMBOL_NOVERS(sys_ticks2nsec);
EXPORT_SYMBOL_NOVERS(sys_calctime);

/* Our clock-parameters as dynamically allocated arrays */
struct clk *clk_array;
struct appclk *appclk_array;
int count_clk;  /* index of clk_array + 1 */
int count_appclk;  /* index of appclk_array + 1 */

unsigned long cpu_khz;
unsigned long tsc_quotient;
unsigned long last_tsc_low; /* lsb 32 bits of Time Stamp Counter */
unsigned long long *last_appclk_time;
unsigned long long tsc_old;
unsigned long tsc_low_old;
unsigned long tsc_high_old;

#endif
```

A.2.2 clkobj.c

```
/*
  clkobj.c

  This kernel-module stores logical software-clocks and offers some system
  calls to operate on them

  Written by Andreas Moser (c) 2002, GPL
*/

#define EXPORT_SYMTAB

#include <clkobj.h>

/* Initializes a clock */
asmlinkage int sys_initclk(unsigned long ip, struct clk_desc *rd) {
    struct timeval tv;
    unsigned long long now_nsec, now_ticks;
    unsigned long long tsc;
    unsigned long tsc_low, tsc_high;

    if (count_clk == MAXNUMBEROFCLOCKS) {
        return -EINVAL;
    }
    do_gettimeofday(&tv);
    now_nsec = (unsigned long long)tv.tv_sec * 1000000000 +
               (unsigned long long)tv.tv_usec * 1000;
    sys_nsec2ticks(&now_nsec, &now_ticks, cpu_khz);
    rdtsc(tsc_low, tsc_high);
    tsc = ((unsigned long long)tsc_high << 32) + (unsigned long long) tsc_low;

    /* Store values in clock-struct */
    clk_array[count_clk].alpha = now_ticks - tsc;
    clk_array[count_clk].beta = 0;
    clk_array[count_clk].ip = ip;

    count_clk++;

    /* Store data for calling routine in clock-descriptor  */
    rd->desc = count_clk;    /* clock descriptor is count_clk */
    rd->max_clk_drift_ppm = MAX_CLK_DRIFT_PPM;
    rd->cpu_khz = cpu_khz;
    rd->ip = ip;
    return 1;
}
```

```
/* Set clock by new time [ticks] and beta (changes alpha and beta) */
asmlinkage int sys_setclk(int clk_desc, unsigned long long *new_time,
int *new_beta) {
    unsigned long long tsc;
    unsigned long tsc_low, tsc_high;
    unsigned long long low_temp, high_temp;

    if (clk_desc > count_clk) {
        return -EINVAL;
    }

    /*
      alpha = new_time - tsc - new_beta*tsc
    */
    rdtsc(tsc_low, tsc_high);
    tsc = ((unsigned long long)tsc_high << 32) + (unsigned long long) tsc_low;

    if(*new_beta >= 0) {
        high_temp = (((unsigned long long)tsc_high << 32) >> BETASHIFT) *
                    (unsigned long long)(*new_beta);
        low_temp =  ((((unsigned long long)tsc_low << 32) >> BETASHIFT) *
                    (unsigned long long)(*new_beta)) >> 32;
        clk_array[clk_desc-1].alpha = *new_time - tsc - (high_temp + low_temp);
    }
    else if(*new_beta < 0) {
        high_temp = (((unsigned long long)tsc_high << 32) >> BETASHIFT) *
                    (unsigned long long)abs(*new_beta);
        low_temp = ((((unsigned long long)tsc_low << 32) >> BETASHIFT) *
                   (unsigned long long)abs(*new_beta)) >> 32;
        clk_array[clk_desc-1].alpha = *new_time - tsc + high_temp + low_temp;
    }
    clk_array[clk_desc-1].beta = *new_beta;
    return 1;
}


/* Set time by new time [ticks] (changes alpha only) */
asmlinkage int sys_settime(int clk_desc, unsigned long long *new_time) {
    unsigned long long tsc;
    unsigned long tsc_low, tsc_high;
    unsigned long long low_temp, high_temp;

    if (clk_desc > count_clk) {
        return -EINVAL;
    }
```

```
    /*
      alpha = new_time - tsc - new_beta*tsc
    */
    rdtsc(tsc_low, tsc_high);
    tsc = ((unsigned long long)tsc_high << 32) + (unsigned long long) tsc_low;

    if(clk_array[clk_desc-1].beta >= 0) {
        high_temp = (((unsigned long long)tsc_high << 32) >> BETASHIFT) *
                    (unsigned long long)(clk_array[clk_desc-1].beta);
        low_temp =  ((((unsigned long long)tsc_low << 32) >> BETASHIFT) *
                    (unsigned long long)(clk_array[clk_desc-1].beta)) >> 32;
        clk_array[clk_desc-1].alpha = *new_time - tsc - (high_temp + low_temp);
    }
    else if(clk_array[clk_desc-1].beta < 0) {
        high_temp = (((unsigned long long)tsc_high << 32) >> BETASHIFT) *
                    (unsigned long long)abs(clk_array[clk_desc-1].beta);
        low_temp =  ((((unsigned long long)tsc_low << 32) >> BETASHIFT) *
                    (unsigned long long)abs(clk_array[clk_desc-1].beta)) >> 32;
        clk_array[clk_desc-1].alpha = *new_time - tsc + high_temp + low_temp;
    }
    return 1;
}


/* Set new rate and don't let the time jump (changes alpha and beta) */
asmlinkage int sys_setrate(int clk_desc, int *new_beta) {
    unsigned long long tsc;
    unsigned long tsc_low, tsc_high;
    unsigned long long low_temp, high_temp;
    int beta_diff;

    unsigned long long now1, now2;
    struct clkval cv1, cv2;
    sys_readtime(clk_desc, &now1);
    sys_getrawtime(clk_desc, &cv1);
    /*
      new_alpha = old_alpha + (old_beta - new_beta)*tsc
    */
    rdtsc(tsc_low, tsc_high);
    beta_diff = clk_array[clk_desc-1].beta - *new_beta;

    if(beta_diff >= 0) {
        high_temp = (((unsigned long long)tsc_high << 32) >> BETASHIFT) *
                    (unsigned long long)beta_diff;
        low_temp =  ((((unsigned long long)tsc_low << 32) >> BETASHIFT) *
                    (unsigned long long)beta_diff) >> 32;
        clk_array[clk_desc-1].alpha = clk_array[clk_desc-1].alpha +
                                      high_temp + low_temp;
    }
```

```
    else if(beta_diff < 0) {
        high_temp = (((unsigned long long)tsc_high << 32) >> BETASHIFT) *
                    (unsigned long long)abs(beta_diff);
        low_temp =  ((((unsigned long long)tsc_low << 32) >> BETASHIFT) *
                    (unsigned long long)abs(beta_diff)) >> 32;
        clk_array[clk_desc-1].alpha = clk_array[clk_desc-1].alpha -
                                      high_temp - low_temp;
    }
    clk_array[clk_desc-1].beta = *new_beta;
    sys_readtime(clk_desc, &now2);
    sys_getrawtime(clk_desc, &cv2);
    tsc = ((unsigned long long)tsc_high << 32) + (unsigned long long) tsc_low;
    return 1;
}


/* Get time of clock clk_desc in terms of raw alpha, beta and tsc */
asmlinkage int sys_getrawtime(int clk_desc, struct clkval *cv) {
    unsigned long long tsc;
    unsigned long tsc_low;
    unsigned long tsc_high;

    if (clk_desc > count_clk) {
        return -EINVAL;
    }

    rdtsc(tsc_low, tsc_high);
    tsc = ((unsigned long long)tsc_high << 32) + (unsigned long long) tsc_low;

    cv->alpha = clk_array[clk_desc-1].alpha;
    cv->beta = clk_array[clk_desc-1].beta;
    cv->tsc = tsc;
    cv->cpu_khz = cpu_khz;

    return 1;
}

/* Calculates time [ticks] of clk clk_desc */
extern asmlinkage int sys_readtime(int clk_desc, unsigned long long *now) {
    unsigned long long tsc;
    unsigned long tsc_low, tsc_high;
    unsigned long long low_temp, high_temp;

    if (clk_desc > count_clk) {
        return -EINVAL;
    }
```

```
    /*
      now = alpha + tsc + tsc * beta
    */
    rdtsc(tsc_low, tsc_high);
    tsc = ((unsigned long long)tsc_high << 32) + (unsigned long long) tsc_low;

    if(clk_array[clk_desc-1].beta >= 0) {
        high_temp = (((unsigned long long)tsc_high << 32) >> BETASHIFT) *
                    (unsigned long long)(clk_array[clk_desc-1].beta);
        low_temp =  ((((unsigned long long)tsc_low << 32) >> BETASHIFT) *
                    (unsigned long long)(clk_array[clk_desc-1].beta)) >> 32;
        *now = clk_array[clk_desc-1].alpha + tsc + high_temp + low_temp;
    }
    else if(clk_array[clk_desc-1].beta < 0) {
        high_temp = (((unsigned long long)tsc_high << 32) >> BETASHIFT) *
                    (unsigned long long)abs(clk_array[clk_desc-1].beta);
        low_temp =  ((((unsigned long long)tsc_low << 32) >> BETASHIFT) *
                    (unsigned long long)abs(clk_array[clk_desc-1].beta)) >> 32;
        *now = clk_array[clk_desc-1].alpha + tsc - high_temp - low_temp;
    }
    return 1;
}

/* Calculates time [ticks] with given arguments */
extern asmlinkage int sys_calctime(unsigned long long *alpha,
unsigned long long *tsc, int *beta, unsigned long long *now) {
    unsigned long long tsc_low, tsc_high;
    unsigned long long low_temp, high_temp;

    tsc_high = (*tsc>>32);
    tsc_low = (*tsc<<32);

    if(*beta >= 0) {
        high_temp = ((tsc_high << 32) >> BETASHIFT) *
                    (unsigned long long)(*beta);
        low_temp =  ((tsc_low >> BETASHIFT) *
                    (unsigned long long)(*beta)) >> 32;
        *now = *alpha + *tsc + high_temp + low_temp;
    }
    else if(*beta < 0) {
        high_temp = ((tsc_high << 32) >> BETASHIFT) *
                    (unsigned long long)abs(*beta);
        low_temp =  ((tsc_low >> BETASHIFT) *
                    (unsigned long long)abs(*beta)) >> 32;
        *now = *alpha + *tsc - high_temp - low_temp;
    }
    return 1;
}
```

```
int div_mult(unsigned long long *dividend, unsigned int divisor,
unsigned int mult, unsigned long long *result)
{
    /* dividend is divided by 'divisor' and multiplied by 'mult'.
     * the basic task of this function is a division of a long long
     * type by an unsigned int. This is done using asember similar
     * as in /asm/div64.h.
     * the remainder of the division (__mod) is multiplied by mult
     * and the same operation as before is applied once more
     * to the result (temp_L1).
     */

    unsigned long long quotient, temp_L1, temp_L2;
    {
        unsigned long __upper, __low, __high, __mod;

        asm("":"=a" (__low), "=d" (__high):"A" (*dividend));
        __upper = __high;
        if (__high) {
                __upper = __high % divisor;
                __high = __high / divisor; }
        asm("divl %2":"=a" (__low), "=d" (__mod):"rm" (divisor),
            "0" (__low), "1" (__upper));
        asm("":"=A" (quotient):"a" (__low),"d" (__high));
        temp_L1 =  (unsigned long long) __mod * mult;
        asm("":"=a" (__low), "=d" (__high):"A" (temp_L1));
        __upper = __high;
        if (__high) {
                __upper = __high % divisor;
                __high = __high / divisor; }
        asm("divl %2":"=a" (__low), "=d" (__mod):"rm" (divisor),
            "0" (__low), "1" (__upper));
        asm("":"=A" (temp_L2):"a" (__low),"d" (__high));
        *result =  mult * quotient + temp_L2;
    }
    return 1;
}

extern asmlinkage int sys_nsec2ticks(unsigned long long *nsec,
unsigned long long *ticks, unsigned int cpu_khz)
{
    /* transform nanoseconds into ticks */
    div_mult(nsec, 1000000, cpu_khz, ticks);
    return 1;
}
```

```
extern asmlinkage int sys_ticks2nsec(unsigned long long *ticks,
unsigned long long *nsec, unsigned int cpu_khz)
{
    /* transform ticks into nanoseconds */
    div_mult(ticks, cpu_khz, 1000000, nsec);
    return 1;
}

/* Returns pointer to clock-struct clk_desc */
extern asmlinkage int sys_getclkpntr(int clk_desc, struct clk **c) {
    *c = &clk_array[clk_desc-1];
    return 1;
}

/*
  Initializes an application clock
  Not EXPORTED (user space available only)
*/
asmlinkage int sys_init_appclk(int *appclk_desc, int max_slope,
int number_of_parents, int *parents) {
    int i;
    struct timeval tv;
    unsigned long long now_nsec, now_ticks;
    unsigned long long tsc;
    unsigned long tsc_low, tsc_high;

    if (count_appclk == MAXNUMBEROFCLOCKS) {
        return -EINVAL;
    }

    /*
      Provide max_slope to be positive
      (to distinguish upper and lower boundary)
    */
    if (max_slope < 0) max_slope = -max_slope;

    /* Allocate memory for parent clocks */
    appclk_array[count_appclk].parents = (int*)kmalloc(
                                          number_of_parents * sizeof(int),
                                          GFP_KERNEL);

    do_gettimeofday(&tv);
    now_nsec = (unsigned long long)tv.tv_sec * 1000000000 +
               (unsigned long long)tv.tv_usec * 1000;
    sys_nsec2ticks(&now_nsec, &now_ticks, cpu_khz);
    rdtsc(tsc_low, tsc_high);
    tsc = ((unsigned long long)tsc_high << 32) + (unsigned long long) tsc_low;
```

```
    /* Store values in appclock-struct */
    appclk_array[count_appclk].alpha = now_ticks - tsc;
    appclk_array[count_appclk].beta = 0;
    appclk_array[count_appclk].max_slope = max_slope;
    appclk_array[count_appclk].number_of_parents = number_of_parents;
    for (i=0; i<number_of_parents; i++) {
        appclk_array[count_appclk].parents[i] = parents[i]-1;
    }
    last_appclk_time[count_appclk] = (unsigned long long)0;

    /* Finish */
    count_appclk++;
    *appclk_desc = count_appclk;  /* clock descriptor is count_appclk */
    return 1;
}


/*
  Get time of application clock appclk_desc
  Not EXPORTED (user space available only)
*/
asmlinkage int sys_gettime_appclk(int appclk_desc, unsigned long long *nsec) {
    int i;
    unsigned long long alpha;
    int beta;
    unsigned long long high_temp;
    unsigned long long low_temp;
    unsigned long long current_time;
    unsigned long long best_time;
    unsigned long long best_alpha;
    int best_beta;
    unsigned long long tsc;
    unsigned long tsc_low;
    unsigned long tsc_high;
    unsigned long long temp1;
    unsigned long long temp2;
    unsigned long long upper_limit;
    unsigned long long lower_limit;

    best_time=(unsigned long long)0;
    best_alpha=(unsigned long long)0;
    best_beta=0;

    rdtsc(tsc_low, tsc_high);
    tsc = ((unsigned long long)tsc_high << 32) + (unsigned long long)tsc_low;
```

```
/* Find best clock (maximum of reference clocks) */
for (i=0; i<appclk_array[appclk_desc-1].number_of_parents; i++) {
    /* Compute time of each reference clock */
    alpha = clk_array[appclk_array[appclk_desc-1].parents[i]].alpha;
    beta = clk_array[appclk_array[appclk_desc-1].parents[i]].beta;
    if(beta >= 0) {
        high_temp = (((unsigned long long)tsc_high << 32) >> BETASHIFT) *
                    (unsigned long long)(beta);
        low_temp =  ((((unsigned long long)tsc_low << 32) >> BETASHIFT) *
                    (unsigned long long)(beta)) >> 32;
        current_time = alpha + tsc + high_temp + low_temp;
    }
    else if(beta < 0) {
        high_temp = (((unsigned long long)tsc_high << 32) >> BETASHIFT) *
                    (unsigned long long)abs(beta);
        low_temp =  ((((unsigned long long)tsc_low << 32) >> BETASHIFT) *
                    (unsigned long long)abs(beta)) >> 32;
        current_time = alpha + tsc - high_temp - low_temp;
    }

    if (current_time > best_time) {
        best_time = current_time;
        best_alpha = alpha;
        best_beta = beta;
    }
}

/* First time: just return the time of the best reference clock */
if (last_appclk_time[appclk_desc-1] == 0) {
    printk("First time... return best_time\n");
    last_appclk_time[appclk_desc-1] = best_time;
    tsc_old = tsc;
    tsc_low_old = tsc_low;
    tsc_high_old = tsc_high;
    sys_ticks2nsec(&best_time, &best_time, cpu_khz);
    *nsec = best_time;
    return 1;
}

/* Copy alpha and beta to application clock */
appclk_array[appclk_desc-1].alpha = best_alpha;
appclk_array[appclk_desc-1].beta = best_beta;

/* Check whether our time value is within the allowed boundaries */
/* Improve readability by pre-computing some parts */
/* temp1 = max_slope * tsc */
high_temp = (((unsigned long long)tsc_high << 32) >> BETASHIFT) *
            (unsigned long long)appclk_array[appclk_desc-1].max_slope;
low_temp =  ((((unsigned long long)tsc_low << 32) >> BETASHIFT) *
```

```
                (unsigned long long)appclk_array[appclk_desc-1].max_slope)
                >> 32;
    temp1 = high_temp + low_temp;

    /* temp2 = max_slope * tsc_old */
    high_temp = (((unsigned long long)tsc_high_old << 32) >> BETASHIFT) *
                (unsigned long long)appclk_array[appclk_desc-1].max_slope;
    low_temp =  ((((unsigned long long)tsc_low_old << 32) >> BETASHIFT) *
                (unsigned long long)appclk_array[appclk_desc-1].max_slope)
                >> 32;
    temp2 = high_temp + low_temp;

    upper_limit = tsc - tsc_old + last_appclk_time[appclk_desc-1] +
                  temp1 - temp2;
    lower_limit = tsc - tsc_old + last_appclk_time[appclk_desc-1] -
                  temp1 + temp2;

    if (best_time > upper_limit) {
        /*
          Our timevalue would jump too much upwards
          -> return upper boundary value
        */
        last_appclk_time[appclk_desc-1] = upper_limit;
        sys_ticks2nsec(&upper_limit, &upper_limit, cpu_khz);
        *nsec = upper_limit;
    }
    else if (best_time < lower_limit) {
        /*
          Our timevalue would jump too much downwards
          -> return lower boundary value
        */
        last_appclk_time[appclk_desc-1] = lower_limit;
        sys_ticks2nsec(&lower_limit, &lower_limit, cpu_khz);
        *nsec = lower_limit;
    }
    else {
        /* we are perfectly within the boundaries */
        last_appclk_time[appclk_desc-1] = best_time;
        sys_ticks2nsec(&best_time, &best_time, cpu_khz);
        *nsec = best_time;
    }
    tsc_old = tsc;
    tsc_low_old = tsc_low;
    tsc_high_old = tsc_high;

    return 1;
}
```

```
/*
  Elicit frequency of cpu-clock doing the same as at boot-time
*/
unsigned long calibrate_tsc(void)
{
       /* Set the Gate high, disable speaker */
        outb((inb(0x61) & ~0x02) | 0x01, 0x61);

       /*
        * Now let's take care of CTC channel 2
        *
        * Set the Gate high, program CTC channel 2 for mode 0,
        * (interrupt on terminal count mode), binary count,
        * load 5 * LATCH count, (LSB and MSB) to begin countdown.
        */
        outb(0xb0, 0x43);                   /* binary, mode 0, LSB/MSB, Ch 2 */
        outb(CALIBRATE_LATCH & 0xff, 0x42); /* LSB of count */
        outb(CALIBRATE_LATCH >> 8, 0x42);   /* MSB of count */

        {
                unsigned long startlow, starthigh;
                unsigned long endlow, endhigh;
                unsigned long count;

                rdtsc(startlow,starthigh);
                count = 0;
                do {
                        count++;
                } while ((inb(0x61) & 0x20) == 0);
                rdtsc(endlow,endhigh);

                last_tsc_low = endlow;

                /* Error: ECTCNEVERSET */
                if (count <= 1)
                        goto bad_ctc;

                /* 64-bit subtract - gcc just messes up with long longs */
                __asm__("subl %2,%0\n\t"
                        "sbbl %3,%1"
                        :"=a" (endlow), "=d" (endhigh)
                        :"g" (startlow), "g" (starthigh),
                         "0" (endlow), "1" (endhigh));

                /* Error: ECPUTOOFAST */
                if (endhigh)
                        goto bad_ctc;
```

```
                /* Error: ECPUTOOSLOW */
                if (endlow <= CALIBRATE_TIME)
                        goto bad_ctc;

                __asm__("divl %2"
                        :"=a" (endlow), "=d" (endhigh)
                        :"r" (endlow), "0" (0), "1" (CALIBRATE_TIME));

                return endlow;
        }
        /*
         * The CTC wasn't reliable: we got a hit on the very first read,
         * or the CPU was so fast/slow that the quotient wouldn't fit
         * in 32 bits..
         */
bad_ctc:
        return 0;
}

/*
  Initialization of the module (executed once when 'insmod'ed)
  replaces (unused) system calls by our own implementations
  This function is not time-critical
*/
int init_module(void)
{
    unsigned long tsc_quotient;

    old_call_initclk = (int(*)(int))(sys_call_table[__NR_initclk]);
    sys_call_table[__NR_initclk] = (unsigned long)sys_initclk;

    old_call_setclk = (int(*)(int))(sys_call_table[__NR_setclk]);
    sys_call_table[__NR_setclk] = (unsigned long)sys_setclk;

    old_call_settime = (int(*)(int))(sys_call_table[__NR_settime]);
    sys_call_table[__NR_settime] = (unsigned long)sys_settime;

    old_call_setrate = (int(*)(int))(sys_call_table[__NR_setrate]);
    sys_call_table[__NR_setrate] = (unsigned long)sys_setrate;

    old_call_getrawtime = (int(*)(int))(sys_call_table[__NR_getrawtime]);
    sys_call_table[__NR_getrawtime] = (unsigned long)sys_getrawtime;

    old_call_readtime = (int(*)(int))(sys_call_table[__NR_readtime]);
    sys_call_table[__NR_readtime] = (unsigned long)sys_readtime;

    old_call_init_appclk = (int(*)(int))(sys_call_table[__NR_init_appclk]);
    sys_call_table[__NR_init_appclk] = (unsigned long)sys_init_appclk;
```

```
    old_call_gettime_appclk = (int(*)(int))(sys_call_table[__NR_gettime_appclk]);
    sys_call_table[__NR_gettime_appclk] = (unsigned long)sys_gettime_appclk;

    old_call_nsec2ticks = (int(*)(int))(sys_call_table[__NR_nsec2ticks]);
    sys_call_table[__NR_nsec2ticks] = (unsigned long)sys_nsec2ticks;

    old_call_ticks2nsec = (int(*)(int))(sys_call_table[__NR_ticks2nsec]);
    sys_call_table[__NR_ticks2nsec] = (unsigned long)sys_ticks2nsec;

    old_call_calctime = (int(*)(int))(sys_call_table[__NR_calctime]);
    sys_call_table[__NR_calctime] = (unsigned long)sys_calctime;

    clk_array = (struct clk*)kmalloc(
                            MAXNUMBEROFCLOCKS*sizeof(struct clk),
                            GFP_KERNEL);
    appclk_array = (struct appclk*)kmalloc(
                                  MAXNUMBEROFCLOCKS*sizeof(struct appclk),
                                  GFP_KERNEL);
    last_appclk_time = (unsigned long long*)kmalloc(
                                           MAXNUMBEROFCLOCKS*
                                           sizeof(unsigned long long),
                                           GFP_KERNEL);

    /* Calculate cpu_khz with tsc_quotient */
    tsc_quotient = calibrate_tsc();
    {       unsigned long eax=0, edx=1000;
            __asm__("divl %2"
            :"=a" (cpu_khz), "=d" (edx)
            :"r" (tsc_quotient),
            "0" (eax), "1" (edx));
    }

    printk("tsc-quotient: %lu\n", tsc_quotient);
    printk("cpu_khz:      %lu\n", cpu_khz);
    count_clk = 0;
    return 0;
}


/*
  Cleanup of the module (executed once when 'rmmod'ed)
  sets back the former system calls
*/
void cleanup_module(void)
{
    if (sys_call_table[__NR_initclk] != (unsigned long)sys_initclk) {
        printk("Somebody else also played with our ");
        printk("system call (initclk)\n");
        printk("The system may be left in ");
```

```
    printk("an unstable state!\n");
}
sys_call_table[__NR_initclk] = (unsigned long)old_call_initclk;

if (sys_call_table[__NR_setclk] != (unsigned long)sys_setclk) {
    printk("Somebody else also played with our ");
    printk("system call (setclk)\n");
    printk("The system may be left in ");
    printk("an unstable state!\n");
}
sys_call_table[__NR_setclk] = (unsigned long)old_call_setclk;

if (sys_call_table[__NR_settime] != (unsigned long)sys_settime) {
    printk("Somebody else also played with our ");
    printk("system call (settime)\n");
    printk("The system may be left in ");
    printk("an unstable state!\n");
}
sys_call_table[__NR_settime] = (unsigned long)old_call_settime;

if (sys_call_table[__NR_setrate] != (unsigned long)sys_setrate) {
    printk("Somebody else also played with our ");
    printk("system call (setrate)\n");
    printk("The system may be left in ");
    printk("an unstable state!\n");
}
sys_call_table[__NR_setrate] = (unsigned long)old_call_setrate;

if (sys_call_table[__NR_getrawtime] != (unsigned long)sys_getrawtime) {
    printk("Somebody else also played with our ");
    printk("system call (getrawtime)\n");
    printk("The system may be left in ");
    printk("an unstable state!\n");
}
sys_call_table[__NR_getrawtime] = (unsigned long)old_call_getrawtime;

if (sys_call_table[__NR_readtime] != (unsigned long)sys_readtime) {
    printk("Somebody else also played with our ");
    printk("system call (readtime)\n");
    printk("The system may be left in ");
    printk("an unstable state!\n");
}
sys_call_table[__NR_readtime] = (unsigned long)old_call_readtime;

if (sys_call_table[__NR_init_appclk] != (unsigned long)sys_init_appclk) {
    printk("Somebody else also played with our ");
    printk("system call (init_appclk)\n");
    printk("The system may be left in ");
    printk("an unstable state!\n");
```

```
    }
    sys_call_table[__NR_init_appclk] = (unsigned long)old_call_init_appclk;

    if (sys_call_table[__NR_gettime_appclk] !=
                                    (unsigned long)sys_gettime_appclk) {
        printk("Somebody else also played with our ");
        printk("system call (gettime_appclk)\n");
        printk("The system may be left in ");
        printk("an unstable state!\n");
    }
    sys_call_table[__NR_gettime_appclk] =
                                   (unsigned long)old_call_gettime_appclk;

    if (sys_call_table[__NR_nsec2ticks] != (unsigned long)sys_nsec2ticks) {
        printk("Somebody else also played with our ");
        printk("system call (nsec2ticks)\n");
        printk("The system may be left in ");
        printk("an unstable state!\n");
    }
    sys_call_table[__NR_nsec2ticks] = (unsigned long)old_call_nsec2ticks;

    if (sys_call_table[__NR_ticks2nsec] != (unsigned long)sys_ticks2nsec) {
        printk("Somebody else also played with our ");
        printk("system call (ticks2nsec)\n");
        printk("The system may be left in ");
        printk("an unstable state!\n");
    }
    sys_call_table[__NR_ticks2nsec] = (unsigned long)old_call_ticks2nsec;

    if (sys_call_table[__NR_calctime] != (unsigned long)sys_calctime) {
        printk("Somebody else also played with our ");
        printk("system call (ticks2nsec)\n");
        printk("The system may be left in ");
        printk("an unstable state!\n");
    }
    sys_call_table[__NR_calctime] = (unsigned long)old_call_calctime;

    kfree(clk_array);
    kfree(appclk_array);
    kfree(last_appclk_time);
}
```

A.2.3 clkobj_client.c

```
/*
  clkobj_client.c

  Client in user-space which uses the system calls
  provided by the kernel-module clkobj.o

  Written by Andreas Moser (c) 2002, GPL
*/

#include <stdio.h>
#include <clkobj.h>
#include <asm/msr.h>

extern int initclk(unsigned long ip, struct clk_desc *rd);
extern int setclk(int clk_desc, unsigned long long *new_time,
                  int *new_beta);
extern int settime(int clk_desc, unsigned long long *new_time);
extern int setrate(int clk_desc, int *new_beta);
extern int getrawtime(int clk_desc, struct clkval *cv);
extern int init_appclk(int *appclk_desc, int max_slope,
                       int number_of_parents, int *parents);
extern int gettime_appclk(int appclk_desc, unsigned long long *nsec);

int main() {
    int status, i;
    int appclk_desc;
    int max_slope;
    int number_of_parents;
    int parents[2];
    unsigned long my_ip;
    unsigned long long time;
    unsigned long long appclk_time;
    int beta;
    struct clk_desc cd;
    struct clk_desc cd2;
    int timeconst;
    int waitconst;
    FILE* file_id;
    unsigned long tsclow, tschigh;
    unsigned long long tsc;
    struct clkval cv;
    unsigned long long now;
    unsigned long long now2;

    my_ip = (unsigned long)120;
    timeconst = 25;
    waitconst = 300000;
    max_slope = 9000000;
```

```
number_of_parents = 2;
cd.desc = 1;

/* Prepare output file */
file_id = fopen("clkobj_client_output", "w+");

/* Initialize two reference clocks */
status = initclk(my_ip, &cd);
status = initclk(my_ip, &cd2);

/* The two reference clocks are different... */
status = readtime(cd2.desc, &time);
time = time - (unsigned long long)3000000000;
status = settime(cd2.desc, &time);
beta = 16777215;
status = setrate(cd.desc, &beta);
beta = 10000;
status = setrate(cd2.desc, &beta);

/* Initialize application clock */
parents[0] = cd.desc;
parents[1] = cd2.desc;
status = init_appclk(&appclk_desc, max_slope,
                     number_of_parents, parents);
for(i=0; i<25; i++) {
    status = gettime_appclk(appclk_desc, &appclk_time);
    getrawtime(cd.desc, &cv);
    readtime(cd.desc, &now);
    readtime(cd2.desc, &now2);
    ticks2nsec(&now, &now, cv.cpu_khz);
    ticks2nsec(&now2, &now2, cv.cpu_khz);
    fprintf(file_id, "tsc:\t%Lu\tbeta:\t%Lu\talpha:\t%Lu\tc_out:\t%Lu\n",
            cv.tsc, appclk_time, now, now2);
    usleep(waitconst);
}

beta = -9000000;
status = setrate(cd.desc, &beta);
for(i=0; i<15; i++) {
    status = gettime_appclk(appclk_desc, &appclk_time);
    getrawtime(cd.desc, &cv);
    readtime(cd.desc, &now);
    readtime(cd2.desc, &now2);
    ticks2nsec(&now, &now, cv.cpu_khz);
    ticks2nsec(&now2, &now2, cv.cpu_khz);
    fprintf(file_id, "tsc:\t%Lu\tbeta:\t%Lu\talpha:\t%Lu\tc_out:\t%Lu\n",
            cv.tsc, appclk_time, now, now2);
    usleep(waitconst);
}
```

```
    beta = 16777215;
    status = setrate(cd2.desc, &beta);
    for(i=0; i<15; i++) {
        status = gettime_appclk(appclk_desc, &appclk_time);
        getrawtime(cd.desc, &cv);
        readtime(cd.desc, &now);
        readtime(cd2.desc, &now2);
        ticks2nsec(&now, &now, cv.cpu_khz);
        ticks2nsec(&now2, &now2, cv.cpu_khz);
        fprintf(file_id, "tsc:\t%Lu\tbeta:\t%Lu\talpha:\t%Lu\tc_out:\t%Lu\n",
                cv.tsc, appclk_time, now, now2);
        usleep(waitconst);
    }

    beta = 16777215;
    status = setrate(cd2.desc, &beta);
    for (i=0; i<timeconst; i++) {
        status = gettime_appclk(appclk_desc, &appclk_time);
        getrawtime(cd.desc, &cv);
        readtime(cd.desc, &now);
        readtime(cd2.desc, &now2);
        ticks2nsec(&now, &now, cv.cpu_khz);
        ticks2nsec(&now2, &now2, cv.cpu_khz);
        fprintf(file_id, "tsc:\t%Lu\tbeta:\t%Lu\talpha:\t%Lu\tc_out:\t%Lu\n",
                cv.tsc, appclk_time, now, now2);
        usleep(waitconst);
    }
    close(file_id);
    return 0;
}
```

A.2.4 wrapperfunctions.s

```
; taken from Linux Assembly-Howto (2001)
; modified by Andreas Moser, 2002, GPL

section .text

; int initclk(unsigned long ip, struct clk_desc *rd)
global initclk
initclk:
    push    ebp         ; save (frame) basepointer
    mov     ebp,esp     ; copy stack pointer, pointing at end of stack
    add     ebp,4       ; after this 'add', ebp[i] means:
                        ; ebp[0] = return address
                        ; ebp[4] = arg1 (32bit), unsigned long
                        ; ebp[8] = arg1 (32bit), struct clk_desc*
    push    ebx         ; we catch a segfault unless we do this
    mov     eax,230     ; sys_call_table[230]
    mov     ebx,[ebp+4] ; put first argument to ebx
    mov     ecx,[ebp+8] ; put second argument to ecx
    int     0x80        ; exec syscall (return value is in eax)
    pop     ebx
    pop     ebp         ; restore stack, so that last entry is return address
    ret


; int setclk(int clk_desc, unsigned long long *new_time, int *new_beta)
global setclk
setclk:
    push   ebp
    mov    ebp,esp
    add    ebp,4
    push   ebx
    mov    eax,231
    mov    ebx,[ebp+4]
    mov    ecx,[ebp+8]
    mov    edx,[ebp+12]
    int    0x80
    pop    ebx
    pop    ebp
    ret


; int settime(int clk_desc, unsigned long long *new_time)
global settime
settime:
    push   ebp
    mov    ebp,esp
    add    ebp,4
    push   ebx
```

```
    mov    eax,232
    mov    ebx,[ebp+4]
    mov    ecx,[ebp+8]
    int    0x80
    pop    ebx
    pop    ebp
    ret


; int setrate(int clk_desc, int *new_beta)
global setrate
setrate:
    push   ebp
    mov    ebp,esp
    add    ebp,4
    push   ebx
    mov    eax,233
    mov    ebx,[ebp+4]
    mov    ecx,[ebp+8]
    int    0x80
    pop    ebx
    pop    ebp
    ret


; int getrawtime(int clk_desc, struct clkval *cv)
global getrawtime
getrawtime:
    push   ebp
    mov    ebp,esp
    add    ebp,4
    push   ebx
    mov    eax,234
    mov    ebx,[ebp+4]
    mov    ecx,[ebp+8]
    int    0x80
    pop    ebx
    pop    ebp
    ret


; int readtime(int clk_desc, unsigned long long *now)
global readtime
readtime:
    push   ebp
    mov    ebp,esp
    add    ebp,4
    push   ebx
    mov    eax,235
```

```
    mov    ebx,[ebp+4]
    mov    ecx,[ebp+8]
    int    0x80
    pop    ebx
    pop    ebp
    ret


; int nsec2ticks(unsigned long long *nsec, unsigned long long *ticks,
;                unsigned int cpu_khz);
global nsec2ticks
nsec2ticks:
    push   ebp
    mov    ebp,esp
    add    ebp,4
    push   ebx
    mov    eax,238
    mov    ebx,[ebp+4]
    mov    ecx,[ebp+8]
    mov    edx,[ebp+12]
    int    0x80
    pop    ebx
    pop    ebp
    ret


; int ticks2nsec(unsigned long long *ticks, unsigned long long *nsec,
;              unsigned int cpu_khz);
global ticks2nsec
ticks2nsec:
    push   ebp
    mov    ebp,esp
    add    ebp,4
    push   ebx
    mov    eax,239
    mov    ebx,[ebp+4]
    mov    ecx,[ebp+8]
    mov    edx,[ebp+12]
    int    0x80
    pop    ebx
    pop    ebp
    ret


; int calctime(unsigned long long *alpha, unsigned long long *tsc,
;              int *beta, unsigned long long *now)
global calctime
calctime:
    push   ebp
```

```
    mov    ebp,esp
    add    ebp,4
    push   ebx
    mov    eax,240
    mov    ebx,[ebp+4]
    mov    ecx,[ebp+8]
    mov    edx,[ebp+12]
    mov    esi,[ebp+16]
    int    0x80
    pop    ebx
    pop    ebp
    ret


; int init_appclk(int *appclk_desc, int max_slope,
;                 int number_of_parents, int *parents)
global init_appclk
init_appclk:
    push   ebp
    mov    ebp,esp
    add    ebp,4
    push   ebx
    mov    eax,236
    mov    ebx,[ebp+4]
    mov    ecx,[ebp+8]
    mov    edx,[ebp+12]
    mov    esi,[ebp+16]
    int    0x80
    pop    ebx
    pop    ebp
    ret


; int gettime_appclk(int appclk_desc, unsigned long long *nsec)
global gettime_appclk
gettime_appclk:
    push   ebp
    mov    ebp,esp
    add    ebp,4
    push   ebx
    mov    eax,237
    mov    ebx,[ebp+4]
    mov    ecx,[ebp+8]
    int    0x80
    pop    ebx
    pop    ebp
    ret
```

Index