Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Boris Zweimueller

Uhrensynchronisation in einem Ad-Hoc-Netz von BTnodes

Studienarbeit SA-2004-06
Wintersemester 2003/2004

Betreuer: Lennart Meier,
Philipp Blum

Verantwortlicher:
Prof. Dr. Lothar Thiele

27.2.2004

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Uhrensynchronisation in einem Ad-hoc-Netz von BTnodes

Boris Zweimüller
zboris@student.ethz.ch

4. März 2004

Studienarbeit SA-2004-06
Wintersemester 2003/2004
Institut TIK, Prof. L. Thiele
Betreuer: Lennart Meier, Philipp Blum

Zusammenfassung

Diese Arbeit beschreibt die Implementierung eines Algorithmus zur Uhrensynchronisation in einem Ad-hoc-Netz. In einem Ad-hoc-Netz basiert die Kommunikation nicht auf einer zugrundeliegenden Infrastruktur. Es erlaubt den teilnehmenden Knoten quasi aus dem Stegreif mit anderen Knoten Verbindungen aufzubauen und zu kommunizieren. In dieser Arbeit wurden BTnodes als Knoten verwendet. Das sind kleine, programmierbare Netzknoten, die im Wesentlichen aus einem Mikrokontroller und einem Radiomodul bestehen. Die verwendeten Radiomodule ermöglichen Drahtlose Kommunikation gemäss dem Bluetooth-Standard [2]. Bei der Implementierung muss die Unsicherheit über die Verzögerung, die insbesondere bei der Kommunikation mittels Bluetooth durch die grosse Variabilität der Übertragungszeit auftritt, auf ein Minimum reduziert werden. Zusätzlich müssen architekturbedingte Engpässe, zum Beispiel bei der Berechnung und Speicherung sehr grosser Zahlen, berücksichtigt werden. Ein Lösungsvorschlag mit allen nötig gewordenen Erweiterungen und Änderungen am Betriebssystem der BTnodes wird hier vorgestellt und dessen Implementierung detailliert erklärt.

Inhaltsverzeichnis

1 Einführung

1.1 BTnodes

BTnodes wurden an der ETH Zürich am Institut für Technische Informatik und Kommunikationsnetze und am Institut für Verteilte Systeme entwickelt. Ein BTnode ist eine eigenständige Rechen- und Wireless-Kommunikationsplattform, welche auf einem Bluetooth-Funk-Modul und einem Mikrokontroller basiert. BTnodes dienen hauptsächlich als Experimentier- und Demonstrationswerkzeug für die Forschung im Bereich der mobilen Ad-hoc-Netze (MANETs) und der verteilten Sensornetze. In dieser Arbeit wurden BTnodes der Revision 2.2 eingesetzt.

1.2 Motivation

Die Aufgabe dieser Semesterarbeit bestand hauptsächlich darin, einen Algorithmus zur Synchronisation von BTnodes in einem Ad-hoc-Netz zu implementieren. Ad-hoc bedeutet, dass die Kommunikation zwischen den Knoten nicht auf einer zugrundeliegenden Netzwerkinfrastruktur basiert, sondern dass die Knoten aus dem Stegreif mit beliebigen anderen Knoten, die sich in ihrer Nähe befinden, Kontakt aufnehmen können. Die einzelnen Netzknoten sollen selbständig externe Zeitinformation wie z. B. GPS- oder NTP[1]-Signale aufnehmen und daraus mittels Nachrichtenaustausch eine möglichst genaue gemeinsame Zeit berechnen.

Als Beispiel stelle man sich die Messung der Geschwindigkeit eines Autos vor, welches an zwei Messvorrichtungen vorbeifährt: Die Messgeräte zeichnen auf, zu welcher Zeit das Auto an ihnen vorbeifährt, woraus dann durch Kenntnis des Abstandes die momentane Geschwindigkeit berechnet werden kann. Würden die beiden Messgeräte die Passier-Zeit nicht mit synchronisierten Uhren messen, wäre eine korrekte Berechnung der Geschwindigkeit des Autos nicht möglich.

Eine der Kernfragen ist die Synchronisationsgenauigkeit, die auf BTnodes - mit all ihren technischen Einschränkungen - erreicht werden kann. Während der Arbeit wurden verschiedene Ansätze untersucht, um einen möglichst hohen Grad an Synchronisation zu erreichen.

1.3 Übersicht

Die nächsten drei Abschnitte erläutern die Arbeit genauer. Abschnitt 2 beschreibt in einem ersten Teil den verwendeten Synchronisationsalgorithmus und geht auf dessen Besonderheiten ein. In einem zweiten Teil werden die Nachteile und Probleme besprochen, die unter anderem einen speziellen Algorithmus für die Kommunikation nötig machten, der ebenfalls genau erklärt

[1] NTP: network time protocol.

wird. Im Abschnitt 3 werden die Implementierungen der zuvor besprochenen Algorithmen detailliert beschrieben und einige technische Schwierigkeiten aufgezeigt, die sich durch die Verwendung von BTnodes ergaben. Abschnitt 4 beschreibt, wie die implementierten Algorithmen mit BTnodes getestet wurden, wie die verwendete Testumgebung aussah und welche Messergebnisse erreicht wurden. Im Anhang A ist der gesamte Quellcode der Arbeit abgedruckt, um dem Leser Implementationsdetails aufzuzeigen.

2 Algorithmen

2.1 Synchronisationsalgorithmus

In diesem Abschnitt werden die Funktionsweise des implementierten Synchronisationsalgorithmus theoretisch erklärt und dessen Vorteile gegenüber einem traditionellen Algorithmus erläutert.

2.1.1 Einführung

Wie in Abschnitt 1.2 beschrieben sind synchronisierte Uhren für die Koordination von verteilten Aktionen nötig. Ein Uhrensynchronisationsdienst für ein Ad-Hoc-Netz hat grundlegend andere Probleme zu bewältigen als vergleichbare Dienste für konventionelle, auf einer Infrastruktur basierende Netze. Die wichtigsten Punkte sind hier aufgeführt: *Robustheit:* In einem mobilen Ad-hoc-Netz existiert keine ständige, stabile Verbindung zwischen den Knoten. Das Netzwerk kann zeitweise unterbrochen oder aufgeteilt sein. *Energie-Effizienz:* Synchronisation kann nur mittels Kommunikation erreicht und aufrechterhalten werden. Weil in MANETs typischerweise drahtlos kommuniziert wird, ist diese Kommunikation bei Betrachtung des Energieverbrauchs sehr teuer (batteriebetriebene Geräte). Der Uhrensynchronisationsdienst sollte mit minimaler Kommunikation einen möglichst hohen Grad an Synchronisation aufrechterhalten können. *Ad-hoc-Anordnung:* Der Uhrensynchronisationsdienst kann sich nicht auf eine festgelegte Konfiguration oder eine fixe Infrastruktur verlassen. Deshalb können keine traditionellen Algorithmen wie z.B. NTP auf MANETs angewendet werden.

Der in dieser Arbeit implementierte Algorithmus stammt aus [3], er berücksichtigt alle oben aufgeführten Punkte und wird in [1] detailliert erklärt. Zusätzlich gibt [1] eine Analyse, die zeigt, dass der Algorithmus worst-case-optimal ist. An dieser Stelle wird nur die Funktionsweise des Algorithmus erklärt.

2.1.2 Garantierte Zeitgrenzen

Das Kernelement des Algorithmus ist, dass die einzelnen Knoten nicht einen geschätzten Zeitwert besitzen, sondern sogenannte garantierte Zeitgrenzen.

Jeder Knoten besitzt eine untere und eine obere Zeitgrenze, zwischen denen sich die Echtzeit garantiert befindet. Die Differenz zwischen der oberen und der unteren Zeitgrenze wird als *Unsicherheit* des Knotens über die Echtzeit bezeichnet. Die Verwendung von garantierten Zeitgrenzen bietet gegenüber der Verwendung eines geschätzten Zeitwerts einige Vorteile; zwei davon sind hier aufgeführt:

- Bei der Synchronisation zweier Knoten ist die Berechnung der besten Zeitgrenzen eindeutig und sehr einfach: Beide am Synchronisationsvorgang beteiligten Knoten nehmen die grössere untere und die kleinere obere Grenze als neue Zeitgrenzen.

- Die Grenzen sind *garantiert*, das heisst, dass sich die Echtzeit mit Sicherheit zwischen den Grenzen befindet. Für sicherheitskritische Anwendungen kann dies notwendig sein.

2.1.3 Szenario

Wir nehmen folgendes Szenario an: Ein System bestehe aus Knoten, welche eine lokale Uhr besitzen, sich in einem *local state* befinden und über Berechnungsfähigkeit verfügen. Vorgänge im System können in Form von zwei Events beschrieben werden: Source und Communication Events. *Source Events*: Jeder Knoten des Netzes kann von aussen (via GPS, NTP) Grenzen der jetzigen Echtzeit erhalten. *Communication Events*: Für die Synchronisation müssen die Knoten ihre aktuellen Zeitgrenzen austauschen. Dabei treffen sich zwei Knoten, tauschen ihre momentanen Zeitgrenzen aus und berechnen wie in Abschnitt 2.1.5 beschrieben neue Grenzen. Aufgrund der unbekannten Nachrichtenverzögerung D, können Zeitgrenzen nicht perfekt, d.h. ohne Fehler, übertragen werden. Für die unbekannte Verzögerung D müssen ebenfalls Grenzen verwendet werden. In dieser Arbeit wurde ein Algorithmus verwendet, der Grenzen für die Verzögerung dynamisch bestimmt (Abschnitt 2.2).

2.1.4 Uhrenmodell

Jeder Knoten besitzt eine interne Hardware Uhr. Der Wert dieser Uhr zur Zeit t wird als lokale Zeit $h(t)$ des Knotens betrachtet. Basierend auf der Geschwindigkeit $dh(t)/dt$ der Uhr definieren wir ihren Drift, d.h. die Abweichung ihrer Geschwindigkeit von der "korrekten" Geschwindigkeit, zur Zeit t als

$$\rho(t) = \frac{dh(t)}{dt} - 1. \tag{1}$$

Wir definieren die Eigenschaften der Uhr zum Zeitpunkt eines Events e als $h_e = h(t_e)$ (aktuelle lokale Zeit) und $\rho_e = \rho(t_e)$ (Drift). Die Qualität der Synchronisation hängt von den Annahmen ab, die über die Eigenschaften der

Uhr getroffen werden. Der hier beschriebene Algorithmus setzt Uhren voraus, deren maximaler Drift beschränkt ist. Eine Konstante $\rho^{\max}$ beschränkt den Drift einer beliebigen Uhr gemäss

$$-\rho^{\max} \leq \rho(t) \leq \rho^{\max}. \tag{2}$$

2.1.5 Ablauf des Algorithmus

Wie bereits erwähnt, besitzt jeder Knoten einen Zustand. Dieser besteht aus der lokalen Zeit h_{LS} und den zugehörigen Zeitgrenzen T^l_{LS}, T^u_{LS}. Bei jedem Event e berechnet der Algorithmus untere und obere Grenzen basierend auf der Information des aktuellen Zustandes (T^l_{LS}, T^u_{LS}, h_{LS}) und der momentanen lokalen Zeit h_e. Die aktuellen Grenzen berechnen sich somit wie folgt:

$$T^l_e = T^l_{LS} + \frac{h_e - h_{LS}}{1+\rho^{\max}} \qquad T^u_e = T^u_{LS} + \frac{h_e - h_{LS}}{1-\rho^{\max}}. \tag{3}$$

Diese Berechnung wird in der Prozedur `calculate_bounds(...)`, siehe Abschnitt 3.2, vorgenommen. Wenn sich zwei Knoten synchronisieren wollen, läuft der Algorithmus folgendermassen ab:

1. Zu Beginn besitzt jeder Knoten einen gültigen *local state* mit h_{LS}, T^l_{LS} und T^u_{LS}.

2. Bei Auftreten eines Source oder Communication Events, werden die lokalen Zeitgrenzen entsprechend (3) aktualisiert.

3. Sowohl bei einem Source als auch bei einem Communication Event erhält der Knoten neue Zeitgrenzen $\hat{T}^l$ und $\hat{T}^u$, entweder von aussen oder von einem anderen Knoten. Das Intervall $\hat{T}^u - \hat{T}^l$, das sich durch Subtraktion der Zeitgrenzen ergibt, wird nun mit dem Intervall der aktualisierten lokalen Zeitgrenzen geschnitten:

$$T^l_e := \max\{T^l_e, \hat{T}^l\} \qquad T^u_e := \min\{T^l_e, \hat{T}^l\}. \tag{4}$$

4. Zum Schluss wird der *local state* mit den neu berechneten Werten aktualisiert:

$$h_{LS} = h_e \qquad T^l_{LS} := T^l_e \qquad T^u_{LS} := T^u_e \tag{5}$$

Die Kommunikation, die für den Synchronisationsalgorithmus nötig ist, bringt einige Schwierigkeiten mit sich. Diese und deren Lösung werden im nächsten Abschnitt besprochen.

2.2 Algorithmus zur Kommunikation

2.2.1 Motivation

Wie in Abschnitt 2.1 beschrieben, tauschen die an der Synchronisation beteiligten Knoten ihre momentanen Zeitgrenzen mittels aus. Die grösste Schwierigkeit bei der Kommunikation ist die Unsicherheit über die Verzögerung d, d.h. die Zeit zwischen Absenden einer Nachricht und deren Ankunft beim Empfänger. Für eine genaue Synchronisation ist es notwendig, diese Zeit möglichst genau zu kennen, da sie beim Empfänger zu den erhaltenen Zeitgrenzen dazugezählt werden muss.

Der in Abschnitt 2.1 erläuterte Synchronisationsalgorithmus vernachlässigt die Übertragungszeit, die bei jedem Nachrichtenaustausch vorhanden ist. Wir wollen sie in unsere Berechnung einbeziehen, bzw. die Unsicherheit über sie minimieren, um eine möglichst gute Synchronisation zu erreichen. Da es - insbesondere bei grosser Variabilität der Verzögerung - unmöglich ist, diese Verzögerung genau zu bestimmen, wurde in dieser Arbeit ein spezieller Algorithmus verwendet, um die Unsicherheit über die Verzögerung zu minimieren. Dieser Algorithmus wird im nächsten Abschnitt erläutert.

2.2.2 Minimierung der Unsicherheit über die Verzögerung

Der hier beschriebene Algorithmus minimiert die Unsicherheit über die Verzögerung bei der Übertragung einer Nachricht zwischen den Knoten P und Q auf eine berechenbare Grenze. Abbildung 1 verdeutlicht den Kommunikationsablauf.

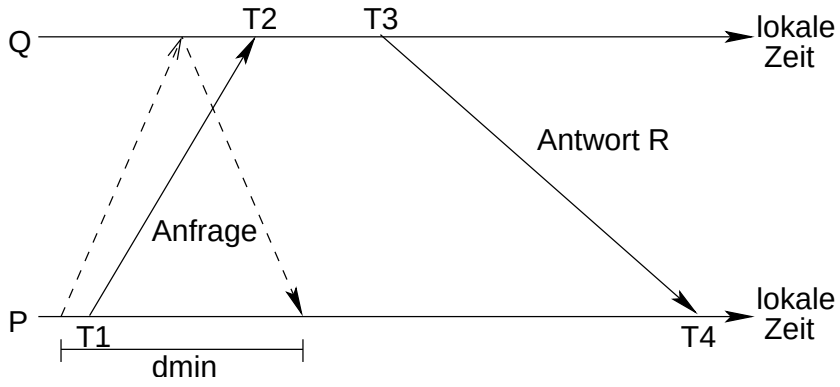

Abbildung 1: Kommunikationsablauf.

Wenn Q nur eine Nachricht von P erhält, hat Q keine Information über die Übertragungszeit dieser Nachricht. Durch den Austausch mehrerer Nachrichten ist es für P und Q jedoch möglich, die Grenzen für die Übertragungszeit zu erhalten. Zu diesem Zweck formulieren wir den Nachrichtenaustausch

als *Anfrage-Antwort*-Vorgang. Knoten P sendet zum Zeitpunkt $T1$ (Zeit der lokalen Uhr von P) eine Nachricht an Knoten Q, welche dieser zum Zeitpunkt $T2$ (Zeit der lokalen Uhr von Q) empfängt. Darauf berechnet Knoten Q die Antwort zur Anfrage von P und sendet diese zum Zeitpunkt $T3$ (Zeit der lokalen Uhr von Q) an P zurück. Die Nachricht trifft zum Zeitpunkt $T4$ (Zeit der lokalen Uhr von P) bei P ein. Mit den Zeiten $T1$ und $T4$ kann die Laufzeit der beiden Nachrichten von Knoten P zu Knoten Q und wieder zurück berechnet werden. Daraus lässt sich dann eine Schranke für die relevante Zeit $R = T4 - T3$ herleiten, obschon die beiden Zeiten ($T3$ und $T4$) von unterschiedlichen Uhren stammen. Die totale Dauer eines *Anfrage-Antwort*-Vorgangs zwischen P und Q beträgt:

$$t_{total} = T4 - T1. \tag{6}$$

Durch Subtraktion der Zeit, die für die Berechnung der Antwort verstreicht, erhält man die Übertragungszeit der beiden Nachrichten:

$$U = (T4 - T1) - (T3 - T2). \tag{7}$$

Der Nachrichtenalgorithmus ist nun so definiert, dass Knoten P nur diejenigen *Antwort-Nachrichten* von Q zur weiteren Berechnung verwendet, die innerhalb einer bestimmten Zeit nach der Anfrage bei P ankommen. Die Laufzeit dieser Nachrichten ist nicht grösser als ein bestimmtes $d_{\max}$:

$$U = (T4 - T1) - (T3 - T2) \leq d_{\max}. \tag{8}$$

Für *Antwort-Nachrichten* mit $U > d_{\max}$ wird die Anfrage von Knoten P wiederholt.
Damit ist es jetzt möglich, die Unsicherheit über die Übertragungszeit eines *Anfrage-Antwort*-Vorgangs zu reduzieren, doch die Unsicherheit über die Aufteilung der Übertragungszeit auf Hin- und Rückweg bleibt. Insbesondere bei der Anwendung des Synchronisationsalgorithmus aus Abschnitt 2.1 ist diese Aufteilung wichtig, da dort die Dauer des Rückwegs zur Berechnung von Zeitgrenzen verwendet wird. Diese Zeitgrenzen sind *garantiert* (siehe Abschnitt 2.1) und müssen natürlich auch nach einer Synchronisation *garantiert* bleiben. Wir definieren $d_{\min}$ (in Abbildung 1 gestrichelt eingezeichnet) als die minimale Dauer eines *Anfrage-Antwort*-Vorgangs (von P zu Q und wieder zuruck zu P). Ausserdem machen wir die Annahme, dass im Falle einer Laufzeit von $d_{\min}$ der Hin- und Rückweg gleich lange dauern ($d_{\min}/2$). Wenn wir R_{real} als effektive Dauer für den Rückweg festsetzen, gilt:

$$\frac{d_{\min}}{2} \leq R_{real} \leq U - \frac{d_{\min}}{2}. \tag{9}$$

Weil $\frac{d_{\min}}{2}$ der halben minimalen Übertragungszeit, und damit der minimalen Dauer des Rückwegs entspricht, definieren wir diesen kürzesten Rückweg als:

$$R_{\min} = \frac{d_{\min}}{2}. \tag{10}$$

Unter der Annahme, dass der Hinweg optimal, d.h. $d_{\min}/2$ war, lässt sich der maximale Fehler für den Rückweg ausdrücken als:

$$\alpha = \frac{R_{real}}{R_{\min}}. \tag{11}$$

Daraus ergibt sich mit (9) und (10) als Abschätzung für den maximalen Fehler:

$$\alpha = \frac{R_{real}}{R_{\min}} \leq \frac{U - \frac{d_{\min}}{2}}{\frac{d_{\min}}{2}}. \tag{12}$$

Durch weiteres Vereinfachen und Einsetzen von (8) erhält man für den Fehlerfaktor α:

$$\alpha = \frac{R_{real}}{R_{\min}} \leq \frac{U - \frac{d_{\min}}{2}}{\frac{d_{\min}}{2}} \leq \frac{2d_{\max}}{d_{\min}} - 1. \tag{13}$$

Damit die Zeitgrenzen auch nach einem Synchronisationsvorgang noch garantiert sind, ist folgende Aufteilung der gemessenen Übertragungszeit nötig: Beim Empfängerknoten wird zur unteren empfangenen Zeitgrenze $d_{\min}$ dazuaddiert. Zur oberen Zeitgrenze wird $U - d_{\min}$ hinzuaddiert. Dadurch bleiben die Grenzen garantiert und die Unsicherheit über die Verzögerung wurde wie oben beschrieben auf einen Faktor

$$\alpha \leq \frac{2d_{\max}}{d_{\min}} - 1. \tag{14}$$

minimiert.

3 Technische Realisierung

3.1 Events

Im Folgenden soll ein kurzer Überblick über die Eventsteuerung der BTnode Systemsoftware gegeben werden. Eine detaillierte Beschreibung entnehme man [4].

Die Systemsoftware eines BTnode ist aus Energiespargründen eventgesteuert und könnte als eine grosse ASM[2] dargestellt werden. Das heisst, dass eine Software-Komponente, wie z.B. ein Gerätetreiber, einen Event generieren kann, um andere Komponenten von einer Zustandsänderung in Kenntnis zu setzen und weitere Aktionen zur Eventbehandlung auszulösen. Nehmen wir an, ein Byte würde am seriellen Port ankommen. Der UART Treiber

[2]Abstrakte Zustands Maschine (abstract state machine).

müsste dann eine daran interessierte Software-Komponente informieren, so dass diese die angekommenen Daten lesen und verarbeiten kann.

Die zentrale Steuereinheit, die generierte Events verarbeitet, ist der Dispatcher. Dieser implementiert kooperatives Multitasking. Alle generierten Events werden vom Dispatcher in einer Warteschlange nach dem FIFO-Prinzip[3] verwaltet und gelangen der Reihe nach zur Ausführung. Nur ein Event ist jeweils aktiv und wird komplett abgearbeitet, bevor die Kontrolle an den nächsten Event übergeben wird. Dies erklärt, weshalb ein Event je nach Systemlast (Anzahl der Events in der Warteschlange) unterschiedlich schnell zur Ausführung gelangt.

Der Entwickler kann für bekannte Events eigene Behandlungsroutinen (sog. Callback Routinen) programmieren, die er beim Dispatcher registrieren muss. Der Dispatcher wird dann bei der Abarbeitung seiner Event-Warteschlange die Kontrolle der passenden Behandlungsroutine (z.B. der Software-Komponente, die das empfangene Byte verarbeitet) übergeben.

Besondere Beachtung erfordern die *timeout-events*. Diese werden nicht direkt in die Event-Warteschlange eingefügt, sondern in einer separaten Tabelle verwaltet. Bei jedem Timer-Interrupt[4] wird dann geprüft, ob einer der *timeout-events* zur Ausführung gelangen soll. Allenfalls wird er in die Event-Warteschlange eingefügt. Ein *timeout-event* wird nach dessen Registrierung beim Dispatcher nur einmal ausgeführt. Deshalb müssen weitere *timeout-events* neu beim Dispatcher registriert werden.

3.2 Synchronisationsalgorithmus

Für die Implementierung des Synchronisationsalgorithmus wurden folgende Variablen, Eventhandler und Prozeduren verwendet:

1. **Variablen**

   - `bt_addr_t current_remote_addr`

     In der Variablen `current_remote_addr` wird die Adresse des momentanen Synchronisationspartners gespeichert. Dies ist nötig, um bei einem ankommenden *connection-event* festzustellen, ob es sich um eine Verbindungsanfrage oder um eine Verbindungsaufbau-Bestätigung handelt. Eine genauere Erläuterung folgt in der Erklärung des Eventhandlers `conn_cb()`. Die Adresse des momentanen Partners wird ebenfalls von der Prozedur `send()` (siehe Abschnitt 3.3) verwendet, um Nachrichten an ihn zu senden.

   - `bt_addr_t last_remote`

     In der Variablen `last_remote` wird die Adresse des Knotens gespeichert, mit welchem zuletzt ein Zeitgrenzen-Austausch statt-

[3] First in first out.

[4] Bei jedem Überlauf des Hardware-Zeitzählers wird ein Timer-Interrupt generiert.

gefunden hat. Dadurch wird verhindert, dass in zwei aufeinanderfolgenden Synchronisationsdurchläufen der gleiche Knoten als Partner gewählt wird (vorausgesetzt, es gibt mehr als einen Knoten in der Umgebung).

- `bigint lowerbound`
  Speichert die untere Grenze zum Zeitpunkt der letzten Aktualisierung der Zeitgrenzen (siehe Abschnitt 2.1). Der abstrakte Datentyp `bigint` dient zur Speicherung grosser Ganzzahlen und wird in Abschnitt 3.4 beschrieben.
- `bigint upperbound`
  Speichert die obere Grenze zum Zeitpunkt der letzten Aktualisierung der Zeitgrenzen.
- `bigint timeboundsset`
  Der Zeitpunkt der letzten Aktualisierung der Zeitgrenzen gemessen in der lokalen Zeit des Knotens.
- `u32 syncfactor`
  Wie bereits erwähnt ist die grösste Zahl, die auf einem BTnode dargestellt werden kann, eine vorzeichenbehaftete 16-Bit-Zahl, d.h. [-32'768, 32'768]. Ein Synchronisationsablauf wird mittels eines *timeout-events* (siehe Erklärung des Eventhandlers `timeout_cb()`) ausgelöst, welcher zuvor beim Dispatcher (siehe 3.1) registriert werden muss. Aufgrund der grössten darstellbaren Zahl wäre die maximale Dauer zwischen zwei Synchronisationsabläufen ca. 33 Sekunden. Da wir aber an viel längeren Intervallen interessiert sind, kann mit dem Faktor `syncfactor` die effektive Wartezeit zwischen zwei Synchronisationsvorgängen auf $32'768 *$ `sync- factor` ausgedehnt werden. Details zur Implementierung sind in der Beschreibung des Eventhandlers `time out_cb()` aufgeführt. Diese Variable kann über eine Terminaleingabe verandert werden, damit nicht bei jeder Änderung eine Neucompilierung mit anschliessendem Aufladen auf den BTnode nötig ist.

2. **Eventhandler**

- `void timeout_cb(call_data_t call_data,`
  `cb_data_t cb_data)`
  timeout-events des Typs 0 sind für den Start des Synchronisationsvorgangs zuständig. Nach der ersten Registrierung in der `main(...)` Prozedur sind dannach die oben erwähnte Variable `syncfactor` und die lokale, statische Variable `sync` für die Wiederregistrierung und die Häufigkeit des Abarbeitens zuständig. Bei der Abarbeitung der Eventbearbeitungsroutine wird zuerst die Variable `sync` um eins inkrementiert und dann mit

```
if (sync % syncfactor == 0) {
// do synchronisation
...
}
```

überprüft, ob die mit dem Faktor `syncfactor` festgesetzte Zeit bereis verstrichen ist. Durch verschiedene Initialwerte von `sync` auf den einzelnen BTnodes wird erreicht, dass nicht alle beteiligten Knoten zur gleichen Zeit einen Synchronisationspartner suchen (siehe auch Abschnitt 4).

Falls die festgesetzte Zeit $30'000 * $`syncfactor` noch nicht abgelaufen ist, wird der *timeout-event* wieder registriert und die Kontrolle dem Dispatcher übergeben, der den nächsten Event zur Ausführung auswählt. Andernfalls wird mit dem Synchronisationsvorgang begonnen. Dazu führt der Knoten ein *inquiry*[5] aus, um potentielle Synchronisationspartner zu finden. Aus den gefundenen Geräten wird zufällig eines als Partner ausgewählt, wobei jedoch sichergestellt wird, dass nicht zweimal hintereinander derselbe Knoten gewählt wird. Dafür wird die Adresse des jeweiligen Partners in `last_remote` gespeichert. Zum Schluss wird eine L2CAP[6] zum Synchronisationspartner aufgebaut und der `timeout-event` wieder registriert.

- `void conn_cb(call_data_t call_data, cb_data_t cb_data)`

  Events des Typs *connection-event* werden von der Systemsoftware in zwei Fällen generiert: Falls ein anderes Gerät eine Verbindung aufbauen will oder wenn eine zuvor initiierte Verbindung erfolgreich zustande gekommen ist. Um diese beiden Typen zu unterscheiden, wurde die Variable `current_remote_addr` eingeführt. Wenn die Adresse des Synchronisationspartners mit derjenigen in `current_remote_addr` übereinstimmt, ist der Knoten der Initiator einer Verbindung, welche erfolgreich aufgebaut wurde. In diesem Fall wird mit dem Nachrichtenaustausch begonnen und eine Initialnachricht gesendet (siehe Abschnitt 3.3). Andernfalls ist der Knoten ein Partner und muss nicht auf den Event reagieren; er wird in Kürze eine Synchronisationsnachricht empfangen (siehe ebenfalls Abschnitt 3.3).

3. **Prozeduren**

   - `void calculate_bounds()`

[5] inquiry ist der Vorgang des Suchens anderer Bluetooth Geräte in der Umgebung.

[6] L2CAP: Logical Link Control and Adaptation Layer Protocol. Eine verbindungsorientierte, reihenfolgeerhaltende Bluetooth-Verbindung, vergleichbar mit einer TCP/IP Verbindung.

Wie in Abschnitt 2.1.5 in (3) beschrieben, werden die Zeitgrenzen in dieser Prozedur aktualisiert. Während die untere Zeitgrenze in jedem Fall aktualisiert wird, geschieht dies mit der oberen Grenze nur, wenn sie nicht einen Wert von unendlich hat.

3.3 Kommunikationsalgorithmus

Die Implementierung des vorgestellten Kommunikationsalgorithmus zur Minimierung der Unsicherheit über die Verzögerung wird in den folgenden Abschnitten detailliert erläutert.

3.3.1 Nachrichtenprotokoll

Zur Implementierung des in Abschnitt 2.2 beschriebenen Algorithmus wurde ein einfaches Nachrichtenprotokoll definiert. Wie in Abbildung 2 dargestellt, werden für das Protokoll drei verschiedene Nachrichtentypen verwendet, die im Folgenden erklärt werden. Während der Erläuterung der drei Typen wird davon ausgegangen, dass Knoten P (aus Abbildung 2) mit der Synchronisation beginnt. Die Nachrichtentypen sind wie folgt definiert:

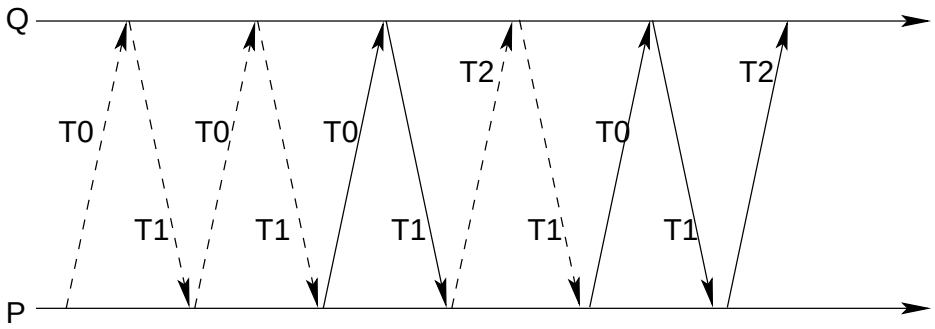

Abbildung 2: Nachrichtenprotokoll: Im Idealfall werden je eine T0-, T1-, und T2-Nachricht geschickt. Die gestrichelten Pfeile deuten Nachrichtenwiederholungen an, die aufgrund einer zu langen Übertragungszeit nötig wurden.

- **Typ T0:**
  Nachrichtentyp T0 startet das ganze Protokoll und wird von demjenigen Knoten gesendet, der sich mit einem anderen Knoten synchronisieren will, in Abbildung 2 ist dies Knoten P. Die Nachricht ist eine Aufforderung (Anfrage) an Q, seine aktuellen Zeitgrenzen an den anfragenden Knoten P zu senden. Q sendet daraufhin eine Nachricht vom Typ T1 zurück. Es ist zu beachten, dass in dieser ersten Initialnachricht T0 keine Zeitgrenzen von P zu Q übertragen werden, da Q nichts über die Verzögerung der ersten Nachricht herausfinden kann (siehe Abschnitt 2.2) !

- **Typ T1:**
  Eine Nachricht vom Typ T1 enthält einerseits die Zeitgrenzen (Antwort) von Q und ist gleichzeitig eine Aufforderung (Anfrage) an P, seine Zeitgrenzen mitzuteilen. P kann an dieser Stelle des Nachrichtenaustausches prüfen, ob die Nachricht T1 innerhalb der festgelegten Zeit $U \leq d_{\max}$ (siehe Abschnitt 2.2) seit dem Absenden der T0-Nachricht gekommen ist. Falls $U \leq d_{\max}$, war die Verzögerung klein genug. P wird nun mit den erhaltenen Zeitgrenzen gemäss dem Synchronisationsalgorithmus aus Abschnitt 2.1 neue Zeitgrenzen berechnen und eine Nachricht vom Typ T2 an Q retournieren. Falls $U > d_{max}$, wird P die Nachricht verwerfen und eine neue Anfrage an Q stellen und somit das Protokoll erneut mit einer Nachricht vom Typ T0 beginnen. Dieser Vorgang kann solange wiederholt werden bis $U \leq d_{\max}$ ist. Falls die Nachricht eine Wiederholung war, die aufgrund einer T2 Nachricht gesendet wurde, welche eine Laufzeit von $U > d_{\max}$ hatte, werden natürlich die Zeitgrenzen des Empfängers ebenfalls neu berechnet.

- **Typ T2:**
  Eine Nachricht vom Typ T2 enthält nur noch Zeitgrenzen (in Abbildung 2 von P), die auf Nachrichten des Typs T1 hin geschickt werden. Q kann wiederum wie beim Nachrichtentyp T1 ausrechnen, ob die Nachricht innerhalb der festgesetzten Zeit zurückkam. Dann berechnet Q entweder seine neuen Zeitgrenzen und beendet die Kommunikation, oder er wiederholt die Anfrage mit einer Nachricht vom Typ T1. Dabei kann es vorkommen, dass diese erneute T1 Nachricht ebenfalls eine Übertragungszeit $U > d_{max}$ benötigt, wodurch das Protokoll dann wieder mit einer T0-Nachricht gestartet wird, wie in Abbildung 2 abgebildet.

3.3.2 Implementierung

Für die Implementierung des Kommunikationsalgorithmus wurden folgende Variablen und Prozeduren verwendet:

1. **Variablen**

   - `TIME_MESSAGE_LENGTH`
     Definiert die Länge einer Nachricht, welche für das Versenden der Zeitgrenzen verwendet wird. Das Format einer Nachricht ist in Abbildung 3 dargestellt. Neben den beiden Feldern für den Typ und die Verarbeitungszeit mit fixer Grösse werden natürlich zwei Felder für die Zeitgrenzen benötigt. Da diese unterschiedliche Grösse haben können (siehe Abschnitt 3.4) kann auch die Länge einer Zeit-Nachricht variabel sein.

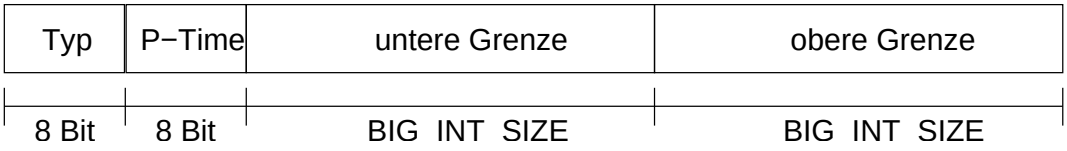

Abbildung 3: Format der Nachricht für den Zeitgrenzenaustausch.

- `MAX_SEND_REPETITION`
  Definiert, wie oft während des Ablaufs des Nachrichtenprotokolls Nachrichten wiederholt werden, wenn eine Antwort nicht innerhalb der festgesetzten Zeit (d_{max}) eintreffen sollte.
- `u8 sent`
  Speichert den Zeitpunkt, in dem der Knoten eine Nachricht absendet. Der Datentyp `u8` wird in der BTnode Systemsoftware definiert und entspricht einem `unsigned char` in `C`.
- `u8 received`
  Speichert den Zeitpunkt, in dem der Knoten eine Nachricht empfängt.
- `u8 dborder`
  Die Zeitschwelle, innerhalb welcher eine auf eine *Anfrage-Nachricht* zurückkommende *Antwort-Nachricht* noch *akzeptiert* wird. Dieser Wert entspricht dem d_{max} aus Abschnitt 2.2.
- `u8 dmin`
  Die minimale Laufzeit, die eine Nachricht von einem Knoten zu einem anderen Knoten benötigt. Dieser Wert entspricht dem $\frac{d_{min}}{2}$ aus Abschnitt 2.2.
- `u8 message_counter`
  Zählt die Anzahl Wiederholungen von Nachrichten. Falls

  ```
  (message_counter == MAX_SEND_REPETITION)
  ```

  gilt, wird der Wiederholvorgang abgebrochen.

2. **Prozeduren**

- `void data_received_cb(call_data_t call_data, cb_data_t cb_data)`
  Die Funktion `data_received_cb` behandelt *data received events* und wird immer dann ausgeführt, wenn eine neue Nachricht ankommt (siehe Abschnitt 3.1). Diese Funktion implementiert das oben beschriebene Kommunikationsprotokoll. Zuerst wird mit

  ```
  received = (u8)inp(TCNT0);
  ```

die Empfangszeit des Knotens gespeichert. Falls noch nicht zu viele Wiederholungen aufgetreten sind, wird die angekommene Nachricht auf ihren Typ geprüft und die entsprechenden Aktionen eingeleitet. Falls der Nachrichtenaustausch beendet ist (nach einer Nachricht vom Typ T2 oder falls zu viele Wiederholungen aufgetreten sind) wird die Verbindung zum entfernten Knoten abgebaut, der Variablen `current_remote_addr` der Wert 0 zugewiesen und der `message_counter` ebenfalls auf den Wert 0 gesetzt, damit später ein neuer Synchronisationsvorgang gestartet werden kann.

Weil d_{min} vom Benutzer bestimmt wird, kann es sein, dass dafür ein zu grosser Wert gewählt wurde. Mit der Abfrage

```
if ( (((received - sent) - ptime) - dmin) < dmin) {
        dmin = (received - sent) - ptime - dmin;
}
```

wird festgestellt, ob die Zeit, die zur oberen Grenze hinzugezählt wird, kleiner ist als d_{min}. Dies ist aber nur dann möglich, wenn d_{min} zu gross gewählt war. In diesem Fall wird d_{min} aktualisiert.

- `void sendmessage(u16 channel_pos, u16 local_cid, u16 messagetype)`

  Die Funktion `sendmessage` sendet eine Nachricht des Typs `messagetype` an den momentanen Synchronisationspartner. Die aktuellen Zeitgrenzen werden berechnet und zusammen mit dem Nachrichtentyp in der Zeichenkette `message` (siehe Abbildung 3) gespeichert. Danach wird mit:

  ```
  ptime = (u8)inp(TCNT0) - received;
  ```

  die Verarbeitungszeit berechnet und ebenfalls in `message` gespeichert. Gerade bevor die Nachricht mit `btn_bt_data_send(...)` gesendet wird, wird noch die aktuelle Sendezeit in `sent` gespeichert, damit der Knoten, wenn er auf seine Nachricht eine Antwort erhält, die Laufzeit berechnen kann (siehe Abschnitt 2.2).

3.3.3 Genauigkeit der lokalen Zeitmarken

Eine Schwierigkeit, die sich im Zusammenhang mit dem Kommunikationsalgorithmus ergab, war, die Stufe zu wählen, auf welcher die lokale Zeit von ankommenden und weggehenden Nachrichten gespeichert werden sollte. Zeitmarken auf Hardwareebene, d.h. Zeitmessung beim Absenden oder Empfangen eines Bytes durch das Bluetooth-Modul, ist die genaueste Methode. Sie wäre aber mit sehr grossen Änderungen in der Systemsoftware verbunden gewesen: Alle ankommenden und ausgehenden Daten müssten

darauf untersucht werden, ob sie der Beginn einer Zeitgrenzen-Nachricht sind, um dann die lokale Zeit zu speichern. Zusätzlich wäre die Implementierung einer Art Transportprotokoll nötig geworden, um Pakete korrekt zu segmentieren und wieder zusammenzusetzen und der richtigen Anwendung zuzuordnen. Weil der Bluetooth-Standard[2] bereits ein solches Transportprotokoll - L2CAP - bereitstellt, war es naheliegend, dieses zu verwenden. Damit ist auch die parallele Ausführung anderer Programme neben dem Synchronisationsalgorithmus einfach möglich, weil jede L2CAP-Verbindung eine eigene PSM[7] besitzt.

In der momentanen Implementierung werden die Ankunfts- und Absendezeiten jeweils beim Auftreten eines *data_received_events* respektive vor dem Aufruf der Funktion `send()` gespeichert. Bei ankommenden Nachrichten könnte man die Zeit auch vor dem Generieren des Events durch die Systemsoftware messen, um einer allfälligen Verzögerung bei der Eventabarbeitung (siehe Abschnitt 3.1) vorzubeugen. Die jetzige Software erzeugt jedoch keine zusätzlichen Events. Ebenso könnte man die Zeitmessung für ausgehende Nachrichten in der Systemsoftware vornehmen. Dies müsste allerdings in einer der oberen Schichten des Bluetooth-Stacks geschehen, bevor die Nachricht segmentiert wird, damit nur bei Nachrichten des Synchronisationsalgorithmus die Zeit gespeichert wird. Wenn sowohl in der Systemsoftware als auch in der eigentlichen Anwendung die Sendezeiten gemessen werden, lässt sich jedoch kein Zeitunterschied feststellen (mit einer Zeitauflösung in Millisekunden). Aus den oben genannten Gründen wurde bewusst auf Änderungen der Systemsoftware verzichtet und die Zeitmessung in der Anwendung ausgeführt, um die Implementierung einfacher zu halten.

3.4 Grosse Ganzzahlen

Für die Darstellung der Zeitgrenzen verwendet der Algorithmus eine Auflösung von Millisekunden. Da auf einem BTnode nur vorzeichenbehaftete 16-Bit-Zahlen[8] dargestellt werden können, musste eine Möglichkeit gefunden werden, sehr grosse Ganzzahlen auf einem BTnode effizient zu speichern und zu berechnen. Da nicht nur einfache Addition und Subtraktion, sondern auch Division, Multiplikation und verschiedene relationale Operatoren unterstützt werden mussten, war die Implementierung einer Bibliothek die flexibelste Lösung. Diese Bibliothek (siehe Anhang A.2) stellt alle nötigen Operationen für beliebig lange Zahlen zur Verfügung.
Eine Ganzzahl wird durch folgende Struktur repräsentiert:

```
typedef struct bi {
```

[7]PSM: Protocol Service Multiplexer Number: Die PSM identifiziert den jeweiligen L2CAP Kanal und damit die Anwendung, an die die Daten weitergeleitet werden, vergleichbar mit Portnummern bei TCP.

[8]Typ S16 auf dem BTnode.

```
    u8 data[BIG_INT_SIZE];
    u8 pos;
}bigint;
```

Die Konstante `BIG_INT_SIZE` kann im Headerfile `bigint.h` (siehe Anhang A) gesetzt werden und bestimmt die maximale Grösse der verwendeten Zahlen. Das `data`-Feld verwaltet die einzelnen Ziffern der Ganzzahl. Um Speicher zu sparen, werden die Zahlen zur Basis 128 dargestellt. Eine Ziffer kann daher Werte zwischen 0 und 127 annehmen. Die Variable `pos` verwaltet den von der Zahl benötigten Speicher, indem sie den Feld-Index (von `data`) speichert, bis zu welchem Speicherplatz gebraucht wird. Dies ist vor allem bei Zahlen sinnvoll, die sehr gross werden können, weil dann bei Berechnungen jeweils nur der momentan von der Zahl benötigte Speicher und nicht das ganze `data`-Feld traversiert werden muss. Eine Basis von 256 wäre natürlich noch effizienter, doch besteht das Problem, dass auf einem BTnode nur vorzeichenbehaftete 16-Bit-Zahlen berechnet und dargestellt werden können. Das Ergebnis einer Multiplikation kann deshalb unter Umständen nicht mehr in einem Register gespeichert werden und hätte einen Überlauf zur Folge, wodurch es verloren ginge (z.B. $255 * 255$). Die Bibliothek benutzt standardmässige Langzahlarithmetik für die Rechenoperationen. Die Implementierung dieser Operationen kann den Quelldateien entnommen werden (siehe Anhang A). An dieser Stelle soll nur auf eine spezielle Prozedur näher eingegangen werden.

- `void driftdivision(bigint * big, bool direction)`

  Wie in Abschnitt 2.1 beschrieben muss der Drift der internen Uhr eines Knotens berücksichtigt werden. Für ein bestimmtes Zeitintervall muss daher die maximal mögliche Abweichung berechnet werden. Dies ergibt sich wie bereits erwähnt aus:

$$T_e = \frac{h_e - h_{LS}}{1 \pm \rho^{\max}} = \frac{1}{1 \pm \rho^{\max}}(h_e - h_{LS}) \tag{15}$$

  In der Ganzzahl-Bibliothek wurde eine Funktion `simpledivbigint()` implementiert, die nur Divisoren kleiner als die zur Darstellung der Zahl verwendeten Basis erlaubt (in unserem Fall ist der grösste Divisor 127). Eine allgemeine Divisionsfunktion wäre sehr rechenaufwändig für einen BTnode, weshalb die Funktion `driftdivision` implementiert wurde. Diese berechnet mit Hilfe von `simpledivbigint()` und `multiplybigint()` die Division in (15) effizienter. Weil mit den zur Verfügung gestellten Funktionen nur ganze Zahlen berechnet werden können, wird der Quotient $\frac{1}{1+\rho^{\max}}$ mit 10000 multipliziert und die Variable `mp` mit diesem Wert initialisiert.

  Der Parameter `big` entspricht $h_e - h_{LS}$ (in der aufrufenden Prozedur, meist `calculate_bounds`, berechnet). Die Variable `big` wird nun mit

`mp` multipliziert und das Ergebnis anschliessend wieder durch 10000 dividiert. Der Wert von `big` ist dann $\texttt{big} = \frac{\texttt{big}}{1-\rho^{\max}}$, d.h. er entspricht dem Zeitwert, der eine maximal langsame Uhr hätte. Mit dem Parameter `direction` wird angegeben, ob der Drift für eine langsamere (`direction = 0`) oder für eine schnellere (`direction = 1`) Uhr berechnet werden soll. Für den letzten Fall wird $\texttt{big}_{original} + (\texttt{big}_{original} - \texttt{big}_{langsam})$ berechnet, der Wert für eine schnellere Uhr.

3.5 Änderungen an der Systemsoftware

Die Implementierung des Synchronisationsalgorithmus auf BTnodes machte einige Änderungen und Erweiterungen an der bestehenden Systemsoftware nötig. In Anhang A.3 ist die ganze Echtzeit-Uhr-Implementierung mit den nötig gewordenen Änderungen und Erweiterungen angefügt. Weil jeder Knoten eine fortschreitende lokale Zeit benötigt, deren Darstellung mit den Standard Datentypen des BTnode aber nicht möglich ist (Zahlen grösser als 16-Bit), muss diese Zeit in einem `bigint` (siehe Abschnitt 3.4) gespeichert werden. Die neu eingeführten Variablen und Prozeduren sowie die abgeänderten Prozeduren sind nachfolgend detailliert erklärt.

1. **Variablen**

   - `bigtime`

     Die globale Variable `bigtime` speichert die lokale Zeit des BTnodes seit Systemstart in Millisekunden. Sie ist vergleichbar mit der globalen Variablen `mseconds`, welche aber nur vorzeichenbehaftete 16-Bit-Zahlen speichern kann.

   - `updatevalue`

     Die Variable `updatevalue` wird für das Aktualisieren der lokalen Zeit (`bigtime`) benötigt. Sie wird in der `btn_rtc_init()` Prozedur mit dem Wert 250 initialisiert. Der 8-Bit-Zähler `TCNT0`, der für die Messung der lokalen Zeit benutzt wird, läuft in der aktuellen Konfiguration (siehe `btn_rtc_init()`) alle 250 Millisekunden über. Bei jedem dieser Überläufe wird die Variable `bigtime` um 250 erhöht. Eine genaue Erklärung dieser Aktualisierung folgt in der Beschreibung der Prozeduren.

2. **Prozeduren**

   - `btn_rtc_init()`

     In dieser Prozedur werden zusätzlich die beiden Variablen `bigtime` und `updatevalue` initialisiert.

   - `void btn_rtc_get_64(bigint* big, u8* offset)`

Diese Prozedur retourniert die aktuelle Systemzeit. Die Variable `bigtime` wird dazu an die übergebene Adresse des Parameters `big` kopiert. Diese Kopie ist notwendig, damit der Aufrufer die Variable verändern kann. Da in `bigtime` die Zeit nur in 250 Millisekunden Schritten gezählt wird, muss der momentane Wert des Zählers `TCNT0` ebenfalls an den Aufrufer retourniert werden, damit dieser durch Addition die aktuelle lokale Zeit berechnen kann. Es wurde darauf verzichtet, diese Addition in der Prozedur `btn_rtc_get_64` auszuführen, um nicht zu viel Zeit "im System" zu verbringen.

- `SIGNAL(SIG_OVERFLOW0)`

  Bei jedem Überlauf des Zählers `TCNT0` wird ein Interrupt ausgelöst, der von dieser Routine behandelt wird. Wie oben erwähnt, wird zum aktuellen Zeitwert in `bigtime` der Wert 250, d.h. die Variable `updatevalue` addiert. Weil nicht zu viele Instruktionen während der Interruptbehandlung ausgeführt werden dürfen (siehe Abschnitt 4.2, Prozedur `SIG_INTERRUPT7`), wird die Variable `updatevalue` bereits vorgängig mit dem Aktualisierungswert (hier 250) initialisiert. Die Umwandlung von 250 in den Datentyp `bigint` entfällt somit in der Behandlungsroutine.

4 Ergebnisse

4.1 Testumgebung

Um das Funktionieren des Synchronisationsalgorithmus zu testen, muss man diesen auf einer Ansammlung von BTnodes ausführen und periodisch von allen BTnodes *gleichzeitig* die aktuellen Zeitgrenzen abfragen können. Ausserdem sollte es möglich sein, einzelnen BTnodes von aussen neue Zeitinformation einzuspeisen (vgl. Abschnitt 1.2: NTP, GPS). Dazu wurde eine Testumgebung entwickelt, die in Abbildung 4 dargestellt ist. Um von allen BTnodes gleichzeitig ihre momentane Zeit zu erhalten, wurden diese mit dem Parallelport eines PC's verbunden. Auf dem PC können nun periodisch Signale generiert werden, die bei den angeschlossenen BTnodes einen Interrupt auslösen. Zusätzlich gibt es eine serielle Verbindung vom PC zu einem BTnode, die für die Datenein- und ausgabe über einen Terminal benutzt werden kann.

In der jetzigen Version speichert jeder BTnode bei der Behandlung des Interrupts die aktuellen Zeitgrenzen lokal in einem Feld, welches jederzeit über den Terminal ausgegeben werden kann (siehe Abschnitt 4.2). Eine alternative Implementierung versucht die Zeitinformationen via Bluetooth an einen zentralen Knoten zu übermitteln und ist in Abschnitt 4.5 detaillierter beschrieben.

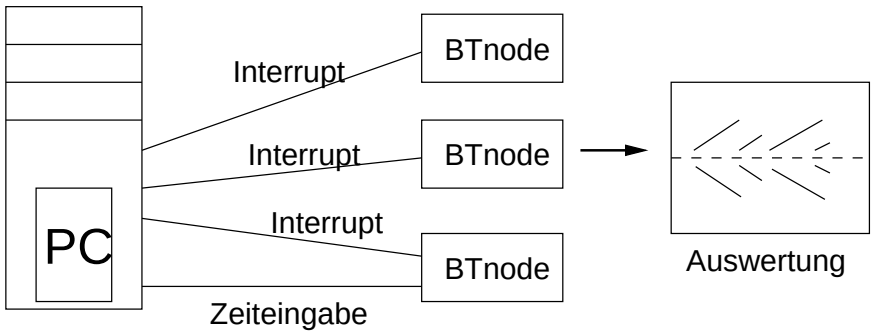

Abbildung 4: Die Testumgebung.

4.2 Implementierung auf BTnodes

4.2.1 Interruptbehandlungsroutine und Speichern der Werte

Das Speichern der aktuellen Zeitgrenzen während der Interruptbehandlung wird mit globalen Variablen realisiert. Folgende Variablen und Prozeduren sind für diese Aufgabe zuständig:

1. **Variablen**

   - `MAX_STORE_VAL`
     Diese Konstante definiert die maximale Anzahl zu speichernder Zeitgrenzenpaare im globalen Feld `timevalues`.
   - `bigint timevalues[2 * MAX_STORE_VAL]`
     Dieses globale Feld dient zur Speicherung der aktuellen unteren und oberen Zeitgrenzen.
   - `u32 pos`
     In der globalen Variablen `pos` ist der aktuelle Index in das Feld `timevalues` gespeichert.

2. **Prozeduren**

   - `SIGNAL (SIG_INTERRUPT7)`
     In der Prozedur `main` wird der BTnode mit

     ```
     // disable external interrupt 7
     cbi (EIMSK, INT7);
     // set external interrupt 7 to trigger
     // on a rising edge
     sbi (EICRB, ISC71);
     sbi (EICRB, ISC70);
     // enable external interrupt 7
     ```

```
sbi (EIMSK, INT7);

// initialize IO Port 7
// set PORTE7 as input
cbi(DDRE, 7);
cbi(PORTE, 7);
```

so konfiguriert, dass an Anschluss `E` auf Pin `7` (PE7) ein externer Interrupt ausgelöst werden kann (siehe Hardware-Referenzhandbuch des Atmel ATmega 128L oder [4]). Dazu werden der Pin 7 von Port E als Eingang initialisiert und die Interruptsteuerregister so gesetzt, dass eine steigende Flanke an Port E, Pin 7 einen Interrupt auslöst. Dieser Interrupt wird dann vom Signalhandler `SIGNAL (SIG_INTERRUPT7)` behandelt. In dieser Behandlungsroutine wird dann mit:

```
btn_disp_put_event(TIMEOUT_EV, 1);
```

ein Timeout-Event des Typs `1` registriert, der vom Dispatcher als Nächstes abgearbeitet wird. Es ist wichtig, nicht in der Interruptbehandlungsroutine selbst das Berechnen und Speichern der aktuellen Werte auszuführen, damit diese nicht zu viele Taktzyklen benötigt. Eine Interruptbehandlungsroutine darf nämlich nicht länger als 700 Zyklen dauern, da im schlechtesten Fall (Baudrate: 57600 baud) alle 700 Zyklen Daten am UART anliegen. Diese Daten müssen innerhalb von 700 Zyklen abgeholt werden, da der Hardwarepuffer nur ein Byte gross ist. Bei Nichtabholen der Daten würden diese von den folgenden überschrieben.

- `void timeout_cb(call_data_t call_data, cb_data_t cb_data)`
  Im Timeouthandler `timeout_cb` werden zwei verschiedene Timeout-Typen unterschieden. Typ 0 ist der eigentliche Synchronisationsvorgang (siehe Abschnitt 3.2) und Typ 1 ist für das Speichern der Zeitwerte im globalen Feld `timevalues` zuständig.

4.2.2 Anpassungen für die externe Zeiteingabe

Wie in Abschnitt 4.1 beschrieben, kann einem BTnode über eine serielle Verbindung Zeitinformation von aussen übermittelt werden. Dazu wird die standardmässige serielle Verbindung benutzt, die normalerweise für Ausgaben des BTnodes in einen Terminal und Eingaben vom Terminal zum BTnode verwendet wird. Weil in unserem Fall ein PC-Programm die Eingabe übernehmen soll, kann nicht der Terminal selbst für die Eingabe benutzt werden. Die serielle Schnittstelle muss direkt angesprochen werden. Angenommen, wir befinden uns auf einem Unix-System und der BTnode ist an die serielle Schnittstelle `/dev/ttyS0` angeschlossen. Dann kann mittels

```
echo -e -n 'teststring\r' > /dev/ttyS0
```

die Zeichenkette `teststring` an den BTnode übermittelt werden. Das `r` löst einen Wagenrücklauf aus. Vergleichbar mit dem in der BTnode Systemsoftware mitgelieferten Beispielprogram `bt_cmd.c` wurden nun Funktionen implementiert, die serielle Eingaben entgegennehmen, diese auswerten und die nötigen Aktionen (weitere Funktionsaufrufe) auslösen. Da Zeitgrenzen mit dem Datentyp `bigint` dargestellt werden, muss die Zeicheneingabe in dieses Zahlenformat umgewandelt werden. Diese Umwandlung kann aber nicht rechnerisch gemacht werden, weil ein BTnode nur Zahlen mit maximal 16-Bit verwalten kann. Deshalb werden auf dem BTnode Variablen mit fixen Werten initialisiert, die dann zur Zeiteingabe, resp. zum Setzen der Zeitgrenzen verwendet werden. Nachfolgend eine Beschreibung der für die Zeiteingabe benötigten Variablen, der möglichen Terminaleingaben und der Prozeduren, die die Eingaben behandeln:

1. **Variablen**

   - `bigint t100000ms`
     Die Variable `t100000ms` wird in der `main()`-Prozedur mit dem Wert $100'000$ initialisiert.
   - `bigint t200000ms`
     Die Variable `t200000ms` wird in der `main()`-Prozedur mit dem Wert $200'000$ initialisiert.
   - `bigint t5min[36]`
     Das Feld t5min wird mit Zeitwerten in fünf-Minuten-Abständen initialisiert. Der Wert von `t5min[0]` ist daher $300'000$, der von `t5min[1]` ist $600'000$ etc. So lassen sich nun die Zeitgrenzen durch Zuweisung des Wertes am jeweiligen Index alle fünf Minuten während drei Stunden neu setzen. Die Lösung mit einem Feld benötigt zwar etwas mehr Speicher, dafür ist aber die Zuweisung neuer Zeitgrenzen ohne Berechnung möglich und deshalb schnell. Für andere Zeiteingabeintervalle lässt sich dieses Feld problemlos anpassen.
   - `bigint t1h`
     Die Variable `t1h` wird in der `main()`-Prozedur mit dem Wert $3'600'000$ initialisiert.

2. **Prozeduren**

   - `void debug_input_cb( call_data_t call_data,`
     `cb_data_t cb_data )`
     Dieser Event wird vom System bei ankommenden Daten an der seriellen Verbindung generiert. Die Daten werden in einen Puffer

kopiert und anschliessend der Funktion `process_cmd(...)` zur Behandlung übergeben.

- `void process_cmd( u8* cmd_buf )`

  Die Funktion `process_cmd` analysiert die Eingabe und führt die entsprechende Aktion aus.

3. **Terminaleingaben**

   - `setbounds s16:s16`

     Mit der Eingabe `setbounds s16:s16` können die Zeitgrenzen auf einen Wert gesetzt werden, der mit dem Datentyp `s16`[9] dargestellt werden kann. Dabei muss natürlich auch die Variable `timeboundsset` (siehe Abschnitt 3.2) aktualisiert werden.

   - `shiftup5m s16`

     Mit der Terminaleingabe `shiftup5m s16` können beide Zeitgrenzen um den Wert $s16 * 5$ Minuten hinaufgesetzt werden. Der übergebene *Multiplikationsfaktor - 1* wird als Index in das oben definierte Feld `t5min` verwendet. Um auf dem BTnode z.B. die Zeitgrenzen 190 und 195 Sekunden zu setzen, wären folgende Terminaleingaben notwendig:

     ```
     setbounds 10000 15000
     shiftup5m 3
     ```

     Damit würden die Zeitgrenzen zuerst auf 10000 und 15000 Millisekunden gesetzt und anschliessend `t5min[3 - 1]` = 180'000 dazugezählt. Auch hier ist nach dem Setzen der Zeitgrenzen wieder eine Aktualisierung der Variable `timeboundsset` nötig.

     Die Terminaleingabe `shiftup5m s16` ist, zusammen mit dem Feld `t5min`, ebenfalls einfach abzuändern und vom Benutzer an das jeweilige Zeiteingabeintervall anzupassen (siehe Abschnitt 4.3).

   - `hourup`

     Diese Eingabe erhöht die momentanen Zeitgrenzen um eine Stunde.

4.3 PC Tool

Dieser Abschnitt beschreibt die für das Testen verwendete PC-Applikation und geht auf deren Implementierung ein. Die Applikation ist nur auf einem UNIX-System lauffähig und besteht aus zwei getrennten Funktionen, der `main()`-Funktion und der `sendInt()`-Funktion. Die `main()`-Funktion

[9] Vorzeichenbehaftete 16-Bit-Zahlen.

enthält im Wesentlichen eine `while`-Schleife, aus der periodisch die Funktion `sendInt()` aufgerufen wird, um auf den angeschlossenen BTnodes einen Interrupt auszulösen. Ebenfalls periodisch wird dem Anker-Knoten die Echtzeit mitgeteilt. Diese beiden Prozeduren werden nun genauer erklärt:

- **int main()** In der erwähnten `while`-Schleife der `main()`-Funktion wird jede Minuten (wählbar) die Funktion `sendInt()` aufgerufen. Die Zeitmessung geschieht in dieser einfachen Implementierung mittels des Systemaufrufs `sleep()`. Die Zeitmessung mit `sleep()` ist sicher nicht die genaueste. Sie war aber für das Testen der Funktionalität des Synchronisationsalgorithmus ausreichend. Bei einer Erweiterung der Testumgebung wäre allenfalls die Verwendung einer genaueren Uhr sinnvoll (z.B. direkt den Zeitzähler des Systems auslesen). Für die Zeiteingabe wird, wie in Abschnitt 4.2.2 bereits erklärt, die serielle Schnittstelle verwendet, indem mit `echo '...' > /dev/ttyS0` direkt darauf geschrieben wird. Um diesen Aufruf aus der Applikation zu benützen, wird die Funktion `system(...)` verwendet. Damit die Eingaben periodisch wiederholt werden, enthält die `while`-Schleife zwei Zähler `int count` und `int mult`. Der Zähler `count` zählt die verstrichene Zeit (in Minuten) und ist dafür zuständig, dass, falls `(count % X) == 0`, eine Zeiteingabe stattfindet. `X` steht dabei für die Häufigkeit der externen Zeiteingaben. Der Zähler `mult` ist die Verbindung zu dem in 4.2.2 erwähnten Feld `t5min`, in welchem in fünf-Minuten-Schritten Zeitwerte für die Zeiteingabe gespeichert sind. Der Wert des Zählers `mult` ist das Argument für die Terminaleingabe `shiftup5m S16` (Abschnitt 4.2.2). Er muss deshalb je nach Häufigkeit der Zeiteingaben unterschiedlich inkrementiert werden. Als Beispiel gehen wir davon aus, dass auf den BTnodes das Feld `t5min` wie beschrieben initialisiert ist (d.h. mit 36 Zeitwerten in fünf-Minuten-Schritten) und eine externe Zeiteingabe alle 15 Minuten erfolgen soll. Dann müsste der Code in der `while`-Schleife der `main()`-Funktion folgendermassen aussehen:

```
if (count % 15 == 0) {
    // initial setting
    system("echo -e -n 'setbounds 20000:20100\r' >
           /dev/ttyS0");
    // create call-string
    sprintf(call, "echo -e -n 'shiftup5m %d\r' >
           /dev/ttyS0", mult);
    // execute call-string -> echo to terminal
    system(call);
    // increase shiftup counter
    // 5 min per shiftup call => three times in 15min
    mult = mult + 3;
```

```
}
```

Für einen `count`-Wert von 45 (d.h. 45 Minuten sind vergangen) würden folgende zwei Terminaleingaben generiert und ausgeführt:

```
echo -e -n 'setbounds 20000:20100\r' > /dev/ttyS0
echo -e -n 'shiftup5m 9\r' > /dev/ttyS0
```

Der BTnode würde dann seine Zeitgrenzen auf $T^l = 20000 +$ `t5min[8]` und $T^u = 20100 +$ `t5min[8]` setzen. Da `count`, `mult`, das Feld `t5min` und ebenfalls die Terminaleingabe `shiftup5m s16` zusammenspielen, muss der Benutzer bei Änderungen der Eingabehäufigkeit oder der für das Setzen der Zeitgrenzen verwendeten Werte (`t5min`) immer alle beteiligten Variablen und Funktionen überprüfen.

- **void sendInt()**

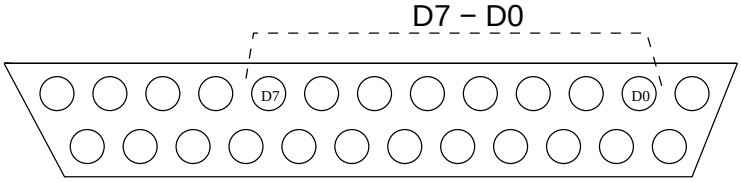

Abbildung 5: Der Parallelport mit Ausgängen D0 - D7.

Die Funktion `sendInt` ist dafür zuständig, auf den am Parallelport angeschlossenen BTnodes einen Interrupt auszulösen. In Abbildung 5 ist eine parallele Schnittstelle mit ihren Ein- und Ausgängen dargestellt. Die Pins, die für die Datenausgabe verwendet werden, sind D0 - D7. Auf einem UNIX-System können nun diese Pins sehr einfach angesprochen werden, indem man auf den entsprechenden Speicherbereich (siehe BIOS) schreibt oder ihn ausliest. Nachdem man die Zugriffsrechte auf den Parallelport mit

```
if(ioperm(BASEPORT, 1, 1)) {
        // Fehler
}
// OK
```

erlangt hat, kann mit

```
outb(15, BASEPORT);
```

direkt ein Byte in den zugehörigen Adressbereich geschrieben werden. Das Schreiben der Zahl 15 wird dabei als Bitmaske interpretiert und bewirkt, dass die Pins D0 bis D3 auf logisch *eins* geschaltet werden. Dieser Spannungswechsel löst auf den angeschlossenen BTnodes ein Signal, d.h. ein Interrupt, aus. Die Spannung muss einige Mikrosekunden auf *eins* gehalten werden, um sicherzustellen, dass der Interrupt tatsächlich ausgelöst wird.

4.4 Messergebnisse

4.4.1 Synchronisationsvorgang

Entscheidend ist, wie gut die Zeitgrenzen nach einem Synchronisationsvorgang bei dem Knoten sind, der anfangs schlechtere Grenzen hatte. Die folgenden zwei Faktoren beeinflussen diesen Vorgang am stärksten und sollen daher genauer betrachtet werden: Die Unsicherheit über die Verzögerung, d.h. wie gut diese mit dem verwendeten Kommunikationsalgorithmus minimiert werden konnte, und die Zeit, welche für lokale Berechnungen benötigt wird. *Kommunikationsalgorithmus:* Die Genauigkeit des Kommunikationsalgorithmus wird einerseits durch die Wahl von d_{max} und d_{min} und andererseits durch die gewählte Methode zur Speicherung von Sende- und Empfangszeiten (siehe Abschnitt3.3.3) bestimmt. *Lokale Berechnungen:* Die Zeit, die für lokale Berechnungen benötigt wird, ist trotz der Einfachheit Berechnungen, nicht zu unterschätzen. Der Mikrokontroller eines BTnodes ist mit 4MHz getaktet, weshalb die verwendeten Rechenoperationen für grosse Ganzzahlen (siehe Abschnitt 3.4) gut einige Millisekunden dauern können (z.B. Prozedur `calculate_bounds(...)`: ca. 2ms). Da aber diese Zeit erstens bei allen Knoten verstreicht und ausserdem sehr hardwareabhängig ist, wurde darauf verzichtet, sie zu berücksichtigen.

Die Vergrösserung der Unsicherheit über die Echtzeit bei einem Knoten während eines Synchronisationsvorgangs hängt also in unserem Fall hauptsächlich von den gewählten Werten d_{min} und d_{max} ab. Empirisch wurden eine minimale Roundtripzeit von 40 Millisekunden und eine durchschnittliche Roundtripzeit von 55 Millisekunden gemessen. Deshalb wurde d_{min} (in Abschnitt 2.1 $d_{min}/2$) initial auf 20 Millisekunden und d_{max} auf 60 Millisekunden festgesetzt. Dadurch ergeben sich natürlich bei der Berechnung der Zeitgrenzen und der Verzögerung (siehe Abschnitt 2.2) Fehler. Die Unsicherheit über die Echtzeit (Differenz der Zeitgrenzen) nimmt durch eine Synchronisation mit obiger Konfiguration maximal 20 Millisekunden (60 - 2 * 20) zu, bewegt sich aber im Mittel zwischen 6 und 12 Millisekunden. Durch Verkleinern von d_{max} könnte diese Zunahme der Unsicherheit theoretisch minimiert werden, in der Praxis hat sich aber $d_{max} < 60$ nicht bewährt, weil Roundtripzeiten unter 55 Millisekunden eher selten vorkommen. Das heisst, dass dann Kommunikationsnachrichten maximal oft

wiederholt werden, ohne das Kommunikationsprotokoll abzuschliessen. Die Anzahl maximaler Wiederholungen sehr hoch zu wählen hat sich aufgrund des momentan verwendeten Bluetooth-Moduls ebenfalls nicht bewährt. Es traten dann vielfach gravierende Kommunikationsfehler auf, die mit einem Absturz des Systems enden.

4.4.2 Testszenarien

Alle Testläufe wurden auf der beschriebenen Testumgebung mit vier BTnodes ausgeführt. Der Synchronisationsalgorithmus läuft dabei auf allen Knoten, die sich folgendermassen synchronisieren: Periodisch bestimmt jeder Knoten zufällig einen Synchronisationspartner und tauscht mit diesem die Zeitgrenzen aus. Die genaue Funktionsweise ist in 3.2 beschrieben. Gleichzeitig erhält ein bestimmter Knoten, der sogenannte *Anker*-Knoten, periodisch genaue Zeitgrenzen vom angeschlossenen PC. Die Synchronisationsgenauigkeit wird also von den beiden Parametern Häufigkeit der Synchronisationen und Häufigkeit der externen Zeiteingaben beeinflusst. Messungen mit verschiedenen Parametern wurden durchgeführt und sollen hier kurz diskutiert werden. Dabei ist zu beachten, dass für die Tests immer nur vier BTnodes verwendet wurden. Weil so alle Knoten zum Anker-Knoten Kontakt haben, besteht die Möglichkeit, dass sie von ihm immer sehr gute Zeitgrenzen erhalten. Daher kann z.B. nicht getestet werden, wie schnell und wie gut sich die Synchronisation in einem Netz "ausbreitet". Auch andere Effekte könnten erst in einem viel grösseren Netz von Sensoren beobachtet werden.

Die Zeitgrenzen werden wie beschrieben während des Testlaufs lokal auf den BTnodes gespeichert und können nach Abschluss des Tests via Terminal ausgegeben und verarbeitet werden. Da der Speicher des BTnodes begrenzt ist, kann nur eine beschränkte Anzahl Zeitmessungen (Interrupts) durchgeführt werden. Damit ist die maximal mögliche Dauer eines Testlaufs auch davon abhängig, wie oft die Zeitgrenzen lokal gespeichert werden. Bei allen folgenden Tests wurde von der PC-Applikation jede Minute ein Interrupt generiert. Dies ermöglicht eine Testdauer von gut drei Stunden, in denen alle BTnodes 180 Zeitwerte speichern.

Um die Messwerte zu untersuchen und zu visualisieren, ist die Darstellungsweise in Abbildung 6 zweckmässig: Der von links nach rechts verlaufende, konusartige Graph stellt den Verlauf der Zeitgrenzen während der vergangenen Zeit dar. Dabei sind nicht die eigentlichen Zeitgrenzenwerte T^l und T^u dargestellt, sondern $T^l - t$ und $T^u - t$, also die Zeitgrenzen minus der Echtzeit. Der obere Ast des Graphen entspricht jetzt dem Verlauf der oberen Zeitgrenze, der untere Ast dem Verlauf der unteren Zeitgrenze. Die X-Achse stellt die Zeit dar, während die Y-Achse die Abweichung der Zeitgrenzen vom Nullpunkt darstellt. So lässt sich die Zunahme der Unsicherheit $(T^u - T^l)$, die durch den Drift der Uhr (siehe Abschnitt 2.1) hervorgerufen wird, sehr gut erkennen. Die X-Werte, bei welchen die beiden Grenzen 'nahe' beieinan-

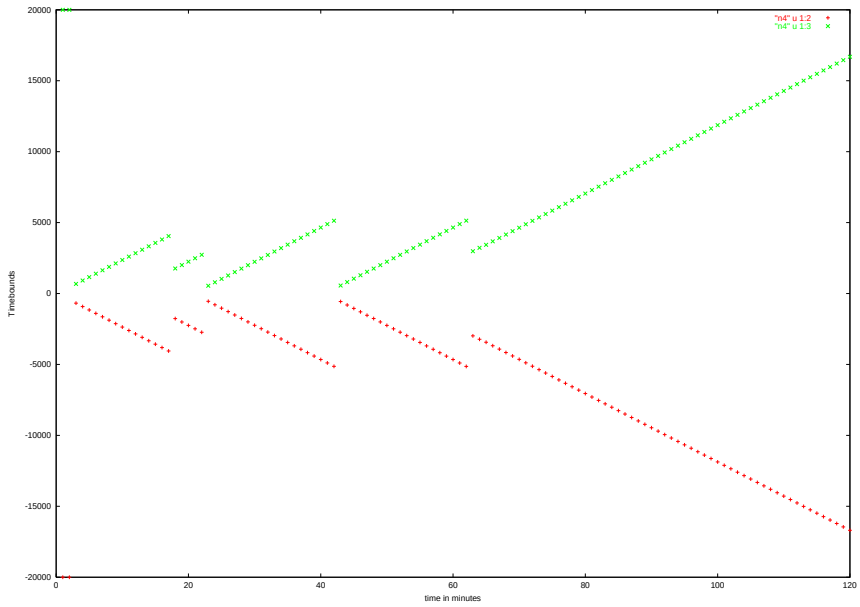

Abbildung 6: Grafik eines Testresultats.

der sind und danach auseinanderlaufen, sind Synchronisationspunkte, resp. externe Zeiteingaben im Falle des Anker-Knotens. Auffallend an der Grafik sind folgende zwei Punkte, die hier erklärt werden sollen: *1.* Die Zeitgrenzen laufen symmetrisch auseinander. *2.* Die Unsicherheit wächst enorm ohne Synchronisation, d.h. das Intervall $T^u - T^l$ wird sehr gross (ca. 8000 Millisekunden bei 18 Minuten). *Zu 1*: Um eine Darstellung zu erhalten, die die Funktionsweise des Algorithmus gut sichtbar macht, wurde nicht die Echtzeit abgezogen, sondern eine Approximation davon. Diese Approximation berechnet sich durch $T^l - (T^u - T^l)/2$, $T^u - (T^u - T^l)/2$. Dadurch wird der Graph symmetrisch. Eine Betrachtung ohne diese Approximation wird bei den Testläufen in Abschnitt 4.4.3 besprochen. *Zu 2*: Die Echtzeit wird den BTnodes wie oben beschrieben durch den PC mitgeteilt. Ein Problem gibt es dann, wenn die beiden Uhren sehr unterschiedliche Geschwindigkeiten besitzen. Es können dann falsche Zeitgrenzen berechnet werden, wie folgendes Beispiel zeigt: Angenommen wir haben zwei Knoten, P und Q, wobei P der vom PC mit aktueller Zeitinformation belieferte Anker-Knoten (mit initialisierten Zeitgrenzen) ist. Die Uhr auf Q läuft nun 1/240 langsamer als die Referenzuhr auf dem PC, d.h. wenn auf der Referenzuhr vier Minuten (240s) verstrichen sind, sind auf dem Knoten Q erst 239 Sekunden vergangen. Der Drift der internen Uhren der Knoten sei durch die Konstante $\rho^{\max} = 100$ ppm [10] beschränkt (d.h. pro 10 Sekunden 1 Millisekunde Abweichung, resp.

[10] ppm: parts per million.

pro 8 Minuten 48 Millisekunden).

```
PC - Uhr:          Knoten P:          Knoten Q:
-----------------------------------------------------
10'000, 11'000     10'000, 11'000     0, unendlich
                 Synchronisation
10'000, 11'000     10'000, 11'000     10'000, 11'000
         Verstreichen von 8 Minuten Echtzeit,
             d.h. 478 Sekunden auf BTnode
490'000, 491'000   487'952, 489'048   487'952, 489'048
             Externer Zeitinput an P
490'000, 491'000   490'000, 491'000   487'952, 489'048
                 Synchronisation
490'000, 491'000   490'000, 489'048   490'000, 489'048
```

Knoten P hat nun nach der letzten Synchronisation falsche Zeitgrenzen berechnet, weil die Uhr des Knotens Q viel langsamer läuft als die Echtzeit. Der Ursprung für den Fehler ist, dass die *PC-Zeit* $\neq$ *Echtzeit* ist. Dies kann korrigiert werden, indem man die Konstante $\rho^{\max}$ für den Drift grösser wählt, daher nicht an die Echtzeit sondern an die PC-Zeit anpasst. In unserem Fall ist eine Konstante $\rho^{\max} = 4'000$ nötig, um den Geschwindigkeitsunterschied der beiden Uhren auszugleichen. Die Zeitgrenzen der einzelnen Knoten sind dann, durch den grösseren Drift, bei der nächsten Synchronisation 'schlechter', wie das im folgenden Beispiel verdeutlicht wird. Maximaler Drift $\rho = 4'000$ ppm (d.h. pro 10 Sekunden 40 Millisekunden Abweichung, resp. pro 8 Minuten 1'920 Millisekunden).

```
PC - Uhr:          Knoten P:          Knoten Q:
-----------------------------------------------------
10'000, 11'000     10'000, 11'000     0, unendlich
                 Synchronisation
10'000, 11'000     10'000, 11'000     10'000, 11'000
         Verstreichen von 8 Minuten Echtzeit,
             d.h. 478 Sekunden auf BTnode
490'000, 491'000   486'088, 490'912   486'088, 490'912
             Externer Zeitinput an P
490'000, 491'000   490'000, 491'000   486'088, 490'912
                 Synchronisation
490'000, 491'000   486'088, 490'912   486'088, 490'912
```

Infolge der grossen Konstante $\rho^{\max}$ nimmt die Unsicherheit über die Echtzeit in der Grafik sehr schnell stark zu.

4.4.3 Testläufe

Es wurden mehrere Testläufe gemacht, wobei einerseits die Häufigkeit der Synchronisationen und andererseits die Häufigkeit der externen Zeiteingaben variiert wurden. In diesen Grafiken wurden nun jeweils zwei Knoten eingezeichnet: Der Anker-Knoten und einer der restlichen drei BTnodes als Vergleich. Die Zeitgrenzen des Anker-Knotens driften natürlich nie weit auseinander, da er alle fünf Minuten eine neue Zeiteingabe erhält. Ausserdem sieht man sehr schön die gleichweit fortlaufenden, symmetrischen Konen, welche ebenfalls durch die periodische Synchronisation in fünf-Minuten-Abständen hervorgerufen werden. Die durchgeführten Testläufe sollen hier besprochen werden:

1. **Zeiteingabe: alle 5min, Synchronisation: alle 10min**
   Ein Testlauf mit periodischen Zeiteingaben alle 5 Minuten und Synchronisation der beteiligten Knoten alle 10 Minuten ist in Abbildung 7 dargestellt. Der zweite Knoten nimmt hier ungefähr alle 20 Minuten an einer Synchronisation teil. Dies erkennt man daran, dass bei einem Synchronisationsvorgang die Unsicherheit über die Echtzeit wieder klein wird und die beiden Zeitgrenzen nahe beieinanderliegen. In diesem Testlauf hat sich der zweite Knoten, aus oben genannten Gründen, beinahe immer mit dem Anker-Knoten synchronisiert. Dies äussert sich darin, dass die Konusäste aufeinanderliegen. Zur Zeit $t = 32$ min hat eine Synchronisation mit einem anderen Knoten stattgefunden, dessen Zeitgrenzen zwar besser, aber nicht annähernd so gut wie diejenigen des Ankerknotens waren.

2. **Zeiteingabe: alle 5min, Synchronisation: alle 20min**
   Ein Testlauf mit periodischen Zeiteingaben alle 5 Minuten und Synchronisation der beteiligten Knoten alle 20 Minuten ist in Abbildung 8 dargestellt. Im Vergleich zur Abbildung 7 zeigt dieser Testlauf wie erwartet, dass die Unsicherheit über die Echtzeit bei weniger häufiger Synchronisation zunimmt. Hier wird auch deutlich, dass die Knoten nicht unbedingt zu jedem gewollten Synchronisationszeitpunkt einen Partner finden, sonst wären in regelmässigeren Abständen Synchronisationen sichtbar.

3. **Zeiteingabe: alle 5min, Synchronisation: alle 30min**
   Ein Testlauf mit periodischen Zeiteingaben alle 5 Minuten und Synchronisation der beteiligten Knoten je alle 30 Minuten ist in Abbildung 9 dargestellt. Auch in diesem Testlauf, mit noch grösseren Intervallen zwischen den Synchronisationsvorgängen, nimmt die Unsicherheit über die Echtzeit weiter zu. Es lässt sich aber in Abbildung 9 gut erkennen, dass sich der abgebildete Knoten nur einmal mit dem Anker-Knoten synchronisiert hat. Die übrigen Synchronisationen (bei

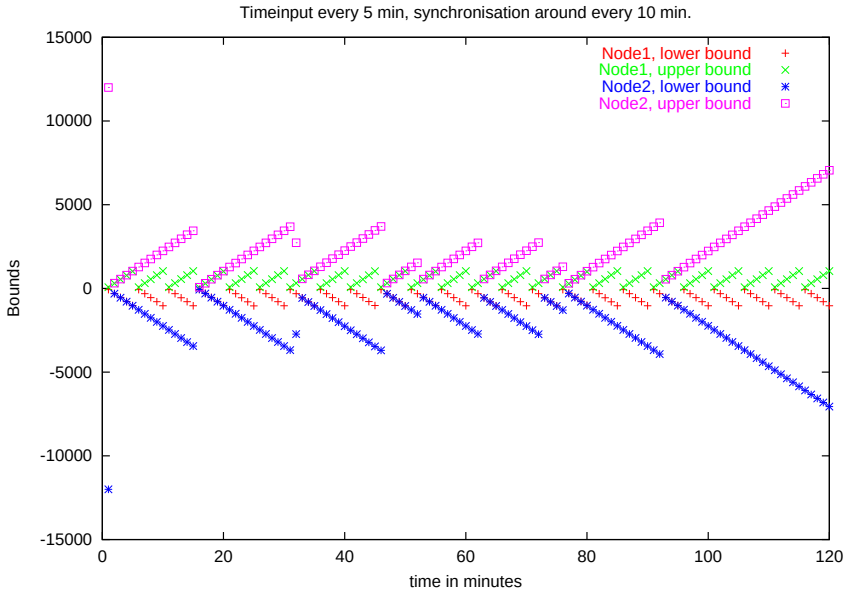

Abbildung 7: Testlauf 1, Input: 5min, Sync: 10min.

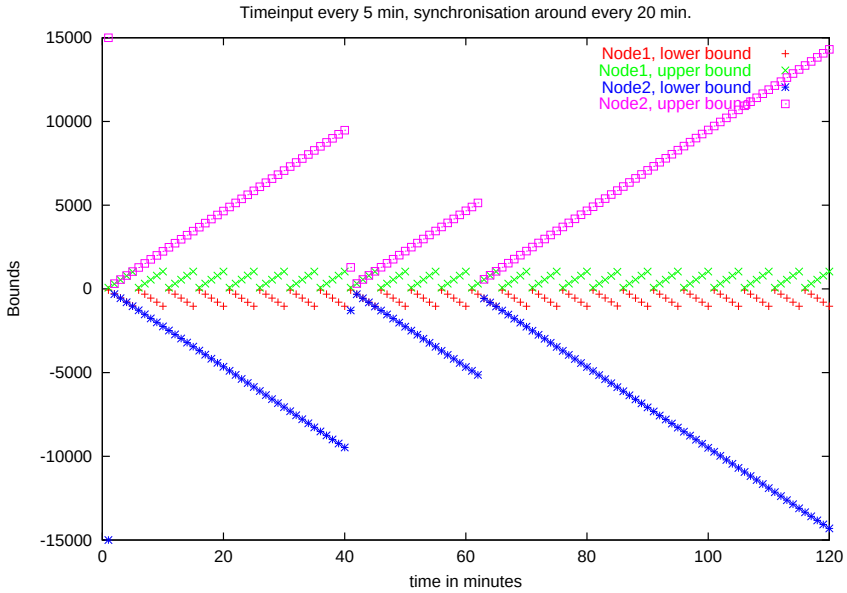

Abbildung 8: Testlauf 2, Input: 5min, Sync: 20min.

$t = 90$ und $t = 150$ Minuten) fanden mit anderen teilnehmenden Knoten statt, welche offenbar bessere Zeitgrenzen besassen.

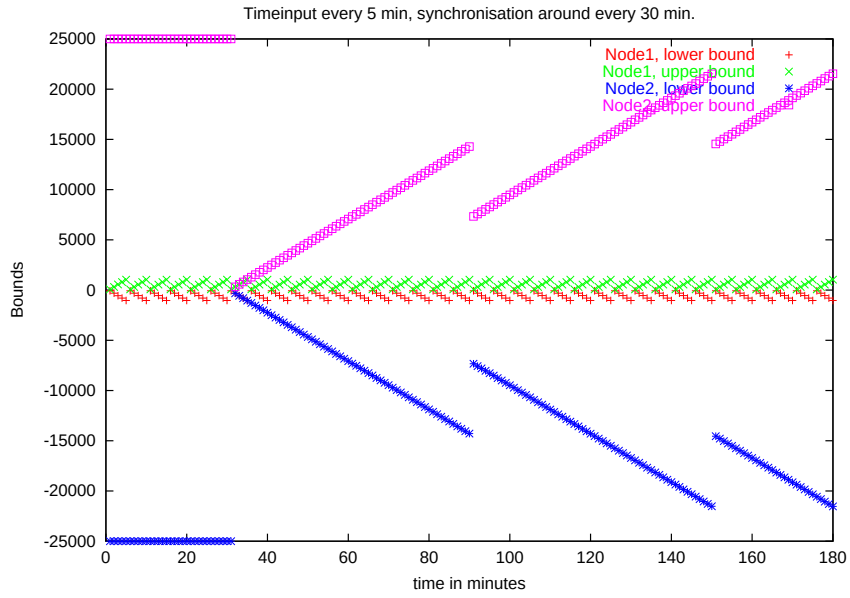

Abbildung 9: Testlauf 3, Input: 5min, Sync: 30min.

4. **Zeiteingabe: alle 1h, Synchronisation: alle 1h**
   Ein Testlauf mit periodischen Zeiteingaben jede Stunde und Synchronisation der beteiligten Knoten ebenfalls alle Stunden ist in Abbildung 10 dargestellt. Ebenfalls wie erwartet nimmt die Synchronisationsgenauigkeit bei seltenen Zeiteingaben und zusätzlich bei seltenen Synchronisationsvorgängen noch mehr ab. Der Testlauf zeigt allerdings, dass der Algorithmus auch über einen längeren Zeitraum, hier 500 Minuten, funktioniert.

5. **Gleich wie Testlauf 1, aber ohne Approximation**
   In Abbildung 11 wurden die gleichen gesammelten Daten zur Darstellung verwendet wie in Testlauf 1 (Abbildung 7). Der Unterschied liegt bei der Aufbereitung der Daten zur Darstellung. Während bei den Testläufen 1 bis 4 jeweils die Approximation $T^l - (T^u - T^l)/2$, $T^u - (T^u - T^l)/2$ verwendet wurde, wurde hier die dem Anker-Knoten mitgeteilte Echtzeit abgezogen, d.h. $T^l - t$, $T^u - t$. Die Grafik zeigt nun, dass die Echtzeit sehr nahe der unteren Zeitgrenze liegen muss, weil diese sehr flach verläuft. Die Unsicherheit über die Echtzeit bleibt natürlich dieselbe wie in Abbildung 7, weshalb die oberen Äste des Graphen sehr steil ansteigen.

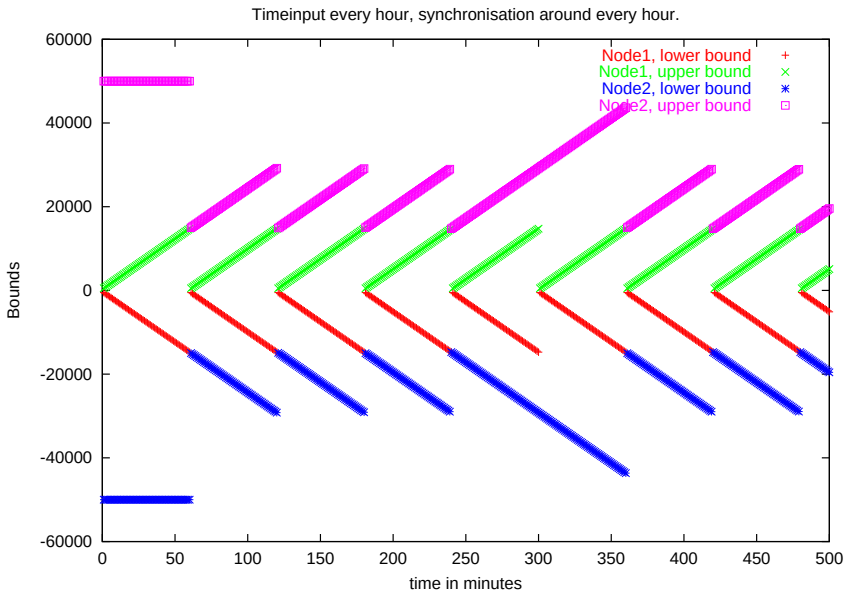

Abbildung 10: Testlauf 4, Input: 1 h, Sync: 1 h.

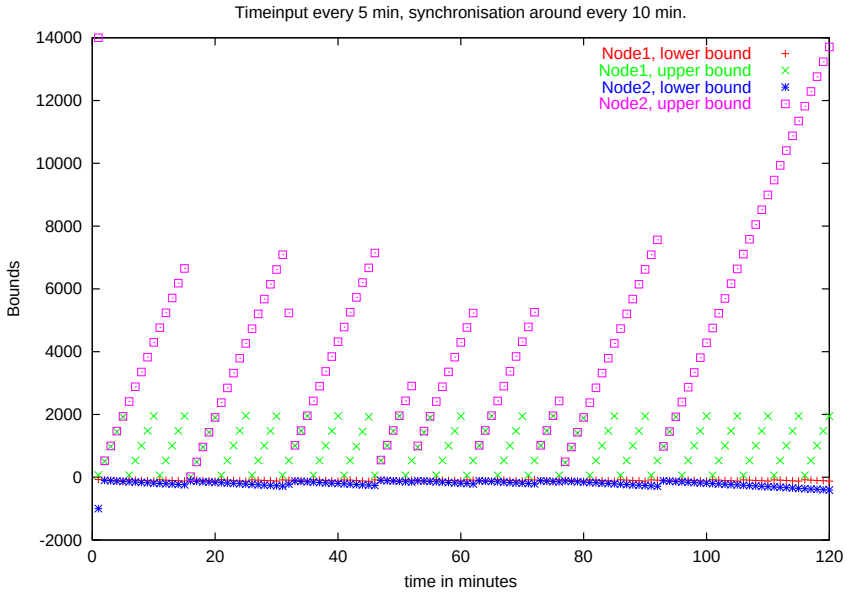

Abbildung 11: Testlauf 1, ohne Approximation.

4.4.4 Interpretation

Zusammenfassend lässt sich anhand der obigen Ergebnisse feststellen, dass die Implementierung des Algorithmus funktioniert. Für genauere Untersuchungen über das Verhalten des Synchronisationsalgorithmus wäre es sinnvoll, in einer weiteren Arbeit grössere Netze zu verwenden.

4.5 Alternative Testumgebung

Ursprünglich war geplant, die Zeitwerte via Bluetooth an einen zentralen Knoten zur Auswertung zu senden, was jedoch mit einigen Problemen verbunden war. Wenn zwei Knoten eine Bluetooth-Verbindung aufgebaut haben, ist der eine im Zustand `master` und der andere im Zustand `slave`. Ein Knoten im `slave`-Zustand kann von anderen Teilnehmern bei einem `inquiry`[11] nicht mehr gefunden werden, er ist also nicht mehr "sichtbar". Da die BTnodes untereinander den Synchronisationsalgorithmus ausführen, dürfen sie mit keinem anderen Gerät, also auch nicht mit dem zentralen Knoten, eine ständige Verbindung eingehen, damit sie für potentielle Synchronisationspartner "sichtbar" bleiben und der Algorithmus überhaupt funktionieren kann. Damit bliebe nur die Möglichkeit, den zentralen Knoten im Zustand `slave` zu betreiben, womit die BTnodes eine Verbindung zu diesem initiieren müssten. Während jeder Behandlung eines Interrupts müsste nun ein BTnode eine Verbindung zum zentralen Knoten aufbauen, seine Zeitinformation senden und die Verbindung wieder abbauen. Gerade dieses häufige Auf- und Abbauen von Verbindungen mehrerer Knoten gleichzeitig hat sich bei der Verwendung von BTnodes als sehr instabil erwiesen, weshalb diese Variante zur Auswertung aufgegeben werden musste. Für eine spätere Implementierung einer neuen Testumgebung mit anderen, stabileren Bluetoothmodulen dürfte aber dieser Ansatz wesentlich flexibler und praktikabler sein als die Ausgabe der Testdaten über einen Terminal.

5 Zusammenfassung und Ausblick

Die in den Abschnitteni 2 und 3 vorgestellten Algorithmen wurden auf BTnodes implementiert und getestet. Durch die Verwendung des speziellen Kommunikationsalgorithmus konnte die Unsicherheit über die Verzögerung deutlich minimiert werden. Die Testergebnisse zeigen, dass die Algorithmen funktionieren. Bei vertiefenden Arbeiten wäre es interessant herauszufinden, wie sich der Algorithmus in einer grösseren Testumgebung verhält. Ausserdem könnte dann auch getestet werden, inwiefern äussere Einflüsse, wie Temperatur- oder Spannungsschwankungen, die Synchronisation beeinflussen.

[11] inquiry ist der Vorgang des Suchens anderer Bluetooth Geräte in der Umgebung.

6 Literatur

Literatur

[1] P. Blum, L. Meier, and L. Thiele. Improved interval-based clock synchronization in sensor networks. In *Proceedings of the Third International Symposium on Information Processing in Sensor Networks (IPSN '04)*, Berkeley, California, USA, April 2004.

[2] J.C. Haartsen. The Bluetooth Radio System. *IEEE Personal Communications*, Febr. 2000.

[3] Keith Marzullo and Susan Owicki. Maintaining the time in a distributed system. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 295–305. ACM Press, 1983.

[4] BTnodes - A Distributed Environment for Prototyping Ad Hoc Networks. `http://www.btnode.ethz.ch.`

A Implementierung

A.1 Synchronisationsalgorithmus

Das File `synchro.c` mit den Implementierung des Synchronisations- und Kommunikationsalgorithmus.

```
/*
 * synchro.c
 *
 * 2003.09.05 Boris Zweimueller <zboris@student.ethz.ch>
 *
 * This file is free software; you can redistribute it and
 * /or modify it under the terms of the GNU General Public
 * License as published by the Free Software Foundation;
 * either version 2 of the License, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even the
 * implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE. See the GNU General Public License
 * for more details.
 *
 * (c) ETH Zurich
 */

/*---------------------------------------------------------------
 * Includes
 *---------------------------------------------------------------
 * */

#include <ctype.h>
#include <stdlib.h>

#include <string.h>
#include <btnode.h>
#include <drivers.h>
#include <bt_rfcomm.h>
#include <bt_rfcomm_diagnostics.h>

#include <dispatcher.h>
#include <debug_uart.h>
#include <bigint.h>


/*---------------------------------------------------------------
 * defines and global variables
 *---------------------------------------------------------------
 * */
```

```
/* terminal */

// what does your terminal generate when you hit enter?
#if defined(__linux__) || defined(__APPLE__)
#define CR '\n'
#elif defined (__AVR__)
// linux/ minicom
#define CR '\r'
#endif // __AVR__


/* buffers */

// buffer for user input on uart
#define READLN_BUF 80

// buffer for incoming data packets
#define DATA_BUF_LEN 4
l2cap_data_buffer_elem_t data_buf[DATA_BUF_LEN];


/* for the exchange of bounds - communication algorithm */

// length of the time message sent to exchange timebounds
// message format: |m-type|p-time|lower|upper|
#define TIME_MESSAGE_LENGTH (2 * sizeof(bigint)) + 2

// how many times we repeat sending a message
// if the delay was to big
#define MAX_SEND_REPETITION 100

// for delay calculations
u8 sent = 0;
u8 received = 255;
u8 dborder = 60; // in milliseconds
u8 dmin = 20;
// to store how many messages have been sent in the
// actual "round"
u8 message_counter = 0;


/* for the synchronization algorithm */

// to store my adress
bt_addr_t local_bd_addr;
// to store the address of the current partner
bt_addr_t current_remote_addr;
// to store temporily the last btnode synchronized with
```

```
bt_addr_t last_remote;

// my bounds
bigint lowerbound;
bigint upperbound;
// when the bounds have been set the last time
bigint timeboundsset;

// to store the synchronize timeout
u32 syncfactor = 20;


/* for testing */

// the maximal number of stored time events
#define MAX_STORE_VAL 1000

// the array for data storage, store lower and upperbound
// on every interrupt
bigint timevalues[2 * MAX_STORE_VAL];
// the next free index in the timevalues array
u32 pos = 0;

// fix time values for bound calculation during runtime
bigint t100000ms;
bigint t200000ms;
bigint t5min[36]; // for 3 hours in 5 min steps
bigint t1h;


/* some necessary forward declarations */

void sendmessage(u16 a, u16 b, u16 c);

/*-------------------------------------------------------
 * function calculate_bounds calculates the actual bounds
 *-------------------------------------------------------
 * */
inline void calculate_bounds() {
    bigint actualtime;
    bigint delta;
    u8 offset;

    // get actual time
    btn_rtc_get_64(&actualtime, &offset);
    // add offset (see /src/avr_128/avr_rtc.c)
    addS16bigint(&actualtime, offset);

    // calculate delta
```

```
    copybigint(&actualtime, &delta);
    subbigint(&delta, &timeboundsset);

    if (ismaxbigint(&upperbound) == 0) {
        bigint temp;
        // upperbound + currentsize is not <  maxbigint
        // calculate drift for faster clock
        copybigint(&delta, &temp);
        driftdivision(&temp, 1);
        // calculate new upper bound
        addbigint(&upperbound, &temp);
    }

    // lowerbound ist always updated
    // calculate drift for slower clock
    driftdivision(&delta, 0);
    // calculate new lowerbound
    addbigint(&lowerbound, &delta);

    // update when bounds have been set the last time
    copybigint(&actualtime, &timeboundsset);

    return;
}

SIGNAL (SIG_INTERRUPT7) {
    // disable interrupts
    cbi (EIMSK, INT7);

    // generate new event of type 1
    btn_disp_put_event(TIMEOUT_EV, 1);
    // clear bit
    cbi(PORTE, 7);
    // enable interrupts
    sbi (EIMSK, INT7);
}

/*-------------------------------------------------------------
 * Function display_prompt prints the menu and waits for
 * a new command.
 *-------------------------------------------------------------
 * */
void display_prompt() {
    // display menu
    printf(NL "Menu:" NL );
    printf("bounds 0xLow 0xUp    - external input of
          lower an upper bound." NL);
    printf("print                - printout my current
          time and bounds." NL);
```

```
    printf("sync ---------------- - searches for btnodes
            and synchronizes."NL);
    printf("syncfactor fac ------ - node synchronizes
----------every factor * 30 seconds."NL);
    printf("printtimevals ------- - prints the stored
----------time values to the terminal."NL);

    printf("to define bounds:"NL);
    printf("setbounds s16:s16 --- - sets lower and
----------upper bound to specified values."NL);
    printf("shiftup5m s16 ------- - shifts both
----------bounds up by s16 * 5 minutes."NL);
    printf("hourup -------------- - shifts both
----------bounds up by 1 hour."NL);

    // display command prompt
    printf( "[" );
    btn_print_bd_addr( local_bd_addr );
    printf( "] > " );
}

/*-------------------------------------------------------------
 * Function process_cmd processes user input.
 *-------------------------------------------------------------
 * */
void process_cmd( u8* cmd_buf ) {
    u32 time16;
    bigint actualtime;
    u8 offset;

    if(cmd_buf[0] == '\0') {
        display_prompt();
        return;
    }

    if (strncmp(cmd_buf, "shiftup5m", 9) == 0) {
        bigint actualtime;
        u8 offset;
        u32 d;
        u8 data[6];

        // extract multiplier
        memcpy(&data, cmd_buf+10, 5);
        data[5] = '\0';
        d = atoi(data);

        // update bounds
        calculate_bounds();
```

```
    // set bounds
    addbigint(&lowerbound, &t5min[d]);
    addbigint(&upperbound, &t5min[d]);
    printbigint(&t5min[d]);

    // update when bouns have been set the last time

    // get actual time
    btn_rtc_get_64(&actualtime, &offset);
    // add offset (see /src/avr_128/avr_rtc.c)
    addS16bigint(&actualtime, offset);

    copybigint(&actualtime, &timeboundsset);
    display_prompt();
    return;
}

if (strncmp(cmd_buf, "hourup", 6) == 0) {
    bigint actualtime;
    u8 offset;

    // update bounds
    calculate_bounds();

    // set bounds
    addbigint(&lowerbound,&t1h);
    addbigint(&upperbound,&t1h);

    // update when bouns have been set the last time

    // get actual time
    btn_rtc_get_64(&actualtime, &offset);
    // add offset (see /src/avr_128/avr_rtc.c)
    addS16bigint(&actualtime, offset);

    copybigint(&actualtime, &timeboundsset);
    display_prompt();
    return;
}

if (strncmp(cmd_buf, "syncfactor", 10) == 0) {
    u32 d;
    u8 data[6];
    memcpy(&data, cmd_buf+11, 5);
    data[5] = '\0';
    d = atoi(data);
    syncfactor = d;
    printf("stored new sync factor %d"NL, d);
    display_prompt();
```

```
    return;
}

// has to come after syncfactor
if(strncmp(cmd_buf, "sync", 4 ) == 0 ) {
    printf( "Synchronizing" NL );
    timeout_cb(0,0);
    display_prompt();
    return;
}

if (strncmp(cmd_buf, "printtimevals", 13) == 0) {
    u32 i;

    for (i = 0; i < pos; i++) {
        printf("%d      ",i);
        printbigint(&timevalues[i++]);
        printf("    ");
        printbigint(&timevalues[i]);
        printf(NL);
    }
    display_prompt();
    return;
}

// must be after 'printtimevals' to not conflict
if(strncmp(cmd_buf, "print", 5 ) == 0 ) {
    time16 = btn_rtc_get();
    // get actual time
    btn_rtc_get_64(&actualtime, &offset);
    // add offset (see /src/avr_128/avr_rtc.c)
    addS16bigint(&actualtime, offset);


    calculate_bounds();

    printf("systemtime 16bit: %d"NL,time16);
    printf("systemtime 64bit dec: ");
    printbigint(&actualtime);
    printf(NL);

    printf("my upperbound is: ");
    printbigint(&upperbound);
    printf(NL);
    printf("my lowerbound is: ");
    printbigint(&lowerbound);
    printf(NL);
    btn_print_l2cap_channel_table();
    btn_print_baseband_connection_table();
```

```
        printf("dmin: %d"NL, dmin);
        printf("dborder: %d"NL, dborder);
        printf("syncfactor: %d"NL, syncfactor);
        display_prompt();
        return;
    }

    if (strncmp(cmd_buf, "setbounds", 9) == 0) {
        bigint actualtime;
        u8 offset;

        s16 val_l;
        s16 val_u;
        u8 data_l[6];
        u8 data_u[6];
        memcpy(&data_l, cmd_buf+10, 5);
        memcpy(&data_u, cmd_buf+16, 5);
        data_l[5] = '\0';
        data_u[5] = '\0';
        val_l = atoi(data_l);
        val_u = atoi(data_u);

        // get actual time
        btn_rtc_get_64(&actualtime, &offset);
        // add offset (see /src/avr_128/avr_rtc.c)
        addS16bigint(&actualtime, offset);
        // set bounds
        s16tobigint(val_l, &lowerbound);
        s16tobigint(val_u, &upperbound);
        // update when bouns have been set the last time
        copybigint(&actualtime, &timeboundsset);
        printf("bounds have been set to: (lower, upper) \
                %d, %d" NL, val_l, val_u);
        display_prompt();
        return;
    }

    printf( "Command not recognized." NL );
    display_prompt();
}

//-------------------------------------------------------------
// function to handle 'data receive' events
//-------------------------------------------------------------
void data_received_cb(call_data_t call_data,
        cb_data_t cb_data) {

    bigint recv_l_bound;
```

```
    bigint recv_u_bound;
    u8 error_code;
    conn_el_t* bb_handle;
    u8 lc;
    u8 type;
    u8 ptime;
    u8 channel_pos;

    // store time on which message arrived
    received = inp(TCNT0);

    // index of the data element
    u8 data_idx       = (u8)((call_data & 0x0000FF) >>  0);

    lc = l2cap_data_buffer[data_idx].local_cid;
    channel_pos = getChan((u16)lc);

    // unpack message
    memcpy(&type, l2cap_data_buffer[data_idx].data, 1);
    memcpy(&ptime, l2cap_data_buffer[data_idx].data + 1,1);

    // check if tried too many times
    if (message_counter == MAX_SEND_REPETITION) {
        // tried to many times to get a 'good' delay,
        // disconnect and do not update bounds
        printf("unable to get a 'good' delay: tried %d
          times, give up now. disconnect."NL,
            message_counter);

        // disconnect
        btn_bt_disconnect(l2cap_data_buffer[data_idx].
            local_cid);
        bb_handle = btn_bt_get_connection_of_local_cid(lc);
        btn_bt_bb_disconnect(bb_handle,
            DISCON_USER_ENDED_CON, &error_code );
        // init remote adress field with zero
        memcpy(current_remote_addr, 0, BT_ADDR_LEN);

        // reset message sent counter to zero
        message_counter = 0;
    }
    else {
        // check type
        switch(type) {
            case 0:
                // it is an "initialize" message in any
                   case
                // we have to send our bounds to the remote
                // node and ask for his bounds, because we
```

```
            // do not know anything about the delay of
            // his first message ==> message type one
            sendmessage(channel_pos, lc, 1);
            break;
        case 1:
            // It is a message of type one. This means
            // the remote node wants tho know our
            // bounds. (and is sending his bounds with
            // the message)
            if ((u8)((received - sent) - ptime) <
                    dborder) {
                // the delay was ok, send message
                // type 2, e.g. don't ask for
                // remote bounds

                // do bound calculation
                calculate_bounds();

                memcpy(&recv_l_bound,
                        l2cap_data_buffer[data_idx].
                        data + 2, sizeof(bigint));
                memcpy(&recv_u_bound,
                        l2cap_data_buffer[data_idx].
                        data + 2 + sizeof(bigint),
                        sizeof(bigint));

                // check received bounds
                //
                // if roundtrip - dmin < dmin =>
                // found new dmin => update dmin
                if ( (((u8)(received - sent) -
                                ptime) - dmin) < dmin)
                                    {
                    dmin = (u8)(received - sent) -
                        ptime - dmin;
                    printf("NEW DMIN HAS BEEN FOUND ! \
                                        %d"NL, dmin);
                }

                // received lowerbound = received
                // lowerbound + minimal possible delay
                addS16bigint(&recv_l_bound, dmin);
                // if received upper bound is not
                // maximal add bigger part of the delay
                // to upper bound
                if (ismaxbigint(&recv_u_bound) == 0) {
                    addS16bigint(&recv_u_bound,
                            (s16)(((u8)(received - sent
                                )
```

```
                                - ptime) - dmin));
            }
            // take max(current lowerbound,
                received
            // lowerbound) as new lowerbound
            if (bigintbigger(&lowerbound,
                        &recv_l_bound) == 0) {
                copybigint(&recv_l_bound,
                        &lowerbound);
            }
            // take min(current upperbound,
                received
            // upperbound) as new upperbound
            if (bigintbigger(&upperbound,
                        &recv_u_bound) == 1) {
                copybigint(&recv_u_bound,
                        &upperbound);
            }
            // and send message type 2, e.g. only
                my
            // bounds to the remote node do not ask
            // for his bounds again
            sendmessage(channel_pos, lc, 2);
        }
        else {
            // the delay was not ok, ask remote
                node
            // again for his bounds ==> send
                message
            // of type one again
            sendmessage(channel_pos, lc, 1);
        }
        break;
    default:
        // T2 message received
        if ((u8)((received - sent) - ptime) <
                dborder) {

            // and make all the bound calculation

            calculate_bounds();

            // store data
            memcpy(&recv_l_bound,
                    l2cap_data_buffer[data_idx].
                    data + 2, sizeof(bigint));
            memcpy(&recv_u_bound,
                    l2cap_data_buffer[data_idx].
                    data + 2 + sizeof(bigint),
```

```
        sizeof(bigint));

// check received bounds
// received lowerbound = received
// lowerbound + minimal possible delay
addS16bigint(&recv_l_bound, dmin);
// if received upper bound is not
// maximal add bigger part of the
// delay to upper bound
if (ismaxbigint(&recv_u_bound) == 0) {
    addS16bigint(&recv_u_bound,
            (s16)(((u8)(received - sent
                )
                    - ptime) - dmin));
}
// take max(current lowerbound,
    received
// lowerbound) as new lowerbound
if (bigintbigger(&lowerbound,
            &recv_l_bound) == 0) {
    copybigint(&recv_l_bound,
            &lowerbound);
}
// take min(current upperbound,
    received
// upperbound) as new upperbound
if (bigintbigger(&upperbound,
            &recv_u_bound) == 1) {
    copybigint(&recv_u_bound,
            &upperbound);

}

// disconnect channel after receiving
// and sending bounds
btn_bt_disconnect(
        l2cap_data_buffer[data_idx].
        local_cid);
bb_handle =
    btn_bt_get_connection_of_local_cid(
        lc);
btn_bt_bb_disconnect(bb_handle,
        DISCON_USER_ENDED_CON,
        &error_code );

// init remote adress field with zero
memcpy(current_remote_addr, 0,
        BT_ADDR_LEN);
```

```
                    // reset message_counter to zero
                    message_counter = 0;
                }
                else {
                    // the delay was to large, ask the
                    // remote node for his bounds again !
                    // ==> send message of type one again
                    sendmessage(channel_pos, lc, 1);
                }
            }
        }

        // free the buffer
        btn_bt_data_free(data_idx);
}

/*-------------------------------------------------------------
 * function to send timebound message to the remote device
 *
 * messagetype is 0 or 1, see datareceive event handler for
 * explanation
 *-------------------------------------------------------------
 * */
void sendmessage(u16 channel_pos, u16 local_cid, u16 mt) {
    u8 error_code;
    u8 message[TIME_MESSAGE_LENGTH];
    u8 type = (u8)mt;
    u8 type = 0;
    u8 ptime = 0;

    calculate_bounds();

    memcpy(message, &type, 1);
    memcpy(message + 2, &lowerbound, sizeof(bigint));
    memcpy(message + 2 + sizeof(bigint), &upperbound,
            sizeof(bigint));

    // increment sent messages counter
    message_counter++;

    // calculate process time and store it in message
    ptime = (u8)inp(TCNT0) - received;
    memcpy(message + 1, &ptime, 1);

    // stamp send time
    sent = inp(TCNT0);

    // send data to remote device
    if (btn_bt_data_send(channel_pos, message,
```

```
                TIME_MESSAGE_LENGTH, &error_code) !=
            TIME_MESSAGE_LENGTH) {
        printf("error sending data");
    }
    // data sent succesfully
}

/*-------------------------------------------------------------
 * Function to handle connection events
 *
 * Triggered when a connection has been established or when
 * an other
 * device wants to build a connection with us.
 *
 * First check the psm on which the event arrives:
 *
 * If it is psm 103 it is a connection event of a partner
 * node which participates in the synchronisation procedure
    .
 *
 * On psm 103 test bt_addr of incoming event to check if it
 * is an establishment event of a node previously connected
 * to (current_remote_bt_addr).
 *
 * If it was a connection-establishing connection event we
 * have to send our timebounds to the remote node. If it
    was
 * a connection event indicating a new connection from an
 * other node we do not need to do anything.
 *-------------------------------------------------------------
 * */
void conn_cb(call_data_t call_data, cb_data_t cb_data) {
    static int i = 0;
    u16 local_cid;
    u16 local_psm;
    u16 channel_pos;
    conn_el_t* bb_handle;

    printf(NL "connection event %d" NL,i);
    // btn_print_l2cap_channel_table();
    local_psm = (u16)(call_data >> 16);
    local_cid = (u16)(call_data & 0xFFFF);
    printf("local_cid: %u, psm: %u" NL, local_cid,
            local_psm);

    // get channel index
    channel_pos = getChan((u16)local_cid);
    if( channel_pos == 0xFF ) {
        printf( "APP-ERROR: no local_cid: %i." NL,
```

```
                local_cid );
        return;
    }

    bb_handle = btn_bt_get_connection_of_bd_addr(
            current_remote_addr);
    // check if its an establishment event (eg. remote
    // adress same as remote adress connected to before)
    if (btn_bt_addr_equal(bb_handle->bt_addr,
                current_remote_addr)) {
        printf("establishment connection - \
        send my bounds"NL);
        // send init message
        sendmessage(channel_pos, local_cid, 0);
    }
    else {
        printf("probably a normal connection - \
        just do nothing"NL);
    }
    printf(NL "connection event %d completed" NL, i++);
}

/*-------------------------------------------------------------
 * function to handle timout events
 *-------------------------------------------------------------
 * */
void timeout_cb(call_data_t call_data, cb_data_t cb_data) {

    u8 num_devices = 0; // devices found on inquire
    u8 r_device = 0; // the device choosen to connect to
    u8 error_code;

    bt_addr_t devices[10];

    if ((u32)call_data == 1) {

        calculate_bounds();

        // store values
        if (pos < MAX_STORE_VAL) {
            copybigint(&lowerbound, &timevalues[pos++]);
            copybigint(&upperbound, &timevalues[pos++]);
            printf("bounds stored !"NL);
        }
        else {
            printf("array to store time values full !"NL);
        }
    }
    else {
```

```
        // event type 0
        static u32 sync = -6; //start soon with first sync!

        sync++;

        if (sync % syncfactor == 0) {
            printf("doing inquiry...." NL);

            // inquire for 4 * 1.28 seconds
            num_devices = btn_bt_inquire( 4, 10, devices );
            if( num_devices == 0 ) {
                printf("no devices found " NL );
            }
            else if( num_devices > 10 ) {
                printf( "Error during inquiry. error \
                        code: %i" NL, num_devices );
            }
            else {
                // found at least one device.
                // choose randomly one remote node and
                // send it my bounds
                printf("found %i device(s):" NL,
                        num_devices );
                r_device = btn_rtc_get() % num_devices;

                // do not connect to the same node twice
                if (num_devices > 1) {
                    // only now we have the possibility
                    // to choose another node
                    if (btn_bt_addr_equal(
                                devices[r_device],
                                last_remote)) {
                        // connected to this
                        // node the last time.
                        if (r_device == 0) {
                            r_device = 1;
                        }
                        else {
                            r_device--;
                        }
                    }
                    // not equal -> go on with choosen adr
                }

                printf("connect to device: %d with adress:"
                    ,
                        r_device);
                btn_print_bd_addr(devices[r_device]);
```

```
                // store the used adress
                memcpy(last_remote, devices[r_device],
                        BT_ADDR_LEN);

                // store current partner
                memcpy(current_remote_addr,
                        devices[r_device], BT_ADDR_LEN);
                // connect on psm 103
                btn_bt_connect(current_remote_addr, 103,
                        &error_code);
                if( error_code ) {
                    printf( "APP-ERROR: couldn't establish \
                                l2cap connection."
                            "error_code=%u."NL, error_code)
                                ;
                }
            }
        }
        btn_timeout_reg(TIMEOUT_EV, 30000, 0);
    }
}

void debug_input_cb( call_data_t call_data,
        cb_data_t cb_data ) {
    static u8 readln[READLN_BUF];
    static s16 index = 0;
    s32 count = 0;

    count = btn_debug_uart_read ( readln+index, 1 );

    if( count == 0 )
        return;
    if( readln[index] == CR ) {
        printf( NL );
        readln[index]='\0';
        index = 0;
        process_cmd( readln );
    }
    else {
        printf( "%c", readln[index] );
        index = index + count;
    }

    if( index == READLN_BUF ) {
        printf( "LINE TO LONG." NL );
        index = 0;
    }
}
```

```
/*-----------------------------------------------------------
 * main function
 *-----------------------------------------------------------
 * */
int main( int argc, char* argv[] ) {

#if defined(__linux__) || defined(__APPLE__)
    fflush( stdout );
    setvbuf( stdout, NULL, _IONBF, 0 );
#endif

    u32 i;

    // unsigned 8 bit value for error code
    u8 error_code;

    // the time bounds with zero and infinity
    s16tobigint(0, &lowerbound);
    setmaxbigint(&upperbound);
    s16tobigint(0, &timeboundsset);

    // init btnode
    btn_system_init( argc, argv, data_buf, DATA_BUF_LEN);

    btn_bt_psm_add( 103 );
    // init debug uart
    btn_debug_uart_init(DEBUG_UART1);

    // set local name
    btn_bt_change_local_name( "synchro", &error_code );

    // init remote adress field with zero
    memcpy(current_remote_addr, 0, BT_ADDR_LEN);
    // init last remote adress field with zero
    memcpy(last_remote, 0, BT_ADDR_LEN);


    // initialize external interrupt register and
    // io ports, disable external interrupt 7
    cbi (EIMSK, INT7);
    // set external interrupt 7 to trigger on a
    // rising edge
    sbi (EICRB, ISC71);
    sbi (EICRB, ISC70);
    // enalbe external interrupt 7
    sbi (EIMSK, INT7);

    // initialize IO Port 7
```

```
// set PORTE7 as input
cbi(DDRE, 7);
cbi(PORTE, 7);

// print my onw adress
btn_bt_get_local_bd_addr( local_bd_addr );
printf( "local BD_ADDR: " );
btn_print_bd_addr( local_bd_addr );
printf( "." NL NL);

printf("welcome ! ready for action !" NL NL);

// register function "cb" for event "event". when "
   event"
// occurs, the callback function "cb" is called with
// "cb_data" as its second argument
//               event              cb     cb_data
btn_disp_ev_reg(TIMEOUT_EV, timeout_cb, TIMEOUT_EV);
btn_disp_ev_reg( BT_CONNECTION_EV, conn_cb,
        BT_CONNECTION_EV );
btn_disp_ev_reg( DEBUG_UART_RCV_EV, debug_input_cb,
        DEBUG_UART_RCV_EV );
btn_disp_ev_reg( BT_DATA_RCV_EV, data_received_cb,
        BT_DATA_RCV_EV );

// register timoutevent for synchronizing
// wait a bit for the first synchronization to take
   place
btn_timeout_reg(TIMEOUT_EV, 30000, 0);


// initialize fix time bounds
bigint tmp;
s16tobigint(25000, &tmp);
addbigint(&t100000ms, &tmp);
addbigint(&t100000ms, &tmp);
addbigint(&t100000ms, &tmp);
addbigint(&t100000ms, &tmp); // has value 100000 now

addbigint(&t200000ms, &t100000ms);
addbigint(&t200000ms, &t100000ms);
// has value 200000 now

addbigint(&t5min[0], &t200000ms);
addbigint(&t5min[0], &t100000ms);
// has value 300000 now == 5min

// initialize 5 min array
for (i = 1; i < 36; i++) {
```

```
        copybigint(&t5min[i - 1], &t5min[i]);
        addbigint(&t5min[i], &t5min[0]);
    }

    copybigint(&t5min[11], &t1h);
    display_prompt();
    // run dispatcher
    btn_disp_run();
    return 0;
}
```

A.2 Ganzzahl-Bibliothek

A.2.1 Bibliothek - bigint.c

Das Headerfile für die Ganzzahlbibliothek:

```
/*
 * bigint.c
 *
 * 2003.10.10 Boris Zweimueller <zboris@student.ethz.ch>
 *
 * This file is free software; you can redistribute it and
 * /or modify it under the terms of the GNU General Public
 * License as published by the Free Software Foundation;
 * either version 2 of the License, or (at your option) any
 * later version.
 *
 * This program is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even the
 * implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.  See the GNU General Public License
 * for more details.
 *
 * This is the implementation of the big-integer library
 * for BTnodes. For implementation details please read
 * the function comments.
 *
 * (c) ETH Zurich
 */

/* includes */
#include "bigint.h"

/*-------------------------------------------------------
 * Function to generate a new bigint from a signed 16 bit
 * value. The unused numbers are initialized with 0.
 *-------------------------------------------------------
 * */
void s16tobigint(s16 number, bigint * big) {
```

```
        u8 i = 0;
        while (number >= 128) {
            (*big).data[i] = (number & 0x7F); // modulo
            number = number >> 7; // division
            i++;
        }
        (*big).data[i++] = number;
        while (i < BIG_INT_SIZE) {
            (*big).data[i++] = 0;
        }
}

/*-------------------------------------------------------------
 * Function to check wether a bigint has value 0
 *-------------------------------------------------------------
 * */
u8 bigintiszero(bigint * big) {
    u8 i;
    u8 res = TRUE;
    for (i = 0; i < BIG_INT_SIZE; i++) {
        if ((*big).data[i] != 0) {
            res = FALSE;
            break;
        }
    }
    return res;
}

/*-------------------------------------------------------------
 * Returns true if big1 is bigger than big2
 *-------------------------------------------------------------
 * */
u8 bigintbigger(bigint * big1, bigint * big2) {
    u8 i;
    for (i = BIG_INT_SIZE - 1; i != 0; i--) {
        if ((*big1).data[i] != (*big2).data[i]) {
            break;
        }
    }
    if ((*big1).data[i] > (*big2).data[i])
        return 1;
    else
        return 0;
}

/*-------------------------------------------------------------
 * Returns true if (big1 == big2)
 *-------------------------------------------------------------
 * */
```

```
u8 bigintequals(bigint * big1, bigint * big2) {
    u8 res = TRUE;
    u8 i;
    for (i = 0; i < BIG_INT_SIZE; i++) {
        if ((*big1).data[i] != (*big2).data[i]) {
            res = FALSE;
        }
    }
    return res;
}

/*-------------------------------------------------------------
 * Sets the passed bigint to the maximal value
 *-------------------------------------------------------------
 * */
void setmaxbigint(bigint * big) {
    u8 i;
    for (i = 0; i < BIG_INT_SIZE; i++) {
        (*big).data[i] = 0x7F; // 127
    }
}

/*-------------------------------------------------------------
 * Returns true if the passed bigint has maximal value
 *-------------------------------------------------------------
 * */
u8 ismaxbigint(bigint * big) {
    u8 i;
    u8 res = TRUE;
    for (i = 0; i < BIG_INT_SIZE; i++) {
        res = res && ((*big).data[i] == 0x7F);

    }
    return res;
}

/*-------------------------------------------------------------
 * Function to subtract two bigints. The result is in
 * big1 afterwards !
 *
 * Precondition: big1 > big2 !!
 *-------------------------------------------------------------
 * */
void subbigint(bigint * big1, bigint * big2) {
    u8 carry = 0;
    u8 i;
    u8 tmp;

    for (i = 0; i < BIG_INT_SIZE; i++) {
```

```
        tmp = (*big1).data[i] - ((*big2).data[i] + carry);
        (*big1).data[i] = (tmp & 0x7F);
        carry = (tmp >> 7); // new carry
    }

}

/*-------------------------------------------------------------
 * Function to add two bigints.
 * The result is in big1 afterwards !
 *-------------------------------------------------------------
 * */
void addbigint(bigint * big1, bigint * big2) {
    u8 carry = 0;
    u8 i;
    s16 tmp;

    if ( ismaxbigint(big1) || ismaxbigint(big2)) {
        setmaxbigint(big1);
        return;
    }
    for (i = 0; i < BIG_INT_SIZE; i++) {
        tmp = (*big1).data[i] + (*big2).data[i] + carry;
        (*big1).data[i] = (u8)(tmp & 0x7F);
        carry = (u8)(tmp >> 7); // new carry
    }
}

/*-------------------------------------------------------------
 * Function to add a signed 16bit value to a bigint
 *-------------------------------------------------------------
 * */
void addS16bigint(bigint * big, s16 s) {
    bigint tmp;
    s16tobigint(s, &tmp);
    addbigint(big, &tmp);
    // result in bigint big
}

/*-------------------------------------------------------------
 * Function to make a simple division of a bigint
 * val is a value 0 >= val <= 127
 * remainder has the remainig value of the int division
 *-------------------------------------------------------------
 * */
void simpledivbigint(bigint * big, u8 val,
        u8 * remainder) {
    u8 carry = 0;
    s16 i;
```

```
    s16 tmp;
    for (i = BIG_INT_SIZE - 1; i >= 0; i--) {
        tmp = (*big).data[i] + 128*carry;
        carry = tmp % val;
        (*big).data[i] = tmp / val;
    }
    *remainder = carry;
}

/*-------------------------------------------------------------
 * Function to multiply to bigints, solution is in "sol"
 *-------------------------------------------------------------
 * */
void multiplybigint(bigint * big1, bigint * big2,
        bigint * sol) {
    u8 s[2 * BIG_INT_SIZE + 1];
    u8 j, i;
    u8 carry = 0;
    s16 temp = 0;

    // initialize solution with 0
    for (i = 0; i < BIG_INT_SIZE; i++) {
        s[i] = 0;
    }

    for (j = 0; j < BIG_INT_SIZE; j++) {
        carry = 0;
        for (i = 0; i < BIG_INT_SIZE; i++) {
            temp = (s16)(s[i + j]) + (s16)((*big1).data[i])
                * (s16)((*big2).data[j]) + (s16)carry;
            carry = (u8)(temp >> 7);
            s[i + j] = (u8)(temp % 128);
        }
        s[i + j + 1] = carry;
    }

    // copy solution to bigint 'sol'
    for (i = 0; i < BIG_INT_SIZE; i++) {
        (*sol).data[i] = s[i];
    }
}

/*-------------------------------------------------------------
 * Function to copy the value of a bigint.
 * big2 has the same value as big1 afterwards.
 *-------------------------------------------------------------
 * */
void copybigint(bigint * src, bigint * dest) {
    memcpy(dest, src, sizeof(bigint));
```

```
    dest->pos = src->pos;
}

/*-------------------------------------------------------------
 * Function to print the single numbers of a bigint
 * in base 128
 *-------------------------------------------------------------
 * */
void print(bigint * big) {
    u8 i = 0;
    for (i = 0;i < BIG_INT_SIZE; i++) {
        printf("%d ", (*big).data[i]);
    }
    printf(NL);
}

/*-------------------------------------------------------------
 * Function to print a decimal representation of a bigint
 *-------------------------------------------------------------
 * */
void printbigint(bigint * big) {
    bigint copy; // copy to not destroy argument big !!
    u8 digit = 0;
    u8 number[50]; // the result numbers
    u8 rem = 0;

    copybigint(big, &copy);

    while (!(bigintiszero(&copy))) {
        simpledivbigint(&copy, 10, &rem);
        number[digit++] = rem;
    }
    while (digit > 0) {
        digit--;
        printf("%d", number[digit]);
    }
    //printf(NL);
}

/*-------------------------------------------------------------
 * makes a special division to calculate the drift of the
 * internal clock
 *
 * ppm = parts per million,
 * eg. ppm = 100 ==> 1ms per 10 seconds
 *
 * here we use ppm = 100.
 *
 * therefore time' = time / (1 + ppm) resp.
```

```
 * time' = 1/(1 + ppm) * time,
 *
 * with 1/(1 + ppm) = 0.9999
 *
 * direction is used to calculate the maximum drift for a
 * clock that is: slower (dir = 0) or faster (dir = 1)
 *
 * XXX not accurate for times bigger than 10^9 because
 * 1/1.001 = 0.999900009999 easy to fix if really neces-
 * sary. Uses much more time to calculate then.
 *-----------------------------------------------------------
 * */

void driftdivision(bigint * big, bool direction) {
    bigint tmp;
    bigint mp;
    u8 rem;

    // make copy
    copybigint(big, &tmp);

    // set bigint mp to the corresponding quotient

    // multiply by 1/(1 + ppm) * 10^4

    // for ppm 100
    // s16tobigint(9999, &mp);

    // for ppm 4000
    s16tobigint(9960, &mp);

    multiplybigint(big, &mp, big);
    // divide
    simpledivbigint(big, 100, &rem);
    simpledivbigint(big, 100, &rem);
    if (direction) {
        // drift "up"
        copybigint(&tmp, &mp);
        subbigint(&tmp, big);
        addbigint(&mp, &tmp);
        copybigint(&mp, big);
    }
    // else we want the drift downwards which
    // is already stored in big.
}
```

A.2.2 Bibliothek - bigint.h

Die Implementierung der Ganzzahlbibliothek:

```
/*
 * bigint.h
 *
 * 2003.10.10 Boris Zweimueller <zboris@student.ethz.ch>
 *
 * This file is free software; you can redistribute it and
 * /or modify it under the terms of the GNU General Public
 * License as published by the Free Software Foundation;
 * either version 2 of the License, or (at your option) any
 * later version.
 *
 * This program is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even the
 * implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE. See the GNU General Public License
 * for more details.
 *
 * This is the header file for the big-integer library on
 * a btnode.
 *
 * (c) ETH Zurich
 */

#include <drivers.h>

#ifndef _BIGINT_H_INCLUDED_
#define _BIGINT_H_INCLUDED_

// length of bigints in bytes
#define BIG_INT_SIZE 10

// the bigint datatype which represents large integers
typedef struct bi {
    u8 data[BIG_INT_SIZE];
    u8 pos;
}bigint;

// input a bigint
void setmaxbigint(bigint* big);
void s16tobigint(short s, bigint* big);

// comparison
bool bigintiszero(bigint* big);
bool bigintequals(bigint* big1, bigint * big2);
bool bigintbigger(bigint* big1, bigint * big2);
bool ismaxbigint(bigint* big);

// operations
void subbigint(bigint* big1, bigint* big2);
```

```
void addbigint(bigint* big1, bigint* big2);
void addS16bigint(bigint* big, short s);
void simpledivbigint(bigint* big, u8 val, u8* remainder);
void multiplybigint(bigint* big1, bigint* big2, bigint* sol
    );
void driftdivision(bigint* big, bool direction);

// handling
void copybigint(bigint* src, bigint* dest);

// output bigint
void printbigint(bigint* big);
void print(bigint* big);

#endif
```

A.3 Änderungen am Betriebssystem

Die Änderungen der Echzeituhr-Implementierung in der Systemsoftware: Das File `src/avr128/avr128_rtc.c`

```
/*
 * avr128_rtc.c - device driver for the real-time clock
 *
 * 2001.01.09 Lukas Karrer <lkarrer@trash.net>
 * 2002.05.23 Oliver Kasten <oliver.kasten@inf.ethz.ch>
 * 2003.09.26 Martin Hinz <hinz@tik.ee.ethz.ch>
 *
 * This file is free software; you can redistribute it and/
 * or modify it under the terms of the GNU General Public
 * License as published by the Free Software Foundation;
 * either version 2 of the License, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even the
 * implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE.  See the GNU General Public License
 * for more details.
 *
 * (c) 2000-2003 ETH Zurich
 */

/*---------------------------------------------------------
 * BUGS
 *---------------------------------------------------------*/

// port_or is not defined in iomacro.h (which gets included
// in io.h) while __port_or has a definition. This seems to
```

```
// be a bug in iomacro.
// #define port_or __port_or
// ditto
// #define port_and __port_and


/*-------------------------------------------------------------
 * includes
 *-------------------------------------------------------------*/
#include <btnode.h>
#include <bigint.h>

#include "rtc.h"
#include "led.h"

#if defined(USE_DISPATCHER)
#include <dispatcher.h>
// dont need timeouts unless we use the dispatcher...
#include "timeouts.h"
extern timeout_entry_t timeout_table[TIMEOUT_TABLE_SIZE];
#endif // USE_DISPATCHER


// current time in 1/1024 seconds
static volatile u32 mseconds;


//BZ-------------------------------------------------------------
bigint bigtime;
bigint updatevalue;
//BZ END---------------------------------------------------------


// var used to count if we should flash LED or not
static u8 rtc_flash;

u8 heartbeat_on=1;

/*-------------------------------------------------------------
 * btn_rtc_init() - initialize rtc
 *-------------------------------------------------------------*/

/*
 * PRE: assumes an external 32.768 kHz Xtal, may not be
 * called twice global Interrupt must be enabled
 * POST: starts counting time in mseconds -> interrupts
 * every 250msec
 */
```

```
void btn_rtc_init() {

    // set time to zero
    mseconds = 0;


    //BZ-----------------------------------------------
    // initialize bigint counter with zero
    s16tobigint(0, &bigtime);
    // for the add to be faster
    s16tobigint(250, &updatevalue);
    //BZ END-------------------------------------------


    // set up timer 0 to use external Xtal with
    // prescaling of 32

    // Note: the following sequence is esential!!
    // see "Async Operation of Timer 0" in ATmega 103
        Manual

    sbi (ASSR, AS0); // switch to external clock on timer 0
    outp (0x00, TCNT0); // set timer 0 to zero
    // wait until TCNT0 is set
    while( ! ( inp(ASSR)  & (1 << TCN0UB)) );

    // Stop Timer0
    port_and (TCCR0, 0xF8);
    // wait until TCCR is set
    while( ! ( inp(ASSR)  & (1 << TCR0UB)) );

    // clear interrupt flag
    sbi( TIFR, TOV0 );

    // set timer/counter0 prescaling factor:
    // value overflow period      value overflow period
    // 0     stopped  never       4     1/2 s   500 ms
    // 1     1/64 s   15.625 ms   5     1   s  1000 ms
    // 2     1/8  s   125 ms      6     2   s  2000 ms
    // 3     1/4  s   250 ms      7     8   s  8000 ms
    // set prescaler of T/C0
    port_or (TCCR0, 0x03); // Start Timer0, prescaling of
        32
    // wait until TCCR is set
    while( ! ( inp(ASSR)  & (1 << TCR0UB)) );

    // enable interrupt on Timer0 overflow
    sbi( TIMSK, TOIE0 );
}
```

```
/*-------------------------------------------------------------
 * btn_rtc_set( u32 msec) - set time to msec
 *-------------------------------------------------------------*/

// PRE: assumes an external 32.768 kHz Xtal POST: sets
// time to msec and restarts Timer0

void btn_rtc_set( u32 newmsec) {

  // Note: the following suquence is esential!! see
  // "Async Operation of Timer 0" in ATmega 103 Manual

  cbi( TIMSK, TOIE0 ); // disable interrupts on Timer0
  // disable interrupts on Timer 0 overflow
  cbi( TIMSK, OCIE0 );

  port_and( TCCR0, 0xF8 ); // Stop Timer0
  // wait until TCCR is set
  while( ! ( inp(ASSR)  & (1 << TCR0UB)) );

  sbi( TIFR, TOV0 ); // clear interrupt flag

  outp( 0x00, TCNT0 ); // set timer 0 to zero
  // wait until TCNT0 is set
  while( ! ( inp(ASSR)  & (1 << TCN0UB)) );

  port_or( TCCR0, 0x03 ); // Start Timer0, prescaling of 32
  // wait until TCCR is set
  while( ! ( inp(ASSR)  & (1 << TCR0UB)) );

  mseconds = newmsec; // set time

  sbi( TIMSK, TOIE0 ); // enable interrupt on overflow
}

/*-------------------------------------------------------------
 * un32 btn_rtc_get() - get time in mseconds
 *-------------------------------------------------------------*/

// PRE: assumes an external 32.768 kHz Xtal POST: returns
// time in true milliseconds

inline u32 btn_rtc_get() {

  // RATIONAL: return msseconds-counter plus the number of
  // ms that have passed since the msseconds-counter has
```

```
  // been updated last. every clock tick (of timer/counter0
      )
  // is 1024th of a second (the external 32kHz clock is
  // prescaled by a factor of 32 resulting in a 1024 Hz
  // clock). to get the number of ms that have passed
  // since the msseconds-counter has been updated last, the
  // TC/0 value needs to be divided by 256 (results in the
  // fraction of a second that has passed) multiplied by
  // 1000ms (to get the value in ms).
  // * 125 >> 7

  // return ( mseconds + ((inp(TCNT0) * 1000) >> 10) );

  return (mseconds + (((u16)inp(TCNT0) * (u16)125) >> 7));
}

//BZ----------------------------------------------------------
inline void btn_rtc_get_64(bigint * big, u8 * offset) {
    *offset = (u8)( ( (u16)inp(TCNT0) * (u16)125) >> 7);
    copybigint(&bigtime, big);
}


/*-----------------------------------------------------------
 * u8 btn_rtc_get_counter() returns the actual counter
 * value the function is inlined and therefore does not
 * cause any PAF activation
 *
 * this is done in a function, because most of the library
 * files are not compiled with avr-gcc but we need fast
 * access to the counter for delay calculations
 *-----------------------------------------------------------
 * */
inline u8 btn_rtc_get_counter() {
    return inp(TCNT0);
}

//BZ END------------------------------------------------------

/*-----------------------------------------------------------
 * SIG_OVERFLOW0 - interrupt handler for pseudo RTC
 *
 * PRE:  Timer 0 overflows, interrupts must be enabled
 * POST: mseconds is increased by 256, flash PB0
 *-----------------------------------------------------------*/
SIGNAL(SIG_OVERFLOW0) {

  u8 i;
  u32 now;
```

```
  //BZ----------------------------------
  u8 carry = 0;
  u16 tmp;
  //BZ----------------------------------

  // we add 250 here, since we want to count true
  // milliseconds
  mseconds += 250;

  //BZ----------------------------------------------------------
  //add 250 to 64bit variable
  for (i = 0; i < BIG_INT_SIZE; i++) {
      tmp = bigtime.data[i] + updatevalue.data[i] + carry;
      bigtime.data[i] = (u8)(tmp & 0x7F);
      carry = (u8)(tmp >> 7); // new carry
  }
  //BZ END------------------------------------------------------

#if defined(USE_DISPATCHER)
  // OLLI: throw out the stupid time_click and replace it
  // btn_disp_put_event((u8)RTC_TIME_CLICK,(call_data_t)0);

  now = btn_rtc_get();

  // there are two possible optimizations:

  // 1) we could check whether (|current_time - timeout_
  // expiry| < 250/2 ), and then trigger the timeout
  // callback. this way the callback is triggered closer
  // to its actual expiry time.

  // 2) more sophisticated: check whether (cuerrent_time <
  // timeout_expiry < cuerrent_time+250) (the timeout
  // expires somewhere in between this interrupt and the
  // next interrupt). if so, set the timer0 compare
  // register so that timer0 generates a compare match at
  // the actual timeout expiry time. enable timer0
  // compare interrupts. in the timer0 compare interrupt,
  // trigger the callbyck function.


  for( i=0; i < TIMEOUT_TABLE_SIZE; i++ ) {

    if( timeout_table[i].expire <= now ) {
      // timeout i has expired

      btn_disp_put_event( timeout_table[i].event,
                          timeout_table[i].call_data );
```

```
      timeout_table[i].expire = RTC_MAX_TIME_VALUE;
      timeout_table[i].call_data = 0;
    }
  }

#endif // USE_DISPATCHER

  if (heartbeat_on) {
              // flash the heartbeat signal (LED0)
              rtc_flash++;

              if( rtc_flash%4 == 0 ) btn_led_set(0);
              else                   btn_led_clear(0);
  }

  CHECK_SP;
}

/*-------------------------------------------------------------
 * btn_heartbeat(u8 state)
 *-------------------------------------------------------------*/
void btn_heartbeat(u8 state) {
        heartbeat_on = state;
        if (!heartbeat_on) btn_led_clear(0);
}

/*-------------------------------------------------------------
 * btn_delay_ms()
 *-------------------------------------------------------------*/
// busy-wait of msec milliseconds (if function is not
// interrupted)
// NOTE: 3.6864 MHz XTal is assumed

void btn_delay_ms( u32 msec ) {
    u32 i,j;

    for( i=0; i < msec; i++ ) {
      for( j=0; j<919; j++ ) {
        asm volatile ("nop");
      }
    }
}

/*-------------------------------------------------------------
 * btn_delay_10us()
 *-------------------------------------------------------------*/
// busy-wait for usec * 10 milliseconds (if function is not
// interrupted) NOTE: 3.6864 MHz XTal is assumed NOTE:
// 10us = 36.864 instructions.. this function should be
```

```
// optimized in ML. XXX olli: what for? to run faster?! :P

void btn_delay_10us( u32 usec ) {
  u32 i;
  for (i=0; i < (4 * usec) ;i++){
    asm volatile ("nop");
  }
}
```

A.4 PC-Tool

Die Implementierung des PC-Tools der Testumgebung:

```
/*
 * sendint.c
 *
 * 2004.01.10 Boris Zweimueller <zboris@student.ethz.ch>
 *
 * This file is free software; you can redistribute it and
 * /or modify it under the terms of the GNU General Public
 * License as published by the Free Software Foundation;
 * either version 2 of the License, or (at your option) any
 * later version.
 *
 * This program is distributed in the hope that it will be
 * useful, but WITHOUT ANY WARRANTY; without even the
 * implied warranty of MERCHANTABILITY or FITNESS FOR A
 * PARTICULAR PURPOSE. See the GNU General Public License
 * for more details.
 *
 * This is used to test the implemented clock synchronisa-
 * tion algorithm implemented in synchro.c.
 * It iterates for a given time and supplies the connected
 * BTnode (serial interface) with correct time.
 * In addition it uses the parallelport to signal
 * interrupts to the connected BTnodes.
 *
 * (c) ETH Zurich
 */

/*---------------------------------------------------------
 * Includes
 *---------------------------------------------------------
 * */
#include <stdio.h>
#include <unistd.h>
#include <asm/io.h>

/*---------------------------------------------------------
```

```
 * Defines
 *-----------------------------------------------------------
 * */

// the adress of the parallelport used for sending
// interrupt to btnodes
#define BASEPORT 0x378

/*-----------------------------------------------------------
 * void sendInt()
 *
 * This function sets the corresponding data bit in the
 * parallelport to logical one. An interrupt is then
 * initiated on the connected btnodes
 *-----------------------------------------------------------
 * */
void sendInt() {
    // Get access to the parallel port
    if (ioperm(BASEPORT, 1, 1)) {
        perror("ERROR: ioperm, could not get permission. \
                Did you start the program as root?\n");
        exit(1);
    }

    // set databit D0 - D3 to 1
    outb(15, BASEPORT);

    // Sleep for a while (200 ms)
    usleep(200000);

    // set data bit to 0
    outb(0, BASEPORT);

    // and release access rights to parallelport
    if (ioperm(BASEPORT, 1, 0)) {
        perror("ERROR: ioperm during releasing port");
        exit(1);
    }
}

/*-----------------------------------------------------------
 * The man procedure contains the loop for the test
 * - periodically generate interrupts
 * - periodically supply anker node with correct time
 *-----------------------------------------------------------
 * */
int main()
{
    int count = 0; // count #minutes the test is running
```

```
    int mult = 1; // how many times we need to shiftup
    int i = 0;
    char call[100];

    system("echo -e -n 'setbounds 20000:20100\r' \
           >/dev/ttyS0");
    // some sleep to not make the interrupt come faster

    while (1) {
        sleep(5);
        sendInt();
        count++;
        printf("sent interrupt %d times.\n", count);
        // terminate after 3 hours
        if (count == 180) {
            exit(1);
        }
        sleep(55);
        // set current correct bounds
        // every ten minutes
        if (count % 10 == 0) {
            printf("%d * 60 seconds passed -> \
                   shift up bounds\n");
            // initial setting
            system("echo -e -n 'setbounds \
                   20000:20100\r' > /dev/ttyS0");
            // create call
            sprintf(call, "echo -e -n 'shiftup5m \
                   %d\r' > /dev/ttyS0", mult);
            // execute call -> echo to terminal
            system(call);
            // increase shiftup counter
            // 5 min per shiftup call => twice in 10min
            mult = mult + 2;
        }
    }

    return 0;
}
```