AG VERNETZTE SYSTEME
FACHBEREICH INFORMATIK
AN DER TECHNISCHEN UNIVERSITÄT
KAISERSLAUTERN

Diplomarbeit

Entwicklung eines leichtgewichtigen Kommunikationsframeworks auf einem Mikrocontroller mit SDL

Marc Krämer

28. September 2005

Entwicklung eines leichtgewichtigen Kommunikationsframeworks auf einem Mikrocontroller mit SDL

Diplomarbeit

Arbeitsgruppe Vernetzte Systeme
Fachbereich Informatik
Technische Universität Kaiserslautern

Marc Krämer

Tag der Ausgabe : 09. Mai 2005
Tag der Abgabe : 28. September 2005

Betreuer : Dipl. Inf. Thomas Kuhn
Referent : Prof. Dr. Reinhard Gotzhein

Ich erkläre hiermit, die vorliegende Diplomarbeit selbständig verfaßt zu haben. Die verwendeten Quellen und Hilfsmittel sind im Text kenntlich gemacht und im Literaturverzeichnis vollständig aufgeführt.

Kaiserslautern, den 28. September 2005

(Marc Krämer)

Inhaltsverzeichnis

Kapitel 1

Einleitung

Mikrocontroller sind heute nahezu allgegenwärtig. Sie sind in Haushaltsgeräte, Telefone sowie Unterhaltungselektronik integriert. Der Einsatz wird in der Zukunft wohl noch steigen, wenn Projekte wie das AMI-Projekt (*Ambient Intelligence*) [AMI] dafür sorgen, daß der Mensch in allen Lebenslagen unterstützt wird. Doch die Entwicklung von Anwendungen für Mikrocontroller ist meist mit hohen Kosten verbunden. Typischerweise werden Mikrocontroller in Assembler oder C programmiert. Die Gründe dafür liegen in den begrenzten Ressourcen des Mikrocontrollers. Hierunter fallen hauptsächlich Arbeitsspeicher und Prozessorleistung. In der Anwendungsentwicklung für moderne PCs werden objektorientierte Programmiertechniken eingesetzt, die den Entwicklungsprozeß beschleunigen. Die Firma CrossBow [XBO] hat die Hardwareplattform MICAz [MIC] entwickelt. Hierbei handelt es sich um einen Mikrocontroller sowie eine ZigBee-kompatible Funkschnittstelle [Zig]. Zur Anwendungsentwicklung des MICAz steht TinyOS [Tin] zur Verfügung.

In der vorliegenden Arbeit soll nun der Versuch unternommen werden, SDL als Ausgangspunkt für die effiziente Programmierung eines Mikrocontrollers zu verwenden. Als Entwicklungsumgebung kommt die SDL-Suite von Telelogic [Tel] zum Einsatz. Diese besitzt bereits Funktionen, um C-Code für Mikrocontroller zu erzeugen, muß jedoch noch auf eine konkrete Hardwareplattform angepaßt werden. Anschließend werden Messungen durchgeführt, die den Overhead und somit die Effizienz durch die Verwendung der formalen Spezifikationssprache SDL (*Specification and Description Language*) zeigen. Da die entwickelte Plattform zur Implementierung und zum Testen von Übertragungsprotokollen eingesetzt werden soll, wurde als Abschluß ein einfaches MAC-Protokoll implementiert.

Zunächst wird eine Einführung in die verwendete Hard- und Software gegeben. Kapitel 3 beschreibt die entwickelte Laufzeitumgebung. Meßwerte zeigen den Einfluß von SDL und der darunterliegenden Laufzeitumgebung auf die Ausführungsgeschwindigkeit eines Programms. Da für mobile Knoten die Energieversorgung ein entscheidendes Kriterium ist, wird diese getrennt in Kapitel 4.2 betrachtet. In Kapitel 5 folgt eine Beispielimplementierung einer MAC-Schicht (*Media Access Control*) für die Funkschnittstelle. Sie zeigt, daß die Programmierung des Mikrocontrollers sich nun einfacher gestaltet. Als letztes folgt eine Abschlußbewertung, welche die vorliegende Arbeit noch einmal zusammenfaßt und noch ausstehende Tätigkeiten nennt.

In den Anhängen sind zusätzlich Hilfestellungen zur Lösung bekannter Probleme sowie die kompletten Daten aus den Laufzeitmessungen angegeben.

Einige Literaturquellen sind nur online verfügbar. Aus diesem Grund liegt dieser Arbeit eine CD bei, die den Stand der Quellen zum Zeitpunkt der Abgabe widerspiegelt. Jede Quelle findet sich hierbei in dem der Referenz entsprechenden Verzeichnis. Zusätzlich ist die aktuelle Laufzeitumgebung im Verzeichnis `Code` und ein Beispiel-SDL-System im Verzeichnis `SDL` beigefügt.

Abschließend möchte ich mich für die Ausgabe des Themas bei Professor Dr. Gotzhein und die Betreuung durch Thomas Kuhn bedanken. Für Kritik und Hilfestellung danke ich den Mitarbeitern Ingmar Fliege, Alexander Geraldy und Christian Webel. Mein Dank für Korrekturen gilt außerdem Alexandra und Thorsten Michels, Karl-Christian Pammer sowie Jan Schäfer. Ganz besonders möchte ich mich auch bei meinen Eltern bedanken, die mir das Studium an der Universität ermöglicht haben.

Kapitel 2

Grundlagen

In diesem Kapitel werden die wichtigsten Grundlagen, die zum Verständnis der Arbeit notwendig sind, dargestellt. Dazu gehört die kurze Vorstellung der verwendeten Hard- und Software sowie die später verwendete Nomenklatur.

2.1 Nomenklatur

Für eine bessere Lesbarkeit sind im Fließtext zur Hervorhebung verschiedene Formatierungen verwendet worden:

- Eigennamen, vor allem englische Begriffe, sind wie *Start of Frame Delimiter* gesetzt.
- Quelltext wurde im Fließtext als `a=a+1;` gesetzt.
- Funktionsaufrufe von C-Funktionen sind als *void Set_Lamps(int val)* gesetzt.
- Funktionsparameter werden zur Unterscheidung im Text als **val** gesetzt.
- Schlüsselwörter, z. B. aus TinyOS sind als **true** gesetzt.
- SDL-Signale sind als SET_LAMP_RED gesetzt.
- Dateinamen sind im Text als `SDLM.nc` gesetzt.
- Befehle der Kommandozeile sind als `make clean install` gesetzt.

Alle in dieser Arbeit verwendeten Abkürzungen sind im Abkürzungsverzeichnis im Anhang I auf Seite 83 enthalten.

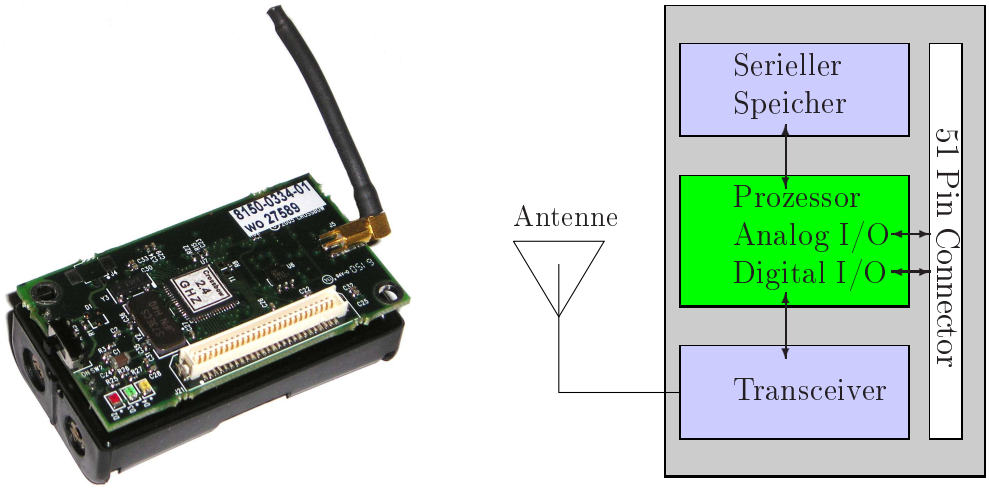

Abbildung 2.1: Bild und Struktur des MICAz Knotens

2.2 Hardware

Bei der verwendeten Hardwareplattform handelt es sich um eine Platine, die von CrossBow [XBO] entwickelt und bestückt wurde. Diese Platine wird als MICAz [MIC] bezeichnet und besteht im wesentlichen aus sechs Hardwarekomponenten:

- ATMEL ATMega 128L, als Hauptprozessor [ATMa]
- Chipcon CC2420, als Sender und Empfänger (*Transceiver*) [CC204]
- Dallas DS2401, als eindeutige Seriennummer [DS2]
- ATMEL AT45DB041B, als serieller Speicher [ATMb]
- drei Leuchtdioden zur Anzeige
- Batterien

Der Aufbau der Hardwarekomponenten des MICAZ und ein Bild ist in Abbildung 2.1 dargestellt. Im folgenden wird jede dieser Hardwarekomponenten genauer beschrieben. Als wichtigste Komponenten sind der Hauptprozessor und der Transceiver zu nennen. Die eindeutige Seriennummer hat eine untergeordnete Bedeutung, und die serielle RAM-Erweiterung wurde in dieser Arbeit nicht benutzt, soll der Vollständigkeit wegen aber erwähnt werden. Die drei eingebauten Leuchtdioden können zur Zustandsanzeige des Knotens verwendet werden. Der letzte Punkt, die Batterien, wird betrachtet, um eine Abschätzung zur Laufzeit eines MICAz-Knotens zu bekommen.

2.2.1 ATMEL ATMega 128L

Der Hauptprozessor des MICAz ist ein ATMega 128L, der mit 4 KByte RAM, 128 KByte ROM sowie 4 KByte EEPROM, das als permanenter Datenspeicher die-

nen kann, ausgestattet ist und bei einer Frequenz von 7,3728 MHz durch einen Quarz-Oszillator getaktet wird. Intern existieren eine *Real Time Clock* (RTC) und vier Timer, zwei davon mit 16-Bit-, zwei mit 8-Bit-Auflösung. Außerdem enthält er zwei serielle UART-Schnittstellen (*Universal Asynchronous Receiver Transmitter*), die eine maximale Übertragungsrate von 256 kbps haben. Der Mikrocontroller sollte mit einer Spannung zwischen 2,7 V und 5,5 V betrieben werden. Bei einer Spannung von 3 V beträgt die Stromaufnahme bei Last 7,5 mA und ohne Last 6,5 mA.

2.2.2 Chipcon CC2420

Der *Transceiver*-Chip von Chipcon arbeitet im 2,4 MHz Band und ist nach dem ZigBee-Standard [Zig] entworfen und verifiziert worden. Die maximale Übertragungsrate beträgt 250 kbps. Er besitzt einen Empfangs- und einen Sendepuffer, die maximal ein Paket mit 128 Byte speichern können. Ein solches Paket kann 121 Byte Nutzdaten beinhalten. Jedes dieser Pakete ist mit einer CRC-16-Prüfsumme (*Cyclic Redundancy Check*) gesichert, die der Chip selbständig berechnen und verifizieren kann. Dadurch wird im angeschlossenen Mikrocontroller keine weitere Rechenarbeit benötigt. Ein *Clear Channel Assessment* sorgt dafür, daß ein Sendewunsch nur bei freiem Medium ausgeführt wird, sofern dies gewünscht ist. Weiterhin besteht die Möglichkeit, die eingebaute *Encryption Unit* zu verwenden, um die zu sendenden Daten zusätzlich zu verschlüsseln.

Der Transceiver-Chip ist durch folgende Zeitschranken charakterisiert: das Einschalten des Oszillators dauert 860 μs und ein PLL-Lock (*Phase-Locked-Loop*) dauert 192 μs. Ein PLL-Lock wird beim Einschalten, einem Kanalwechsel oder auch dem Umschalten vom Empfangs- (RX) in den Sendemodus (TX) sowie umgekehrt nötig. Es sorgt dafür, daß der Oszillator auf der richtigen Frequenz und Phase sendet.

Der Transceiver kann in einem Spannungsbereich von 2,1 V bis 3,6 V arbeiten. Bei der Stromaufnahme sind die Betriebsmodi *Power Down* (PD), *Idle*, RX und TX zu unterscheiden. Im PD-Modus ist der Oszillator ausgeschaltet, dadurch ergibt sich auch die minimale Stromaufnahme von 20 μA. Beim Wechsel aus dem PD-Modus in den RX- oder TX-Modus werden allerdings $192\,\mu s + 860\,\mu s = 1.052\,\mu s$ benötigt. Sobald der Oszillator eingeschaltet wird, befindet sich der Chip im *Idle*-Modus. Erst in diesem Modus können die FIFO-Puffer (*First In First Out*) geschrieben bzw. gelesen werden. Der Stromverbrauch steigt hier auf 426 μA, und die Einschaltzeit beschränkt sich nun lediglich auf die 192 μs für das PLL-Lock. Sobald der Chip seine eigentliche Aufgabe, das Senden oder Empfangen, aufnimmt, steigt der Stromverbrauch bis auf 17,4 mA bzw. 19,7 mA für den Empfang. Durch Regulieren der Sendeleistung kann der Stromverbrauch für das Senden noch etwas verringert werden, nicht jedoch für den Empfang. Für möglichst hohe Laufzeiten des MICAz ist darauf zu achten, diesen Chip so oft wie möglich vom Sende- oder Empfangsbetrieb wenigstens zurück in den Idle-Modus zu schalten. Eine exemplarische Rechnung der Laufzeit eines Knotens wird in Abschnitt 4.2 auf Seite 39 durchgeführt.

Anzumerken ist an dieser Stelle die Besonderheit, daß der Transceiver nach einer Sendeoperation nicht wieder in den vorherigen Zustand, sondern automatisch zurück in den Empfangsmodus wechselt. Dies ist beim Timing und beim Stromverbrauch gesondert zu beachten.

2.2.3 Dallas DS2401

Eine eindeutige Seriennummer wird für den MICAz über den Dallas DS2401 zur Verfügung gestellt. Hierbei handelt es sich um ein rein passives Bauteil, das über einen *Pullup*-Widerstand seriell ausgelesen werden kann. Hierfür wird lediglich ein Ein-/Ausgang des Mikrocontrollers benötigt.

Abbildung 2.2: Aufbau der Seriennummer im DS2401

Die eindeutige Seriennummer gliedert sich in drei Blöcke mit insgesamt 64 Bit. Die ersten 8 Bit beschreiben den *Family Code*, der statisch auf `0x01h` gesetzt ist. Die nächsten 48 Bit kennzeichnen die eigentliche Seriennummer, die durch weitere 8-Bit CRC-Prüfsummen geschützt ist. Die Seriennummer kann z. B. zur Initialisierung von Zufallszahlen oder zur eindeutigen Identifizierung von Knoten im Netz eingesetzt werden.

2.2.4 ATMEL AT45DB041B

Dieser Chip wird über das *Serial Peripheral Interface* (SPI) angesprochen und kann zur dauerhaften Ablage von Meßwerten verwendet werden. Er bietet dazu 528 KByte Speicher, die in 2.048 Seiten zu je 264 Byte organisiert sind. Die Zeit zum Lesen einer Seite beträgt 300 μs, zum Schreiben jedoch 20 ms. Durch zwei Seitenspeicher werden die Operationen gegenüber einem angeschlossenen Mikrocontroller zeitlich entkoppelt, so daß dieser nicht auf das Ende der Schreiboperation warten muß. Die Betriebsspannung muß in einem Bereich zwischen 2,5 V und 3,6 V liegen. Für das Lesen benötigt er dann zwischen 4 mA und 10 mA. Beim Schreiben steigt der Strom auf 15 mA bis 35 mA. Im Vergleich zu allen anderen Komponenten des MICAz ist dieser Wert recht hoch. Es sollte also zugunsten der Laufzeit versucht werden, diesen Chip so selten wie möglich zu verwenden. Im Standby-Modus beträgt seine Stromaufnahme dann lediglich 2 μA.

2.2.5 Batterien

Für den langfristigen Betrieb eines mobilen Knotens sind Batterien der entscheidende Faktor. Sobald die Spannung oder Kapazität zu gering wird, stellt dieser die Arbeit sofort ein. Für eine lange Laufzeit sind neben geringem Stromverbrauch der Komponenten auch eine kleine Minimalspannung und Batterien mit hoher Kapazität nötig. Standardmäßig ist der MICAz mit einem Batterie-Fach für zwei AA-Zellen (auch *Mignon* genannt) ausgestattet. Eine Betrachtung der Laufzeit eines Knotens ist in Kapitel 4.2 auf Seite 39 angegeben.

2.3 Software

Als Softwareumgebung kommt zum einen TinyOS [Tin], zum anderen Telelogic Tau [Tel], als SDL-Entwicklungsumgebung, zum Einsatz. TinyOS ist die Entwicklungsplattform für die Laufzeitumgebung, die alle Funktionen des Mikrocontrollers ansteuert. SDL (*Specification and Description Language*) wird zur Anwendungsentwicklung eingesetzt und benötigt Schnittstellen zur Laufzeitumgebung, um den Mikrocontroller komplett nutzen zu können.

2.3.1 TinyOS

TinyOS ist eine Entwicklung der University of California, Berkeley. Es setzt auf der C-Erweiterung nesC [nes] sowie AVR-GCC [AVRc, AVRb] als Compiler auf. Der AVR-GCC ist ein Gnu-C-Compiler, der für den Einsatz von ATMEL-Mikrocontrollern optimiert ist. Die C-Erweiterung nesC ist ein Precompiler, der für die Entwicklung komponentenbasierter Laufzeitumgebungen mit wenig RAM geschrieben wurde. Der Standard-C-Compiler erlaubt es, nur einzelne C-Dateien anzulegen, diese einzeln zu kompilieren und dann mittels Linker zu einem Programm zusammenzufügen. Für den nesC-Compiler wurden zwei Arten von Dateien definiert: Konfigurationen und Implementierungen. Ein kleines Beispiel für beide Dateiarten ist in Listing 2.1 bzw. Listing 2.2 auf der nächsten Seite gezeigt. Sie bilden damit ein Interface nach, wie es auch häufig in der Hardwareentwicklung Verwendung findet. Die Entwickler programmieren die Implementierungen und stellen diese als fertige Bausteine zur Verfügung, ähnlich einem IC (Integrated Circuit). Später wird in den Konfigurationen festgelegt, wie die Komponenten miteinander interagieren sollen. Dies ist in der Hardwareentwicklung mit der Verdrahtung oder dem Entwickeln einer Platine vergleichbar. Das Resultat wird später als eine einzige C-Datei abgelegt.

```
1 configuration Blink {
2 }
3 implementation {
4   components Main, BlinkM, SingleTimer, LedsC;
5   Main.StdControl -> SingleTimer.StdControl;
6   Main.StdControl -> BlinkM.StdControl;
7   BlinkM.Timer -> SingleTimer.Timer;
8   BlinkM.Leds -> LedsC;
9 }
```

Listing 2.1: Beispielkonfiguration in TinyOS

In der angegebenen Konfiguration in Listing 2.1 wird das Modul mit dem Schlüsselwort **configuration** benannt. Das Schlüsselwort **implementation** gibt keine Implementierung an, sondern legt die benutzten Module und Implementierungen mittels **components** fest. In diesem Beispiel werden die Komponenten Main, BlinkM, SingleTimer und LedsC verwendet. Alle folgenden Zeilen geben die Verknüpfung der Module und Implementierungen an. Im Beispiel werden hier alle verfügbaren StdControls mit dem Modul Main verknüpft. Das Modul Main ist die TinyOS Hauptkomponente

und mit der *Main-Methode* von C zu vergleichen. Die Standard-Controls stellen eine Schnittstelle zur Initialisierung, zum Starten und Stoppen einzelner Module zur Verfügung. Die Zeilen 7 und 8 im Listing stellen eine Verbindung zwischen der Implementierung (Listing 2.2) und einem Timer sowie der Leuchtdiodenansteuerung her.

In Listing 2.2 ist die Implementierung gezeigt. Zunächst wird das Modul mit dem Schlüsselwort **module** benannt und alle Schnittstellen, die nach außen zugänglich sind, mit **provides interface** angegeben. Bei dieser Implementierung beschränken sich die Schnittstellen auf das StdControl-Interface. Alle von einer Implementierung selbst verwendeten Schnittstellen sind im Block **uses** zusammengefaßt. Im Abschnitt **implementation** wird nun die Funktionalität des Moduls beschrieben. **command** leitet eine Funktion ein, die per **call** aufgerufen werden kann. Wie in Zeile 47 bis 51 zu sehen ist, wird bei Auftreten eines Timerereignisses die rote LED umgeschaltet. Der hierfür nötige Timer wurde in der *start*-Funktion in Zeile 31 auf eine Sekunde initialisiert.

```
1  /**
2   * Implementation for Blink application.
3   * Toggle the red LED when Timer fires.
4   **/
5  module BlinkM {
6    provides {
7      interface StdControl;
8    }
9    uses {
10     interface Timer;
11     interface Leds;
12   }
13 }
14 implementation {
15
16   /**
17    * Initialize the component.
18    * @return Always returns SUCCESS
19    **/
20   command result_t StdControl.init() {
21     call Leds.init();
22     return SUCCESS;
23   }
24
25   /**
26    * Start things up. This just sets the rate for the clock component.
27    * @return Always returns SUCCESS
28    **/
29   command result_t StdControl.start() {
30     // Start a repeating timer that fires every 1000ms
31     return call Timer.start(TIMER_REPEAT, 1000);
32   }
33
34   /**
35    * Halt execution of the application.
36    * This just disables the clock component.
```

```
37     * @return Always returns SUCCESS
38     **/
39    command result_t StdControl.stop() {
40      return call Timer.stop();
41    }
42
43    /**
44     * Toggle the red LED in response to the Timer.fired event.
45     * @return Always returns SUCCESS
46     **/
47    event result_t Timer.fired()
48    {
49      call Leds.redToggle();
50      return SUCCESS;
51    }
52  }
```

Listing 2.2: Beispielimplementierung in TinyOS

TinyOS selbst ist komplett in nesC geschrieben und bietet damit die o. g. Vorteile. Da TinyOS gleichzeitig aber auch so offen entwickelt wurde, um für viele Plattformen möglichst einfach anpaßbar zu sein, wird hierbei deutlich mehr Code erzeugt, als für die Anwendung nötig wäre. Durch diesen Umstand kann es durchaus nötig und nützlich sein, von diesem stark modularen Konzept abzuweichen. Weiterhin werden von TinyOS viele Schnittstellen über Funktionsaufrufe zur Verfügung gestellt, die den Code unnötig aufblähen und unübersichtlich machen. Als Beispiel läßt sich jeweils eine Funktion zum Ein- bzw. Ausschalten einer Chipfunktion anstelle einer einzelnen Set-Funktion nennen.

Der Grund, die Entwicklung der Laufzeitplattform trotz der Einschränkungen in TinyOS durchzuführen, liegt in vielen Konstantendefinitionen, die für den MICAz bereits integriert sind, und so das Programmieren deutlich erleichtern. Durch diese umfassenden Definitionen lassen sich Bezeichnungen von Registern oder Positionen von Bits direkt aus der Dokumentation übernehmen. Das größte Problem ergibt sich aber daraus, daß keine dieser Konstanten oder Funktionen dokumentiert sind, sondern alle in Beispielquelltexten von TinyOS oder durch Ausprobieren der Bezeichnungen aus den Hardwaredokumentationen herauszufinden sind. An vielen Stellen ließ sich auch Code durch Redesign weiterverwenden und so einiges an Know-How der TinyOS Entwickler in die Laufzeitplattform einbauen.

2.3.2 Tau SDL

Die Hauptanwendung auf dem MICAz wird mittels der „SDL Suite“ von Telelogic entwickelt. Es soll hier keine Einführung in SDL gegeben werden, dazu sei auf das Internet, u. a. auf die Quellen [SDLa, SDLb] verwiesen.

Um mit SDL Code zu erzeugen, der mit der Laufzeitumgebung zusammenarbeitet, sollte zunächst die Arbeitsumgebung (Anhang A, Seite 51) eingerichtet werden. Sobald dies geschehen ist, kann die Anwendung, unter Berücksichtigung der Einschränkungen (Anhang B, Seite 55), entwickelt werden. Um spezielle Funktionen des Chips anzusprechen, existieren spezielle Packages. Sie erlauben beispielsweise das Setzen

der Zeit von Timern in Sekunden, Milli- und Mikrosekunden oder das Einschalten der LEDs. Zur Übersetzung des SDL-Systems wird der Targeting Expert benutzt, der die Anwendung schließlich in C-Code übersetzt und `Makefiles` generiert. Die wichtigsten Einstellungen sind in Anhang A, Seite 51, angegeben. Sobald der Code erzeugt wurde, wird die Anwendung mit der Laufzeitumgebung übersetzt und auf den Mikrocontroller geladen. Bei dem letzten Schritt kann es häufiger zu Problemen kommen, deshalb sind einige Probleme und deren Lösung in Anhang C (Seite 57) zusammengefaßt.

Kapitel 3

Entwicklung einer Laufzeitplattform für den MICAz

SDL-Systeme können in Telelogic Tau [Tel] mit dem Targeting Expert in ein ausführbares System überführt werden. Für PCs ist dieser Schritt „ein Klick“. Sobald man sich auf die Ebene der Mikrocontroller begibt, müssen vorher Anpassungen an den verwendeten Chip vorgenommen werden. Zu diesem Zweck ist in dieser Arbeit eine Laufzeitplattform in TinyOS 1.12 [Tin] geschrieben worden. Für das Einbinden eines Mikrocontrollers sind Anpassungen an der von Tau zur Verfügung gestellten Schnittstelle und ein spezieller Compiler für die jeweilige Hardware-Architektur notwendig. Für den MICAz sind durch die Möglichkeiten der Funkkommunikation und der beiden seriellen Schnittstellen weitere Anpassungen nötig, um diese auch aus SDL heraus nutzen zu können. Sobald alle Anpassungen durchgeführt sind, kann das SDL-System auf der Zielplattform zur Ausführung gebracht werden.

3.1 Anforderungen

An die Laufzeitplattform werden folgende Anforderungen gestellt:

- Implementieren eines Timers mit hoher Auflösung.
- Auf MICAz integrierte Leuchtdioden von SDL aus zugänglich machen.
- Ansteuern beider UART-Schnittstellen und Übertragen von Daten.
- Erstellen einer Schnittstelle zur Datenübertragung und Ändern von Parametern aus SDL für die UART-Schnittstellen.
- Ansteuern des Transceiver Chips CC2420.
- Erstellen einer Schnittstelle zur Datenübertragung und Ändern von Parametern aus SDL für den Transceiver.
- Integration aller Komponenten in ein SDL-System.
- Kompatibilität zum *SDL Environment Framework* (SEnF) [KGGR05].

3.2 Voraussetzungen

Wie bereits in Kapitel 2.3.1 angesprochen, wird TinyOS als Plattform eingesetzt. TinyOS selbst ist kein Betriebssystem, auf dem Programme laufen. Es ist entwickelt worden, um den Programmierer während der Arbeit durch eine Bibliothek zu unterstützen. Die Unterstützung reicht allerdings nicht so weit, daß Konzepte, wie sie von PC-Betriebssystemen bekannt sind, wie preemptives Multitasking, Multithreading oder dynamische Speicherverwaltung, unterstützt werden. Bei TinyOS handelt es sich vielmehr um eine Menge von Komponenten, die in Module aufgeteilt sind.

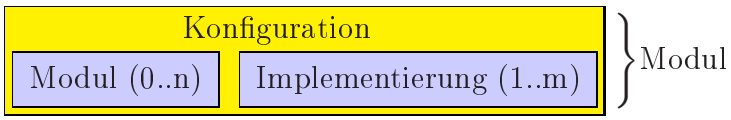

Abbildung 3.1: Zusammenhang Module, Konfigurationen und Implementierungen

Jedes dieser Module besteht aus einer Konfiguration, Implementierungen und evtl. weiteren Modulen. Implementierungen stellen hierbei die eigentliche Funktionalität her. Sie sorgen z. B. für die Ansteuerung der seriellen Schnittstellen. Die Konfiguration stellt nur die Verknüpfung zwischen den Implementierungen und anderen Modulen her. In der Konfiguration werden die offenen Schnittstellen der Module oder Implementierungen so miteinander verknüpft, daß sich ein neues Modul ergibt (siehe dazu Abbildung 3.1). Die serielle Schnittstelle beispielsweise besteht aus einer Konfiguration, die damit das Modul repräsentiert, und zwei Implementierungen. Eine der Implementierungen sorgt für die Hardwareansteuerung, also das Setzen von Registern und Schreiben von einzelnen Bits. Die andere Implementierung stellt eine einfache Schnittstelle bereit, um ein Byte zu verschicken oder zu empfangen. Die Konfiguration verknüpft nun beide Implementierungen so, daß die vereinfachte Schnittstelle auf die Hardwareschnittstelle zugreift. Durch diese Modularisierung lassen sich zu jedem Projekt nur die Module hinzunehmen, die für das Projekt nötig sind, um dadurch auch Speicherplatz zu sparen. Eine Dokumentation zu TinyOS und seinen Modulen fehlt größtenteils. Es sind lediglich einige Beispiele enthalten, die die Funktionsweise und benötigten Module zeigen.

Beim Kompilieren mit dem TinyOS-Compiler wird aus allen Komponenten eine einzelne C-Datei erzeugt. In allen Implementierungen soll ein disjunkter Namensraum zur Verfügung stehen. Da alle Module später zu einer Datei zusammengefaßt werden, wäre dies normalerweise nicht mehr gegeben. Deshalb wird allen deklarierten Variablen oder Funktionen der Name der Datei, gefolgt von einem $-Zeichen, vorangestellt. Die Funktion *ledsCtrl()* der Datei `SDLM.nc` wird damit zu *SDLM$ledsCtrl()*. Die entstandene C-Datei wird mit einem für ATMEL-Prozessoren modifizierten GCC-Compiler kompiliert und anschließend mit einem zusätzlichen Programm auf den Mikrocontroller geladen.

Aufbauend auf den Komponenten von TinyOS soll eine möglichst einfache Entwicklung ermöglicht werden, wofür SDL eingesetzt wird. Tau SDL wurde ausgewählt, da es sich hierbei um ein ausgereiftes, graphisches Entwicklungswerkzeug handelt,

das die Möglichkeit besitzt, C-Code für Mikrocontroller zu erzeugen. Die Modellierungsfunktionen von Tau SDL basieren auf dem SDL Standard von 1996 [IT95]. Bei SDL handelt es sich um eine Spezifikationssprache, die auf asynchron interagierenden erweiterten endlichen Automaten aufbaut. In der Initialisierungsphase können Aktionen ohne vorheriges Ereignis ausgeführt werden. SDL wechselt bei Erfüllung einer Bedingung aus einem Zustand unter ausführen der Transition in einen Folgezustand. Die Bedingung kann das Auftreten eines Signals, eines Timers, eines *Continuous Signals* oder einer *Spontaneous Transition* sein. Dabei werden alle Aktionen in der Transition sequentiell ausgeführt. Unter einen *Continuous Signal* versteht man einen wiederholten Test, der bei Erfüllung die Transition ausführt. Die *Spontaneous Transition* ist eine Transition, die indeterministisch ausgeführt wird. Gibt es eine weitere Transition, deren Bedingung erfüllt ist, wird nicht die *Spontaneous Transition* ausgeführt.

In SDL wird jede Spezifikation durch ein *System* beschrieben. Darin können *Packages* eingebunden werden. Ein Package ist eine Bibliothek und beinhaltet *Blocktypen* oder *Prozesse*. Jeder Blocktyp spezifiziert Signale an seinen Ein- und Ausgängen, die dieser verarbeitet. Innerhalb eines Blocktyps sind Funktionsblöcke sogenannte *Prozeduren* enthalten. Jede Prozedur verarbeitet die an seinen Eingängen angegebenen Signale und erzeugt seinerseits Signale, die an den Ausgängen aufgeführt sind. Innerhalb des Blocktyps kann die Signalweiterleitung festgelegt werden. Hierbei werden sowohl Signalwege zwischen den Prozessen als auch aus dem Blocktyp heraus bzw. hinein beschrieben. Im SDL-System können von diesen Blocktypen Instanzen als *Blöcke* angelegt werden und die Signalwege festgelegt werden. Blöcke können im System auch ohne vorherige Definition eines Blocktyps angelegt werden. Dies ist dann gleichbedeutetend mit einer Instanz eines Blocktyps. Signale, die in oder aus dem System herausgeführt sind, stellen die Schnittstelle zur Umgebung dar. Die Umgebung kann z. B. aus einem anderen System oder Sensoren und Aktuatoren bestehen. Ein einfaches Beispiel für ein System mit einem Block und einem Prozeß ist in den Abbildungen 3.2 auf Seite 20 und 3.3 auf Seite 21 gezeigt.

Der Targeting-Expert von Tau erzeugt aus dem so spezifizierten System C-Quellcode. Der Code beinhaltet das System selbst, die Ein- und Ausgabe-Schnittstelle sowie eine Speicherschnittstelle. Der Code des Systems ist eine direkte Umsetzung der Spezifikation nach festgelegten Regeln. Der *Scheduler*, der von Tau mitgeliefert wird, sorgt für die Abarbeitung aller anstehenden Signale und Timer. Ein Timer ist ein Spezialfall eines Signals. Ein Timer wird auf eine Zeit eingestellt, zu der ein Signal geschickt werden soll. In Tau wird durch den Scheduler nach Ablauf der eingestellten Zeit ein Timer-Signal generiert. Jeder Prozeß besitzt eine Queue, in die Signale, die an ihn geschickt wurden, aufgenommen werden. Sollte das Signal durch den Prozeß nicht empfangen oder explizit zur späteren Verwendung aufgehoben werden, wird es verworfen. In Cmicro ist statt einer Warteschlange pro Prozeß, nur eine zentrale Queue vorhanden. Vorerst ist der von Tau erzeugte Code plattformunabhängig. Durch die fehlende Hardwareanbindung ist er aber ebensowenig ausführbar.

Die Eingabe-/Ausgabeschnittstelle von Tau an die Umgebung wird auf zwei Weisen abgebildet. Zum einen gibt es Komponenten, die selbständig ihre Werte als Signal an das System schicken, die dann in die Warteschlange des Schedulers aufgenommen werden. Andere Komponenten schicken nicht selbständig ihre Werte, sondern

müssen nach einem neuen Wert gefragt werden. Dies bezeichnet man als *Polling*. Der Scheduler führt diesen Vorgang regelmäßig durch, in dem er eine Funktion der Umgebung aufruft. In diese Funktion können dann Werte abgerufen und darauf hin Signale erzeugt werden.

Aufgrund von Optimierungen von *Cmicro*, der Zielplattform für Mikrocontroller von Tau, müssen Signale aus der Umgebung einem Prozeß zugeordnet sein, an den das Signal gesendet wird. Generell müssen Signale aus der Umgebung nicht ihren Zielprozeß kennen, da die Pfade in SDL spezifiziert sind. Ohne eine solche Optimierung würde zu viel Prozessor-Last erzeugt und unnötig Speicher auf der Zielhardware verschwendet werden. Im SDL-System wurde ein Prozeß erzeugt, der für die Verteilung der Umgebungssignale verantwortlich ist. Sofern dieser Empfangsprozeß verwendet wird, müssen nicht für jedes neue System Anpassungen an den Eingabeschnittstellen erfolgen. Kommt ein Signal von der Umgebung nun in SDL in diesem Prozeß an, so sorgt dieser dafür, daß das Signal an den dort angegebenen SDL-Prozeß weitergeleitet wird. Durch diesen Ansatz bleiben alle Signalwege von SDL aus überschaubar und vor allem prüfbar. Durch die angesprochenen Optimierungen ist es sonst möglich, einem beliebigen Prozeß aus der Umgebung Daten zu schicken, auch wenn in SDL dazu kein Signalweg vorhanden ist.

Sobald Signale aus oder an die Umgebung mehr als einen Parameter besitzen, muß hierfür dynamischer Speicher im SDL-System (bzw. in der Implementierung des SDL-Systems) integriert werden. Dies ist auch notwendig, wenn Datentypen verwendet werden, deren Länge zur Compile-Zeit noch nicht feststeht. Zu diesen Datentypen gehören in SDL beispielsweise *Octet_string* oder *Array*. Zu diesem Zweck sind die Funktionsparameter für eine Speicherschnittstelle bereitgestellt. Es müssen dabei die Grundfunktionen, um Speicher zu reservieren und wieder freizugeben, implementiert werden. Speicher, der reserviert wird, sollte vorher auf „0“ initialisiert werden, um Seiteneffekte auszuschließen. Die Dokumentation zu dieser Schnittstellen ist in den Hilfetexten von Tau SDL [TAU] zu finden.

Das SDL-Environment Framework ist eine Sammlung von generischen Schnittstellen. Es ist dadurch möglich, das SDL-System unabhängig von der verwendeten Hardware zu entwickeln. In ihm sind Abstraktionen zu SDL-Basisdiensten wie der aktuellen Zeit oder der Reservierung von dynamischen Speicher definiert. Außerdem werden darin Zusatzfunktionen, wie die Verwaltung von Konfigurationsdateien, das generieren von Zufallszahlen oder die Ansteuerung hardwarespezifischer Funktionen angegeben. Gemäß dieser Vorgaben sind Signale von und an die Umgebung benannt und ihre Parameter gewählt. Ebenso muß der Prozeß, an den alle externen Signale gesendet werden, einen bestimmten Namen haben, der auch im Framework festgelegt ist. Durch das Einhalten dieser Schnittstelle ist es möglich, von der verwendeten Hardware zu abstrahieren. Mittels des *ns*+SDL-Simulators [KGGR05] wird das SDL-System dann so getestet, als würde es auf der Zielhardware laufen. In diesem Simulator lassen sich dann alle Signale, die intern oder extern gesendet werden, mit einem Zeitstempel ausgeben. Sofern die Zielplattform ausreichend dimensioniert ist, um alle Zeitschranken einzuhalten, lassen sich damit die Ergebnisse des Simulators auf die Zielhardware übertragen.

3.3 SDL-Laufzeit-Umgebung

Der Targeting Expert von Tau bietet die Möglichkeit, den C-Code für verschiedene Plattformen zu erzeugen. Die wichtigsten Code-Generatoren sind hierbei CAdvanced und der ressourcenbeschränkte Cmicro. Der CAdvanced-Compiler wird zur Erzeugung von Code für ressourcenstarke Hardwareplattformen, wie dem PC, verwendet. Keiner der Compiler unterstützt den kompletten Sprachumfang. Die Einschränkungen sind beim CAdvanced am kleinsten. Der CAdvanced-Compiler unterstützt beispielsweise das **via all** Konstrukt an Ausgängen nicht, bei dem ein Signal an alle Ausgänge verschickt werden soll. In diesem Fall schickt der CAdvanced-Compiler das Signal nicht an alle Prozesse, sondern nur an einen. Der Cmicro-Compiler hingegen erzeugt optimierten Code für kleine Hardwareplattformen, wie Mikrocontroller. Mit dieser Plattform ist es möglich, ohne dynamischen Speicher auszukommen. Es dürfen dann allerdings keine Signale mit mehr als einem Parameter verschickt, oder Datenstrukturen, die dynamischen Speicher benötigen, verwendet werden. Der erzeugte Code ist dafür sehr klein und bei der Entwicklung des Compilers wurde auf die Erzeugung von sehr effizienten Code geachtet. In den Einstellungen zu dem Compiler lassen sich je nach Anwendungsgebiet weitere Einschränkungen oder Prüfungen ein- bzw. ausschalten. Durch diese Möglichkeit kann somit die Ausführungsgeschwindigkeit erhöht, sowie die Größe des Daten- und Programmspeichers gesenkt werden.

3.3.1 Ein- / Ausgabeschnittstelle

Zur Illustration der Ein- und Ausgabeschnittstellen sind die Listings 3.1 auf der nächsten Seite und Listing 3.2 auf Seite 17 eingefügt.
Die Ausgabeschnittstelle, wie sie in Listing 3.1 gezeigt ist, schaltet die rote LED ein bzw. aus. Der Funktionskopf, der in den Zeilen 1 bis 12 gezeigt ist, ändert sich je nach gesetzten Compiler-Flags. Dieser Funktionskopf wird von Tau vorgegeben. In Zeile 14 wird der Standard-Rückgabewert der Funktion auf **false** gesetzt, d. h. die Umgebung kann das Signal nicht verarbeiten. Mit der **switch**-Anweisung in Zeile 16 werden die gesendeten Signale unterschieden. Signale, die in SDL nicht definiert wurden, stehen zur Compile-Zeit auch nicht zur Verfügung. Ist das Package Lamp nicht eingebunden, existiert auch kein Signal SET_LAMP_RED. Dies wird deshalb mit den #**ifdef**-Anweisungen geprüft. Der Aufruf *Set_Lamp_Red* aus Zeile 19 ruft die Funktion in der Datei `SDL2TOS.c` auf, die für das Schalten eine TinyOS-Funktion verwendet. Der Parameter, den die Funktion aus dem Signal bekommt, wird aus der Struktur der Parameter von xmk_TmpDataPtr geholt. Der Cast auf yPDP_SET_LAMP_RED entspricht immer yPDP_, gefolgt von dem Signalnamen. In diesem Beispiel werden nur statische Daten übertragen. Ist dies nicht der Fall, muß der von dynamischen Strukturen belegte Speicher wieder freigegeben werden.

Die Eingabeschnittstelle ist in Listing 3.2 am Beispiel des SFD-Signals gezeigt. Die Abfrage der Existenz dieses Signals in Zeile 2 wäre nicht nötig. Sollte das Signal nicht gebraucht werden, muß so jedoch der Quellcode nicht mitkompiliert werden. Das Makro in Zeile 3 ist für alle Signale aus der Umgebung zuerst aufzurufen. Danach lassen sich, nach Erzeugen einer Variablen des Schemas yPDef_Signalname, die

```
1  xmk_OPT_INT    xOutEnv (xmk_T_SIGNAL xmk_TmpSignalID
2                          #ifdef XMK_USE_SIGNAL_PRIORITIES
3                            , xmk_T_PRIO xmk_TmpPrio
4                          #endif
5                          #ifdef XMK_USED_SIGNAL_WITH_PARAMS
6                            , xmk_T_MESS_LENGTH xmk_TmpDataLength,
7                             void xmk_RAM_ptr  xmk_TmpDataPtr
8                          #endif
9                          #ifdef XMK_USE_RECEIVER_PID_IN_SIGNAL
10                           , xPID xmk_TmpReceiverPID
11                         #endif
12                         )
13 {
14   xmk_OPT_INT xmk_result = XMK_FALSE;
15
16   switch (xmk_TmpSignalID){
17 #ifdef SET_LAMP_RED
18       case SET_LAMP_RED:{
19         Set_Lamp_Red((bool)((yPDP_SET_LAMP_RED)xmk_TmpDataPtr)->Param1);
20         xmk_result = XMK_TRUE;
21         break;
22       }
23 #endif
24   return xmk_result;
25   }
```

Listing 3.1: Signale an die Umgebung versenden, am Beispiel der roten LED (aus env.c).

einzelnen Parameter, wie in Zeile 5 zuweisen. In Zeile 6 wird das Signal mit allen Parameter an SDL übergeben. Der eventuell verwendete Speicher wird dabei von SDL wieder freigegeben. In Zeile 8 ist der Prozeß angegeben, an den das Signal gesendet werden soll. Der hier angegebene signalReceiver ist ein Treiber, der in SDL das Weiterleiten an den richtigen Prozeß veranlaßt. Innerhalb der env.c ist signalReceiver als eine Definition zu dem SEnF Prozeß XPTID_CC2420Driver abgelegt.

3.3.2 Makros

In Tau werden diverse Makros definiert, wie auch im Code des Listing 3.1 zu sehen ist. Durch die Verwendung einzelner Compiler-Schalter können sich die Aufrufparameter von Funktionen ändern. Um nicht den Überblick in den Parametern zu verlieren, sollte man Funktionsköpfe aus der Dokumentation von Tau oder aus einem Vorlagenverzeichnis kopieren. Sonst kann ein unbedachtes Setzen eines Parameters dafür sorgen, daß das ganze Projekt nicht mehr kompiliert werden kann. Innerhalb des Listings 3.1 wird mittels der **#ifdef**-Anweisung die Existenz eines Signals geprüft. In Listing 3.2 werden einige wichtige temporäre Variablen für den Empfang von Signalen durch den Aufruf von XMK_SEND_TMP_VARS erzeugt.

```
1  void ZIGBEE_Sfd(bool success){
2  #ifdef ZIGBEE_SFD
3      XMK_SEND_TMP_VARS
4      yPDef_ZIGBEE_SFD fromEnv_param;
5      fromEnv_param.Param1 = success;
6      XMK_SEND_ENV( ENV, ZIGBEE_SFD, xDefaultPrioSignal,
7          sizeof( yPDef_ZIGBEE_SFD), &fromEnv_param,
8          GLOBALPID(signalReceiver,0));
9  #endif
10 }
```

Listing 3.2: Signale aus der Umgebung empfangen, am Beispiel von des SFD-Signals (aus `env.c`).

3.3.3 Kritischer Abschnitt

Für die Entwicklung ist die Definition der Makros XMK_BEGIN_CRITICAL_PATH (Listing 3.5 auf Seite 31) bzw. XMK_END_CRITICAL_PATH enorm wichtig. Sie sichern kritische Abschnitte gegen Unterbrechungen durch Interrupts oder auch die Preemption des Schedulers ab. Sofern eigener Code geschrieben wird, sollten diese beiden Makros verwendet werden, um kritische Abschnitte zu schützen. Generell sollten diese Abschnitte so kurz wie möglich sein. Dieser Schutz wird immer dann nötig, wenn unterschiedliche Prozesse auf gemeinsame Variablen zugreifen. Ein einfaches Beispiel für einen kritischen Abschnitt ist das Lesen, Erhöhen des gelesenen Wertes und das Zurückschreiben dieses Wertes. Ein Prozeß wird gestartet und an einer Stelle durch einen anderer Prozeß mit gleicher Berechnung unterbrochen. Nach dessen Berechnungsende wird die Ausführung des ersten Prozesses fortgesetzt. Das resultierende Ergebnis der Berechnung ist allerdings falsch. Der erste Prozeß wird beispielsweise den Wert 3 lesen, danach unterbrochen, der zweite erhöht den Wert auf 4 und schreibt diesen zurück. Dadurch, daß der erste Prozeß den Wert nicht erneut liest, erhöht er ihn ebenfalls auf 4 und schreibt diesen zurück. In der Variablen verbleibt der Wert 4 obwohl er richtig 5 lauten müßte.

3.3.4 Dynamischer Speicher

Dynamischer Speicher wird benötigt, wenn dynamische Datenstrukturen oder Signale mit mehr als einem Parameter verwendet werden. Sobald eines von beiden nötig wird, muß eine Anbindung der Speicherschnittstelle hergestellt werden. In Listing 3.3 ist die Anbindung an die entwickelte Speicherschnittstelle dargestellt. Tau stellt als Schnittstellen die Funktionen *Meminit*, *xAlloc* und *xFree* zur Verfügung. Die Funktion *MemInit* wird zur Initialisierung des Speichers benötigt. Hiermit werden alle Listen, die Verweise auf Speicherstellen enthalten, initialisiert. Die Funktion *xAlloc* wird zur Reservierung von Speicher der angegebenen Größe verwendet. Die Dokumentation von Tau SDL beschreibt, daß in der Standardimplementierung die Funktion *calloc* verwendet wird, die den Speicher zunächst auf „0“ setzt. Ob dieser Schritt nötig ist, bleibt allerdings offen. Um hier Probleme zu vermeiden, wurde die Funktion *calloc* nachempfunden und der Speicher zurückgesetzt. Es bleibt aber zu prüfen, ob dies notwenig ist, da auch hierfür Zeit benötigt wird. Die Funktion *xFree*

```
9  /** Initialization of the mem
10 */
11 void xmk_MemInit( void * _mem_begin, void * _mem_end ){
12     memoryInit();
13 }
14
15 /** allocate rsize space in mem
16 */
17 void * xAlloc( xptrint rsize ){
18     void* ptr=memoryAlloc(rsize);
19     memset(ptr,0,rsize);
20     return ptr;
21 }
22
23 /** free the mem used for this pointer
24 */
25 void xFree ( void  ** xmk_MemPtr ){
26     memoryFree(*xmk_MemPtr);
27     *xmk_MemPtr=0; //CRASH, für SDL nötig!
28 }
```

Listing 3.3: Speicherschnittstelle (aus `mk_cpu.c`).

gibt den durch die *xAlloc*-Funktion reservierten Speicher wieder frei und löscht den Zeiger auf die Speicheradresse.

3.3.5 Einschränkungen

Für die Entwicklung von Systemen, die mit Cmicro übersetzt werden soll ergeben sich einige Einschränkungen. Alle Einschränkungen sind in Anhang B auf Seite 55 beschrieben. Die stärkste Einschränkung stellt die Vermeidung der dynamischen Datentypen dar. Durch die Implementierung der Speicherschnittstelle lassen sich diese dennoch verwenden. Bei den Prozessen muß die Anzahl der Instanzen direkt festgelegt werden und kann während der Laufzeit auch nicht mehr geändert werden. Die Ausdrücke **any** bzw. **Output via all** sollte in einem Produktivsystem ohnehin nicht verwendet werden. Konstrukte wie diese zeigen meist eine Designschwäche. Dadurch wird man während der Entwicklung schon dazu gezwungen, diese Konstrukte zu vermeiden, sofern sie mit Cmicro kompiliert und getestet werden sollen.

3.4 Design

Das Design der Laufzeitplattform wurde maßgeblich durch die Möglichkeiten von TinyOS beeinflußt. Unter Berücksichtigung von möglichst langer Laufzeit und geringem Speicherverbrauch ergeben sich zusätzliche Einschränkungen. Es existieren zusätzlich zur Hardware weitere Einschränkungen in TinyOS selbst. Der mögliche Namensraum von Variablen wird verkleinert, indem TinyOS das $-Zeichen innerhalb aller Variablen und Funktionen verbietet. Der Grund dafür ist der disjunkte Namensraum jeder einzelnen Datei. Jeder Variablen wird der Dateiname, gefolgt von

einem $-Zeichen, vorangestellt. Auf den ersten Blick scheint dies unbedeutend. Der Linker akzeptiert jedoch keine Funktionen, die ein $-Zeichen enthalten.

Der von SDL generierte Code kann nicht direkt zusammen mit TinyOS kompiliert werden. Es müssen aber Funktionen aus TinyOS aus den SDL-Code-Blöcken aufgerufen werden. Normalerweise würden diese Funktionen als extern deklariert und dann ganz normal aufgerufen.

Die Auswirkungen dieses Sachverhalts zeigt das folgende Beispiel: Ist in der Datei A beispielsweise eine Funktion *void test()* definiert, erstellt der Compiler daraus *A$test()*. Der Versuch, diese Funktion aus Datei B aufzurufen und dies mittels der Definition *extern void A$test()* zu realisieren, wird vom Linker abgelehnt. Die Lösung des Problems ist der Aufruf von Funktionen über Variablen, sogenannte Funktionspointer. Hierbei wird ein Pointer, der auf eine Funktion verweist, in einer Variablen übergeben.

Die Anbindung von TinyOS-Quellcode an den von Tau generierten C-Code kann durch diesen Mechanismus erfolgen. Die Signale, die von und an die Umgebung geleitet werden, müssen auf Funktionen der Laufzeitumgebung abgebildet werden. Sobald an ein Signal von der Umgebung mehr als ein Parameter übergeben wird, oder einer der Parameter kein primitiver Datentyp ist, wird dynamischer Speicher benötigt. Unter primitiven Datentypen sind SDL-Datentypen zu verstehen, die sich direkt auf C-Datentypen abbilden lassen und nur statisch Speicherplatz beanspruchen.

3.4.1 Ein- / Ausgabe

Ohne Ein- und Ausgabeschnittstelle kann von dem Mikrocontroller nicht festgestellt werden, ob er überhaupt arbeitet. Durch Hinzufügen weiterer Hardware, die nicht zur Verfügung stand, kann die Funktion natürlich auch durch einen Hardwaredebugger sichergestellt werden. Doch soll jeder Knoten sowieso mit der Umwelt interagieren. Die Implementierung stellt damit keinen zusätzlichen Aufwand dar. Zu den Schnittstellen des Controllers zählen hierbei die Leuchtdioden, die serielle Schnittstelle und auch die Funkschnittstelle des Transceivers. Die Leuchtdioden lassen sich ohne weitere Hilfsmittel direkt ablesen. Für die serielle Schnittstelle reicht ein einfaches Datenkabel und ein PC. Die Funkschnittstelle läßt sich nur mit einem anderen Knoten oder einem speziellen Monitor für den ZigBee-Standard abhören.

3.4.1.1 Leuchtdioden

Die erste implementierte Schnittstelle zur Ausgabe von Daten sind die Leuchtdioden. Zwar lassen sich darüber nur wenig Informationen ausgeben und ein schnelles Blinken ist nur schwer wahrnehmbar. Im Fehlerfall können sie jedoch wichtige Statusmeldungen anzeigen, die über alle anderen Schnittstellen nicht mehr ausgegeben werden können. Um die Leuchtdioden nicht nur in der Laufzeitumgebung ein- und ausschalten zu können, mußte in SDL eine Schnittstelle dafür erstellt werden. Für die Funktion wurden die Signale in einem Package Lamp zusammengefaßt und die erste Signal-Ankopplung mit der Laufzeitumgebung realisiert. Mit diesem Package

läßt sich jede der drei Leuchtdioden über Signale ein- und ausschalten. Nur wenn in einem System das Package eingebunden wird, werden die Funktionsaufrufe später in den Quellcode aufgenommen. Da der Linker unbenutzten Code nicht erkennen kann, werden derzeit die nicht verwendeten Module in der Laufzeitumgebung weiterhin mitkompiliert und verbrauchen dort unnötig Platz.

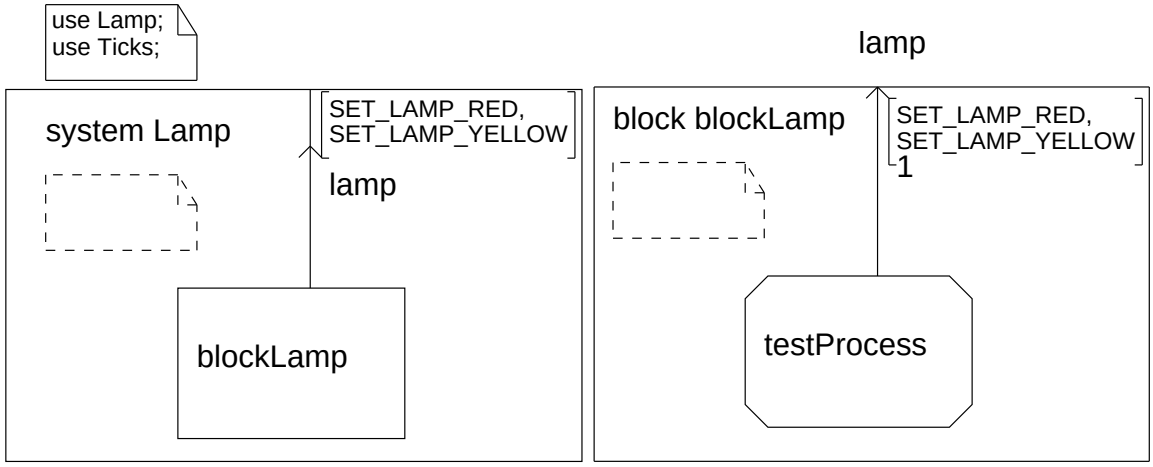

Abbildung 3.2: Beispielsystem für das Senden von Signalen an die Umgebung. Dargestellt: System links, Block rechts.

Die Funktionsweise der Signale, die an die Umgebung geschickt werden sollen, ist derart realisiert, daß in einem Systemprozeß das jeweilige Package eingebunden wird. Die verwendeten Signale werden an die Ausgänge des enthaltenen Blocks geschrieben. Innerhalb des Blocks sind Prozesse enthalten. Jeder Prozeß, der ein Signal nach außen sendet, muß dies durch einen Signalweg an den Rand des umgebenden Blocks deutlich machen. Innerhalb des Prozesses können die Signale genau wie lokale Signale auch verwendet werden. Bevor ein Signal an die Umgebung geschickt wird, werden die Parameter zuerst kopiert. Beachtet werden muß, daß Speicher für dynamische Datenstrukturen an die Umgebung zwar in SDL reserviert wird, aber in der Umgebung freigegeben werden muß. In Abbildung 3.2 und 3.3 ist ein System angegeben, das das Einschalten der roten oder gelben Lampe im Sekundenwechsel zeigt.

Es ist zunächst das System mit dem darin enthaltenen Block blockLamp gezeigt. Die Anweisung use Lamp gibt an, daß das Package Lamp verwendet werden soll, in dem die Signale zur Ansteuerung der Leuchtdioden definiert sind. Die zweite Anweisung use Ticks gibt an, daß zur Umrechnung der Zeit das Package Ticks verwendet wird. Die Bezeichnungen SET_LAMP_RED und SET_LAMP_YELLOW geben an, daß diese Signale an die Umgebung weitergeleitet werden. Die verwendete Bezeichnung lamp stellt den Namen des Signal-Pfades, der rechts im Bild zur Verknüpfung dient, dar. Im Block rechts sind im Signal-Pfad auch beide Signale aufgeführt, die an die Umgebung gesendet werden sollen. In Abbildung 3.3 ist der Prozeß selbst dargestellt. Hierbei wird zunächst ein Timer aufgezogen, der je nach Zustand des Systems die rote Lampe ein- und die gelbe ausschaltet oder umgekehrt. Der Timer wird hier mit Hilfe der beschriebenen Prozedur auf eine Sekunde aufgezogen. Befindet sich das System im Zustand RED, so ist die rote Leuchtdiode eingeschaltet. Nach Ablauf einer

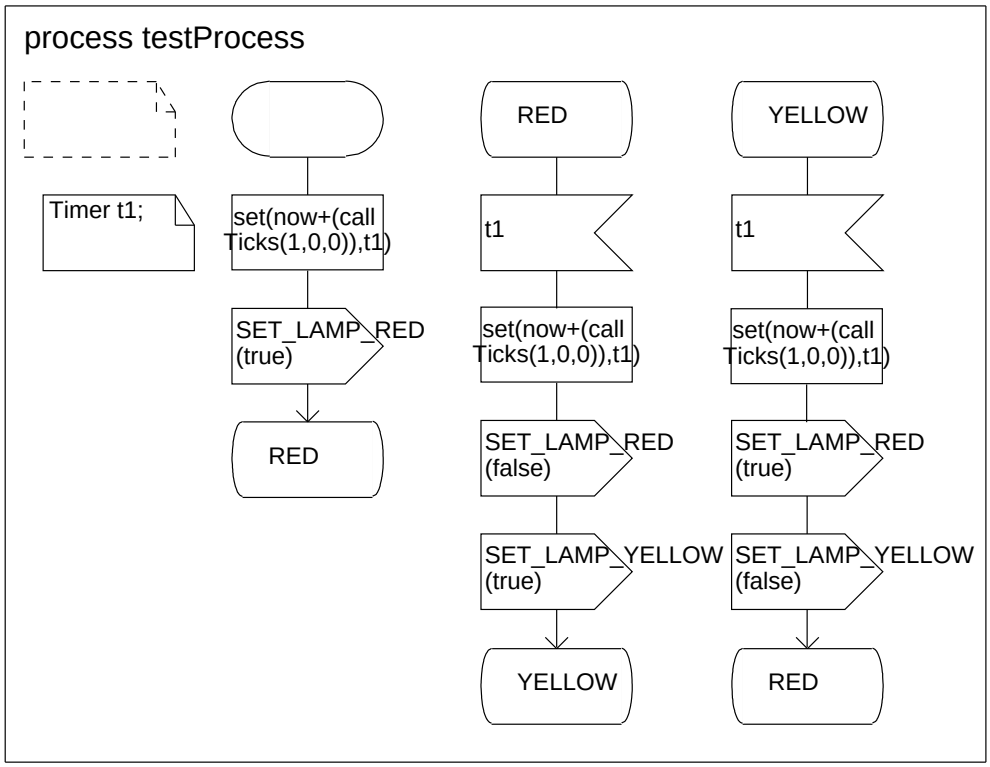

Abbildung 3.3: Beispielsystem (Prozeß) für das Senden von Signalen an die Umgebung.

Sekunde wird die rote LED aus- und die gelbe LED angeschaltet und in Zustand YELLOW gewechselt. In diesem Zustand wird bei Auslösen des Timers zuerst die rote LED ein- und die gelbe ausgeschaltet und zurück in Zustand RED gewechselt, wo das Spiel von vorne beginnt.

3.4.1.2 Serielle Schnittstelle

Nach der Implementierung der Leuchtdiodenfunktion sollen auch „Textdaten“ aus dem SDL-System an die Umgebung versendet werden. Eine serielle Schnittstelle, die in ein neues System nur eingebunden zu werden braucht, steht in TinyOS bereits zur Verfügung. Für die zweite Schnittstelle wurde eine Erweiterung der Ersten durch Kopieren und Ändern einzelner Werte durchgeführt. Zusätzlich wurde eine Erweiterung beider Schnittstellen eingebaut, um die Übertragungsgeschwindigkeit einstellen zu können. Die ursprüngliche Schnittstelle für die Kommunikation über UART sah es nur vor, einzelne Zeichen zu verschicken. Dies wurde so erweitert, daß ganze Zeichenketten ausgegeben werden können. Sollte die Schnittstelle gerade belegt sein, während ein neuer Sendewunsch eintrifft, wird dieser nicht, wie man es erwarten könnte, zwischengespeichert, sondern verworfen und dies an das SDL-System mit einem Signal UART_FAILED gemeldet. Der Grund hierfür liegt darin, daß die Implementierung der seriellen Schnittstelle nicht auf dynamischen Speicher angewiesen ist und somit nur einen Zeiger auf die zu sendenden Daten behält. Die Sendewünsche der beiden Schnittstellen werden jedoch getrennt behandelt.

3.4.1.3 Schnittstelle zum Transceiver

Die Ansteuerung des Chipcon-Transceiver-Chips stellte sich als besonders schwierig heraus. Sowohl für eine Sende- als auch für eine Empfangsoperation sind zunächst eine Menge von Schritten zu befolgen, bevor der Chip eine der beiden Operationen startet. Die Komponenten von TinyOS, die sich mit der Kommunikation des CC2420 beschäftigen, waren sehr verstreut, was die Übersichtlichkeit erschwerte. Für die ZigBee-Technik [Zig] existieren nicht, wie für WLAN (Wireless LAN), preiswerte Adapter, die sich mit einem Überwachungs-Programm bestücken lassen. Während der Entwicklung war es dadurch schwer zu überprüfen, ob Sende- oder Empfangsfehler aufgetreten sind. Es ließ sich nicht unterscheiden, ob Daten gesendet und nicht empfangen oder gar nicht gesendet wurden. Eine gezielte Fehlersuche war so nicht möglich. Durch einige Experimente und ein genaues Studium der Bedienungsanleitung des CC2420 ließ sich dessen Funktion sicherstellen und es wurden einfache Empfangs- und Senderoutinen in das `Wireless.nc`-Modul integriert.

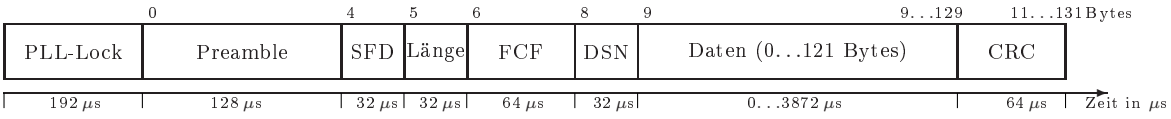

Abbildung 3.4: Format und zeitlicher Verlauf eines Rahmens

Ein Paket, das über die Funkschnittstelle versandt wird, hat das in Abbildung 3.4 gezeigte Format. Bevor eine Sendeoperation durchgeführt werden kann, müssen die Sendefrequenz und die Phase stabilisiert sein. Dies wird als *PLL-Lock* bezeichnet. Innerhalb dieser Einschwingzeit befindet sich der Transceiver-Chip weder im Empfangs- noch im Sendemodus. Beim Design von Systemen muß darauf geachtet werden, daß der Chip während der 192 μs, die das PLL-Lock dauert, auch nicht registriert, ob eine andere Station gerade das Medium belegt. Sobald die Einschwingzeit vorbei ist, wird zunächst eine Präambel von vier Byte geschickt, die einen neuen Rahmen auf dem Medium ankündigt und zur Synchronisation der Stationen dient. Es folgt ein *Start of Frame Delimiter* (SFD), der den Beginn des Rahmens auf dem Medium kennzeichnet. Sobald dieser auf das Medium gelegt wurde, signalisiert der Transceiver-Chip ein erfolgreiches Senden.

Jeder empfangsbereite Knoten fängt nach Erkennen der Präambel und des SFD an, Daten zu empfangen. Vor den Nutzdaten wird ein Längenfeld geschickt, das maximal einen Wert von 127 annehmen darf und sich auf das Feld selbst sowie alle folgenden Felder bezieht. Hieraus folgt, daß maximal 121 Byte Nutzdaten geschickt werden können. Im *Frame Control Field* (FCF) wird angegeben, ob die Daten beispielsweise verschlüsselt sind oder ein *Auto-Acknowledgement* gefordert wird. Die *Data Sequence Number* (DSN) ist ein Feld, das bei jedem gesendeten Paket um einen Zähler hochgezählt wird und dadurch für die Reihenfolgeerhaltung der Nachrichten sorgen kann. Die CRC-Prüfsumme kann von der Hardware selbst berechnet und bei neu ankommenden Paketen auch selbständig überprüft werden. Bei der verwendeten CRC handelt es sich um eine CRC-16-Prüfsumme. Sollte eine andere Prüfsumme gewünscht sein, kann diese im Mikrocontroller berechnet und anstelle der CRC-16-Prüfsumme geschrieben werden. Die automatische Prüfung kann dann nicht mehr

durch den Transceiver erfolgen, sondern muß wie die Berechnung im Mikrocontroller durchgeführt werden.

Die meisten Funktionen des Transceiver Chips sind durch Signale in SDL verfügbar. Die wichtigste ist hierbei die Setup-Funktion, mit der sich der Kanal und die gewünschte Sendestärke einstellen lassen. Weiterhin folgen Signale zum Senden mit und ohne vorherige Prüfung, ob das Medium belegt ist (CCA). Mit einem weiteren Signal lassen sich die Stromspar-Modi des Chips verändern, damit dieser weniger Leistung aufnimmt. Die Signale, die von der Laufzeitumgebung geschickt werden, zeigen sich im Automaten in Abbildung 3.5. Bei dem abgebildeten Automaten handelt es sich um einen Moore-Automaten [Moo56], der jedem Zustand eine Ausgabe zuordnet. An den Kanten oder Übergängen zwischen zwei Zuständen wird die Bedingung notiert. Eine „1“ als Bedingung wird auch manchmal als **true** bezeichnet und kennzeichnet einen Übergang ohne Bedingung. Alle hierin bezeichneten Zustände mit dem Präfix ZIGBEE werden als SDL-Signale verwendet. Zur besseren Unterscheidung sind alle internen Zustände des Transceivers in blauer Schrift gesetzt. Dieselbe Vorsilbe an ausgehenden Kanten sind Signale, die von SDL an die Umgebung geschickt werden. Alle Bedingungen, die nicht durch SDL-Signale beinflußt werden, sind in roter Schrift gesetzt. In dem gezeigten Automaten gilt: Ist der Automat in einem Zustand, und für ein ankommendes Signal ist kein Übergang spezifiziert, so bleibt er in diesem Zustand, und das Signal wird verworfen.

Initial befindet sich das System im Power-Down-Modus. Durch die Initialisierungssequenz von der Laufzeitumgebung (Bedingung init) wird das System bis in den Empfangszustand gebracht. Die Übergänge von und in die Energiesparzustände sind nur vom Empfangszustand aus zu empfehlen. Wie sich die Hardware genau verhält, falls ein ZIGBEE_MODE(0) Signal während des Sendens auftritt, ist nicht spezifiziert. Aus Gründen der Übersichtlichkeit ist der Zustand, der den Oszillator auf dem Übergang zum Power-Down-Zustand ausschaltet, weggelassen.

Ist der Empfangszustand erreicht und ein belegter Kanal wird entdeckt, so wird das Signal ZIGBEE_CCA(false) an das SDL-System gesendet. Sobald nun der SFD empfangen wurde, sendet die Laufzeitumgebung die Daten als ZIGBEE_RECV-Signal an SDL. Danach wird ein freier Kanal mit ZIGBEE_CCA(true) gemeldet. Dies geschieht auch, wenn der Kanal wieder frei ist und kein SFD empfangen wurde. Die Unterscheidung der Signale ZIGBEE_SEND_CCA und ZIGBEE_SEND zeigt sich nur im Zustand ZIGBEE_CCA(false). Bei belegtem Kanal wird die Sendeoperation nicht ausgeführt und als Fehler ein ZIGBEE_SFD(false) zurückgegeben. Das gleiche Signal wird auch gesendet, sofern während einer Sendeoperation ein weiteres Paket gesendet werden soll. Vor dem Senden eines Pakets muß zuerst der Oszillator neu kalibriert werden. Sobald dieser stabilisiert ist, wird das Senden gestartet. Ist der *Start of Frame Delimiter* übertragen, wird ZIGBEE_SENDING an das System gemeldet. Nach Abschluß der Übertragung wird dies mit einem ZIGBEE_SFD(true) bestätigt, der Oszillator erneut kalibriert und in den Empfangszustand gewechselt. Die genaue Beschreibung der Schnittstellen ist im Anhang G zu finden.

Funktionen des Transceivers, die derzeit von der Laufzeitplattform noch nicht unterstützt werden, sind die vorherige Verschlüsselung der Daten und die *Auto-Acknowledgement*-Funktion. Letztere kann allerdings nur für das originale Zigbee-Protokoll verwendet werden und durch den Chip selbst ausgeführt.

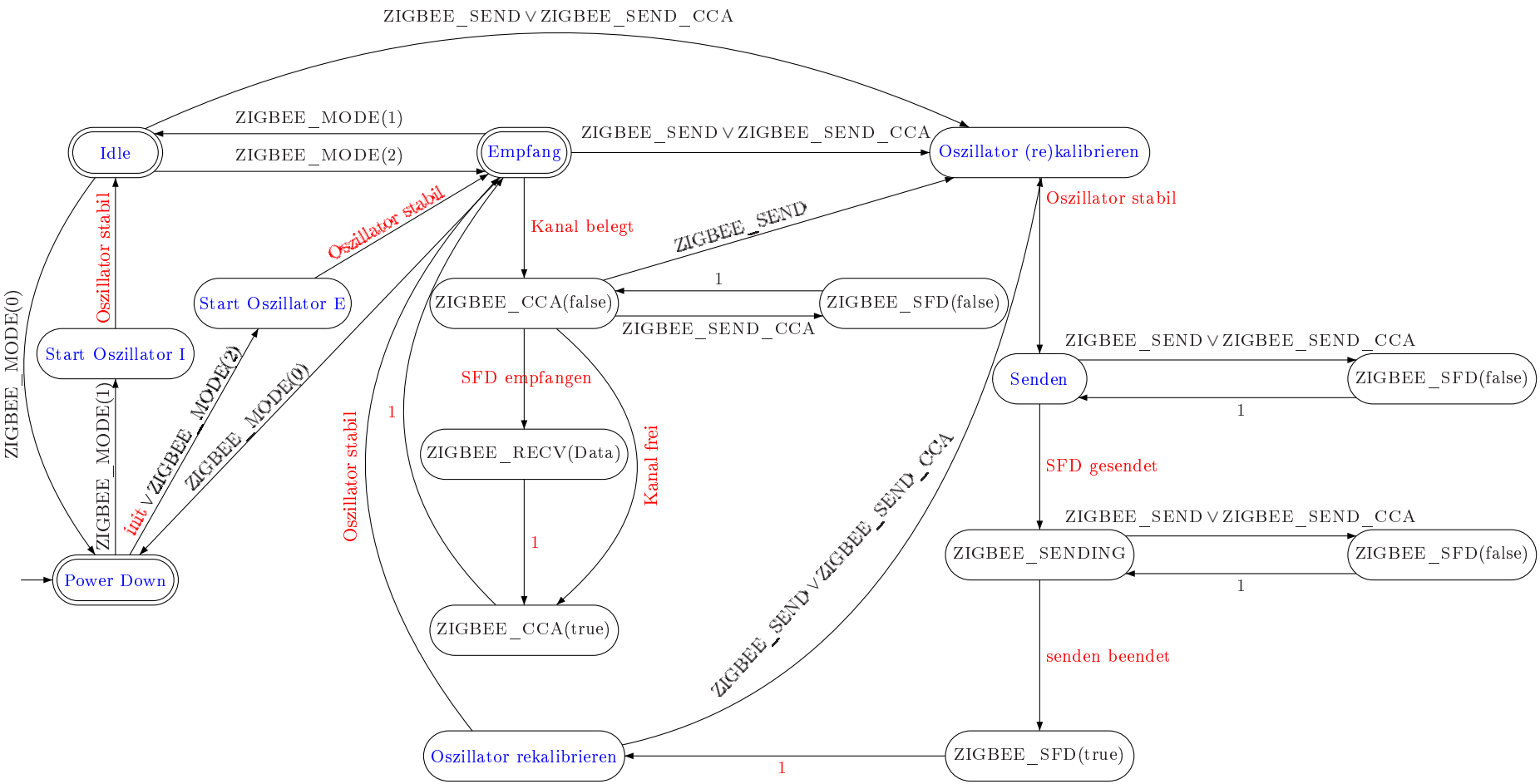

Abbildung 3.5: Signalverarbeitung des Transceivers aus der Sicht von SDL.

3.4.2 Timer

Für die Benutzung von SDL sind Timer entscheidend. Durch sie wird es möglich in Abhängigkeit der Zeit eine Transition auszuführen. Es ist dementsprechend nötig, SDL eine Zeitbasis zur Verfügung zu stellen, durch die die Timer aufgelöst werden können. Mit Hilfe der **now**-Funktion läßt sich so außerdem die Zeit zwischen zwei Ereignissen messen. Der erste Lösungsansatz erfolgte über einen der im Mikrocontroller eingebauten Timer. Dieser Timer hatte eine Auflösung von $1/32.768\text{Hz} \approx 31\,\mu\text{s}$. Ein mögliches Anwendungsszenario für die Laufzeitumgebung besteht in der Modellierung von MAC-Schichten. Hierzu wäre eine Auflösung von mindestens $1\,\mu\text{s}$ wünschenswert.

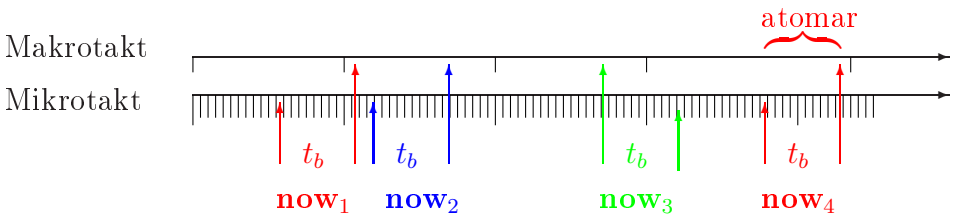

Abbildung 3.6: Problematik der Zeitabfrage

Der nächste Versuch bestand darin, die Timer direkt über den Prozessortakt von 7,3728 MHz laufen zu lassen. Damit wäre eine Auflösung von $1/7.372.800\,\text{Hz} \approx 136\,\text{ns}$ möglich. Da der Prozessor mit gleichem Takt läuft, hätte es passieren können, daß die Zeit für das SDL-System rückwärts (s. unten) läuft. Intern in der Hardware wird zuerst ein Zähler als Mikrotakt bis auf 255 hochgezählt und, sobald dieser überläuft, ein Interrupt ausgelöst. Durch diesen Interrupt wird nun eine Variable um 256, als Makrotakt, weitergezählt. Um die genaue Zeit zu erhalten, wurde der Hardwaretimer mit dem Wert der Variablen verknüpft. Da aber das Auslesen und Verknüpfen des Wertes aus dem Register Zeit (t_b) benötigt, konnte es passieren, daß die Zeit zum Zeitpunkt 254 ausgelesen wurde. Eine Möglichkeit besteht darin, zuerst die Hardwarezeit auszulesen und danach diese mit der Variablen verknüpfen ($\mathbf{now}_1$, Abbildung 3.6). Die Zeit in der Variablen zum Zeitpunkt des Auslesens der Hardwarezeit sei 256. Würde zusätzlich die Hardwarezeit mit einem Wert von 254 ausgelesen, so würde in der Zwischenzeit der Timer überlaufen, und die Variable hätte den Wert 2. Die an das SDL-System gemeldete Zeit wäre nun 2.254 und nicht, wie erwartet, 1.254. Würde nun direkt im Anschluß die Zeit erneut ausgelesen ($\mathbf{now}_2$), so würde von SDL aus gesehen die Zeit rückwärts laufen. Dieser Fall muß ausgeschlossen werden. Betrachtet man den umgekehrten Fall, also zuerst das Auslesen der Variablen und danach die Hardwarezeit bei gleicher Ausgangssituation ($\mathbf{now}_3$), so wäre die gemeldete Zeit beispielsweise 1.003. Beide Effekte können nur schwer beobachtet werden, eine Auswertung erster Meßwerte legte jedoch diese Erklärung für die Werte nahe.

Aufgrund dieser Überlegungen wurde als Zeitbasis der Prozessortakt mittels Hardware-Prescaler um den Faktor acht verkleinert. Daraus resultiert eine Genauigkeit von $8/7.372.800\,\text{Hz} \approx 1{,}1\,\mu\text{s}$. Dies entspricht in etwa der geforderten Genauigkeit und stellt zudem sicher, daß oben geschildeter Fall nicht eintritt. Der Prozessor hat nun acht Takte Zeit, um das Register zu holen und mit dem Wert

```
147   /** get the Systemtime, used by SDL */
148   xmk_T_TIME getSystime(){
149         xmk_T_TIME time;
150         atomic time=(inp(TCNT0)|(systime & 0xFFFFFF00));
151         return time;
152   }
```

Listing 3.4: Funktion zum Auslesen der aktuellen Zeit (aus `SDLM.nc`).

der Variablen zu verrechnen. Zusätzlich wurde der Block durch das Deaktivieren aller Interrupts als kritischer Abschnitt markiert und vor Unterbrechungen geschützt (**now**$_4$). Dies geschieht in TinyOS durch das Schlüsselwort **atomic**. Deshalb wird, wie in Abbidlung 3.6 gezeigt, der Makrotakt erst nach Ende des kritischen Abschnitts erhöht. In Listing 3.4 ist die verantwortliche Funktion aufgeführt. Die Variable systime stellt die Systemzeit ohne die Hardwarezeit dar.

In SDL werden Timer normalerweise mit einer Zeit in Sekunden als Gleitkommawert angegeben. Die minimale Zeit, die sich in SDL auf diese Weise darstellen läßt, ist eine Millisekunde. Dies wäre um den Faktor 1000 schlechter als die geforderte Timerauflösung. Erschwerend kommt noch hinzu, daß für den von Tau eingesetzten Cmicro Compiler, der speziell für Mikrocontroller optimiert ist, die Zeit nur in ganzen Sekunden angegeben werden kann. Im Targeting Expert lassen sich zur Zeitbasis verschiedenste Einstellungen machen, die aber alle keine Auswirkungen auf die Genauigkeit haben. Die Lösung, die in dieser Arbeit gefunden wurde, war die Zeit in SDL durch Hardware-Ticks zu repräsentieren, d. h. die Einheit eines Timers wird in $\sim$1,1 μs als Basis angegeben. Damit die Umrechnung nicht zu schwer zu Handhaben ist, ist in dem Package Ticks eine Prozedur enthalten, die als Argumente die Zeit in Sekunden, Millisekunden und Mikrosekunden entgegennimmt und diese korrekt in Ticks umrechnet.

Die Verwendung dieser Funktion ist gerade bei länger laufenden Timern anzuraten, da dies sicherstellt, daß der Timer die korrekte Zeit läuft. In dieser Prozedur wird Multiplikation und Division verwendet. Beide Funktionen stehen auf Hardwareebene nicht zur Verfügung, weshalb diese Prozedur nicht für kurze Timer verwendet werden sollte. Beide Funktionen werden durch den vom C-Compiler erzeugten Code nachgebildet, was enorme Rechenzeit kostet und in einer späteren Arbeit durch eine geeignete Funktion ersetzt werden sollte.

3.4.3 Dynamischer Speicher

Interfaces, wie die serielle Schnittstelle oder auch die Übertragung von Daten über den Transceiver-Chip, sollen auf jeden Fall mit der Datenstruktur *Octet_string* zur Verfügung stehen. Hierfür war die Verwendung von dynamischem Speicher zwingend erforderlich. Generell wird bei der Verwendung von Mikrocontrollern versucht, auf dynamischen Speicher zu verzichten. Der Einsatz macht eine statische Analyse vor dem Programmieren unmöglich. Da Ressourcen, insbesondere Hauptspeicher, sehr begrenzt verfügbar sind, können einige Programme unerwünschte Effekte zur

Laufzeit zeigen, die sich aus der Verwendung von dynamischem Speicher ergeben. Aus diesem Grund wurde eine eigene Speicherverwaltung entwickelt, die derzeit die *First-Fit*-Strategie beim Reservieren von freiem Speicher verwendet. Durch eine Eigenentwicklung besteht die Möglichkeit, die Strategie jederzeit zu ändern und, sofern gewünscht, einzelne Debug-Funktionen in diese Bibliothek einzubauen. Derzeit ist hier für den MICAz eine Funktion integriert, die den derzeit freien Speicherplatz auf der seriellen Schnittstelle ausgeben kann. Sofern man die Ausgabe aktiviert, sollte man beachten, daß die Hardware *Little-Endian*-Kodierung verwendet, also das niederwertigste Byte zuerst ausgibt. Stehen beispielsweise noch 1384 Byte Speicher zur Verfügung, wäre die Ausgabe in hexadezimaler Schreibweise: `0x68 0x05`.

3.4.4 Weitere Funktionen des Mikrocontrollers

Die zusätzliche Funktionalität des ATMEL ATMega 128L, wie Stromsparen in den verschiedenen Schlafmodi, ist zum jetzigen Zeitpunkt noch nicht implementiert, aber im `SDLM`-Hauptmodul mittels einer *sleep*-Funktion bereits vorbereitet. Um diese Funktion nutzen zu können, müssen zusätzliche Änderungen am Scheduler durchgeführt werden. Nur diesem ist bekannt, wann die nächste Transition ausgeführt wird und er könnte in der Zwischenzeit den Hauptprozessor zum Stromsparen anweisen. Er würde dann nur für die Interruptbehandlung und das Überlaufen des Hardware-Timers aufwachen. Außer den bekannten Funktionen bietet der Prozessor noch diverse Ein- und Ausgänge für digitale und analoge Signale, deren Funktion derzeit auch nicht bereitgestellt ist. Eine Erweiterung um diese Funktionen sollte aber in kurzer Zeit machbar sein, da die Dokumentation des Prozessors [ATMa] hier sehr ausführliche Angaben über die Ansteuerung und Verwendung macht.

3.4.5 Struktur der Laufzeitumgebung

In Abbildung 3.7 auf der nächsten Seite läßt sich die Struktur der Laufzeitumgebung ablesen. Aus Gründen der Übersichtlichkeit, beschränkt sich die Darstellung auf die veränderten oder direkt benötigten Komponenten. Die Hauptkomponenten SDL, SDL2TOS, TinyOS und memlib sind darin zu erkennen. Alle diese Komponenten werden einzeln kompiliert und danach zu einer fertigen Applikation vom Linker zusammengebaut. Innerhalb der Komponenten befinden sich Konfigurationen oder Module (Implementierungen) (siehe Abschnitt 3.2). Eine Konfiguration kann wieder Module oder andere Konfigurationen beinhalten. Eine fertige Komponente wird durch ihre Konfiguration repräsentiert.

Module bilden hierbei, wie man am Namen erkennen kann, nur Teilkomponenten ab. Sollte eine Unterscheidung schwer fallen, ist dies meist an der Nomenklatur zu erkennen: Bei Modulen wird häufig ein „M“ vor der Dateiendung eingefügt, während Konfigurationen entweder nichts oder ein „C“ hinzugefügt wird, d. h., die Datei `SDL.nc` ist die Konfiguration, die direkt zu dem Modul `SDLM.nc` gehört. Zusätzlich zu diesen beiden Dateiarten gibt es auch noch die sogenannten Interface-Dateien, die eine Schnittstelle zu anderen Komponenten definieren; sie sind vergleichbar zu Header-Dateien in C. Die Kommunikation zwischen den Teilkomponenten findet entweder

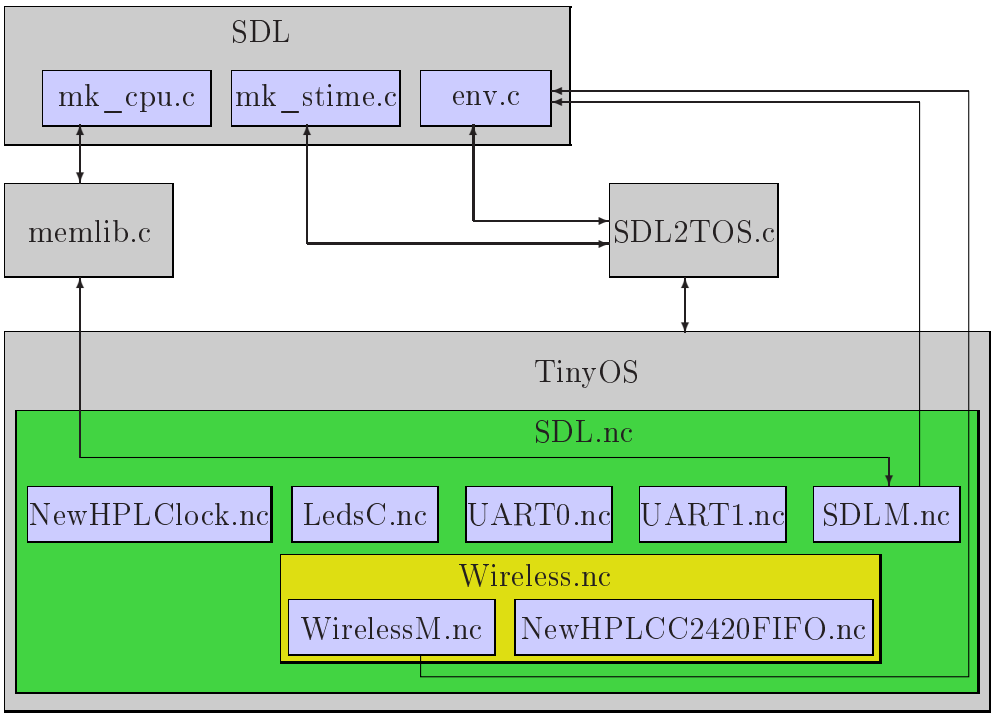

Abbildung 3.7: Struktur der Laufzeitumgebung. Nicht aufgeführt: Interfaces und unveränderte Dateien aus TinyOS.

über die in den Konfigurationen definierten Interfaces statt oder auf den eingezeichneten direkten Verbindungen. Betrachtet man beispielsweise das SDL2TOS-Modul, so gibt es hier Verbindungen zu `mk_stime`, `env` und zu dem Modul `TinyOS`. Dies soll zeigen, daß dieses Modul direkt mit den Modulen `mk_stime` und `env` kommuniziert, aber ganz allgemein mit TinyOS verknüpft ist, und hier Datenfluß durch die jeweiligen Konfigurationen gesteuert wird. Die Kommunikation, die über eine Konfiguration gesteuert wird, ist aus Gründen der Übersichtlichkeit entfallen.

3.4.6 Komponenten

Nachdem die Struktur der Laufzeitumgebung vorgestellt wurde, folgt eine detaillierte Beschreibung der Komponenten. Hierbei erfolgt eine Gliederung nach den Hauptkomponenten TinyOS, SDL2TOS und SDL. Die Komponente SDL2TOS dient als Vermittlungsschnittstelle und wird in diesem Abschnitt nicht detailliert betrachtet. An dieser Stelle soll die Beschreibung der Funktion und der einzelnen Module genügen. Eine genauere Beschreibung der Schnittstellen ist in Anhang G zu finden.

3.4.6.1 TinyOS

Die Komponente TinyOS besteht aus den Grundkomponenten, die für TinyOS nötig sind und automatisch vom Compiler hinzugefügt werden, außerdem aus den Erweiterungen, die im Rahmen dieser Arbeit entwickelt wurden. Die Konfiguration

`SDL.nc` stellt hierbei die Hauptkonfiguration dar, die alle darin enthaltenen Module miteinander verbindet.

- **Real-Time-Clock**
  Das Modul `NewHPLClock.nc` stellt eine Schnittstelle zu der auf dem Chip eingebauten Echtzeituhr (RTC) her. Sobald ein bestimmter Wert intern erreicht ist, wird ein Hardware-Interrupt ausgelöst, der von dieser Komponente aufgefangen wird. Der Wert und auch die mögliche Skalierung werden mit dieser Komponente eingestellt.

  In dem von TinyOS vordefinierten Modul `HPLClock.nc` gab es keine Möglichkeit, die Echtzeituhr mit dem Quarz-Oszillator zu verbinden und damit eine Genauigkeit zu erreichen, die der Taktfrequenz von 7.372.800 Ticks/Sekunde entspricht. Um Timing-Probleme zu vermeiden, wurde dies durch einen Prescaler auf 921.600 Ticks/Sekunde herabgesetzt. Vor der Änderung waren nur 32.768 Ticks/Sekunde über den eingebauten RC-Oszillator möglich. Diese Änderung wurde durchgeführt, um die Timergenauigkeit in SDL zu erhöhen.

- **Leuchtdioden**
  Für die Ansteuerung der Leuchtdioden (LED) auf dem MICAz kommt das Modul `LedsC.nc` zum Einsatz. An dieser Komponente waren keine Änderungen nötig. Es lassen sich damit die drei auf dem Board angebrachten Leuchtdioden getrennt voneinander ein- und ausschalten.

  Während der Entwicklung einer Applikation kommt diesem Modul eine besondere Bedeutung zu. Sollte ein schwerwiegendes Problem vorliegen, steht mit den LEDs oft die einzige verbleibende Schnittstelle nach außen zur Verfügung. Zu diesem Zweck finden sich innerhalb der `SDLM.nc`-Datei einige Debug-Schalter. Sie dienen der Überprüfung von Grundfunktionalitäten des Mikrocontrollers. Je nach ausgeführter Funktion werden dann die dafür vorgesehenen Lampen ein- oder ausgeschaltet. Eine genauere Erläuterung hierzu folgt in den weiteren Absätzen.

- **Serielle Schnittstelle**
  Die beiden seriellen Schnittstellen (UART0, UART1) werden mittels der beiden Module `UART0.nc` und `UART1.nc` gebildet. Hierbei handelt es sich um zwei Konfigurationen, die je ein Modul zur Ansteuerung der Schnittstelle benutzen. In TinyOS wird lediglich die erste Schnittstelle zur Verfügung gestellt. Außerdem ist die Geschwindigkeit der Schnittstelle auf 56,7 kbps fest eingestellt.

  Es wurde somit die zweite Schnittstelle durch Kopieren der ersten und Ändern einiger Hardwareansteuerungen zugänglich gemacht. Die fehlende Einstellmöglichkeit der Übertragungsrate wurde durch eine weitere Schnittstelle für UART0 und UART1 beseitigt. Es kann nun direkt auf Anwendungsebene die Übertragungsrate beider UART-Schnittstellen auch während des Programmlaufs angepaßt werden.

- **Hauptkomponente**
  Das `SDLM.nc`-Modul bildet die Hauptkomponente innerhalb der TinyOS-Umgebung. Dieses Modul wurde im Rahmen der Diplomarbeit als zentrale Komponente, die zur Anbindung von SDL dient, entwickelt.

Innerhalb dieses Moduls werden alle Signale der anderen Komponenten verarbeitet und an das Interface von `SDL2TOS.c` weitergegeben. Die einzige Ausnahme bildet hier der Empfang von Daten der seriellen Schnittstellen oder des Transceiver-Chips, die zugunsten von Größe und Geschwindigkeit direkt an das SDL-Interface weitergereicht werden. Eine Verbindung zu der Speicherbibliothek `memlib` wird benötigt, um Speicher für die Daten des Transceivers zu erhalten. Dieses Modul ist ebenfalls für das Auslesen der Echtzeituhr (RTC) zuständig, ohne die die meisten SDL-Systeme nicht lauffähig wären.

Falls Probleme mit der in SDL entwickelten Anwendung auftreten, können innerhalb dieser Datei verschiedene Debug-Flags gesetzt werden, die eine Überwachung der Laufzeitumgebung ermöglichen. Um die RTC zu überwachen, stehen zwei Flags DEBUG_TIMER bzw. DEBUG_TIMER_PRINT zur Verfügung. Das erste Flag läßt nur die gelbe Lampe im Sekundentakt blinken, während das zweite zusätzlich die aktuelle Zeit auf der seriellen Schnittstelle ausgibt. Die weiteren Flags DEBUG_UART bzw. DEBUG_WIRELESS dienen der Überwachung der seriellen bzw. der Funkschnittstelle. Mit dem ersten Flag wird die rote LED bei einer Kollision verschiedener Sendewünsche ein- bzw. ausgeschaltet. Das letzte Flag gibt die Möglichkeit, das Funkinterface zu überwachen. Bei einem erfolgreichen Empfang von Daten oder einem Sendeversuch wird die grüne LED ein- bzw. ausgeschaltet.

- **Transceiver-Chip**
  Die Konfiguration, die hier noch zu erwähnen ist, stellt das Modul `Wireless.nc` dar. Diese Komponente ist in der ursprünglichen Version von TinyOS nicht enthalten. Ein großer Teil der Implementierungen ist innerhalb von TinyOS so stark fragmentiert und damit auf verschiedene Quellcodedateien verteilt, daß die Zusammenhänge nur noch schwer zu überblicken waren. Aus diesem Grund ist diese Datei entstanden, die fast alle Teile der Funkübertragung zusammenfaßt. Die ursprünglichen Komponenten benötigten zusätzlich zwei Hardwaretimer und ließen sich nicht mit der Hauptkomponente zusammen kompilieren.

  Die Hauptkomponente ist für eine genaue Auflösung der Timer verantwortlich, da sie die Systemzeit verwaltet. Hierfür wurde bereits der feingranulare Timer des MICAz eingesetzt. Die ursprüngliche Komponente der Funkübertragung verwendete aber genau denselben Timer, so daß es hier zu einem Konflikt kam, der sich durch die Entwicklung der neuen Komponente ohne Timer beseitigen ließ.

  – Die Konfiguration beinhaltet auch das `NewHPLCC2420FIFO.nc`-Modul, das für das Lesen und Schreiben des FIFOs des Transceivers zuständig ist. Bevor Daten über den Transceiver übertragen werden können, müssen diese zunächst in den Sendepuffer geschrieben werden. Genauso wird jedes empfangene Paket zunächst in den Empfangspuffer geschrieben, bevor das Paket für die Anwendung zur Verfügung steht. Der Zugriff auf diese beiden Puffer wird von diesem Modul aus gesteuert. Das `NewHPLCC2420FIFO.nc`-Modul wurde komplett neu strukturiert und einfacher gestaltet. Die ursprüngliche Funktion, der Kontrolle beider FIFOs, wurde beibehalten.

– Das Modul `WirelessM.nc` übernimmt die Zugriffsteuerung auf den Funkkanal. Diese Datei ist durch Zusammenfassen aus diversen kleineren Einzelkomponenten entstanden. Die hieraus entstandenen Einbußen an die ursprüngliche Struktur von TinyOS sind für die Laufzeitumgebung nicht relevant. Eine große Änderung der Schnittstelle ergab sich schon durch Zusammenfassen einfacher Ein/Aus-Funktionen zu übersichtlicheren set-Funktionen.

3.4.6.2 SDL2TOS.c

Bei der Datei `SDL2TOS.c` handelt es sich um eine Schnittstellen-Datei, die Anfragen aus dem SDL-Environment auf die jeweiligen Funktionen von TinyOS abbildet. Die genaue Beschreibung dieser Schnittstelle und der Parameter der Funktionen sind in Abschnitt G.2 zu finden.

3.4.6.3 SDL

Bei SDL handelt es sich um den aus Telelogic Tau generierten Code, der zusammen mit einigen Anpassungen an die Laufzeitumgebung kompiliert wird. Tau selbst erzeugt Code für den Scheduler und die in SDL spezifizierten Transitionen. Zusätzlich dazu mußte in der Datei `mk_stime.c` die Abfrage der aus TinyOS zur Verfügung gestellten Systemzeit ermöglicht werden. Die Datei `mk_cpu.c` beinhaltet den Zugriff auf das dynamische Speicherinterface der `memlib.c`. Hier wird, um Fehlern in SDL vorzubeugen, nicht nur der Speicher reserviert. Er wird zusätzlich durch Setzen des Nullwertes „geleert“. In der Datei `env.c` wird das Environment abgebildet. Hierunter versteht man alle Signale, die nicht von SDL selbst verarbeitet werden, oder Signale, die von außerhalb des SDL-Systems kommen, in diesem Fall von der Laufzeitumgebung. Signale wie UART_FAILED, ZIGBEE_SFD oder ZIGBEE_SENDING werden aufgrund der Rückgabewerte der aufgerufenen Funktionen innerhalb von `env.c` erzeugt.

Beim Entwurf eines Systems, das auf der Laufzeitplattform aufbaut, sollte beachtet werden, daß alle Signale des Environments an *einen* Prozeß in SDL geschickt werden. Dieser Prozeß ist in `env.c` als signalReceiver bezeichnet. Der Standardwert hierfür ist XPTID_CC2420Driver.

```
852 #undef XMK_BEGIN_CRITICAL_PATH
853 #define XMK_BEGIN_CRITICAL_PATH __asm volatile ("cli");
854 #undef XMK_END_CRITICAL_PATH
855 #define XMK_END_CRITICAL_PATH __asm volatile ("sei");
```

Listing 3.5: Änderungen an der Datei `ml_typ.h`

An den vorhandenen Dateien von SDL, die zum Kompilieren von Cmicro-Code genutzt werden, mußten zwei kleine Änderungen vorgenommen werden. In der Datei `/opt/tau/runtime/cmicro/include/ml_typ.h` wurde beim Microsoft C++-Compiler, der hier für die Übersetzung eingestellt wurde, der kritische Abschnitt, wie in 3.5, konfiguriert. Beim Eintreten in einen kritischen Abschnitt werden somit

vorher alle Interrupts aus- und nach Ende des Abschnitts wieder eingeschaltet. Die zweite Änderung betrifft die Timerbehandlung in der Datei `/opt/tau/runtime/cmicro/include/sctpred.h`. In SDL sind normalerweise Timer nicht mit der gewünschten Auflösung möglich. Das Problem ließ sich durch einen Cast auf einen längeren Datentyp (*xmk_T_TIME*) beseitigen. Die Änderung ist in Listing 3.6 zu sehen.

```
1175 #ifdef XTIMEASINT_TICKS
1176 #define SDL_DURATION_LIT(R,I,D)       I
1177 #else
1178 #define SDL_DURATION_LIT(R,I,D)   ((xmk_T_TIME)(I)*SDT_TICKPERSEC +\
1179           (D)/SDT_NANOSECPERTICK)
1180 #endif
```

Listing 3.6: Änderungen an der Datei `sctpred.h`

3.4.6.4 memlib

Die `memlib.c` wurde von Thomas Kuhn entwickelt, um unabhängig vom C-Compiler dynamischen Speicher auf dem Mikrocontroller reservieren (allokieren) zu können. Die Komponente bietet Funktionen, die den C-Funktionen *malloc*, *free* und *realloc* entsprechen. Aus SDL werden diese Funktionen intern durch den Aufruf von *xAlloc* bzw. *xFree* verwendet. Innerhalb der Header-Datei `memlib.h` lassen sich einige Parameter wie die Größe des dynamischen Speichers oder Debug-Flags für den MICAz einschalten. Bei gesetztem Debug-Flag wird die aktuelle Größe des verfügbaren, freien Speichers auf der seriellen Schnittstelle ausgegeben und ermöglicht so das Finden von Speicherlöchern.

Kapitel 4

Ressourcen

In den vorherigen Kapiteln wurden alle Funktionen von SDL und der Laufzeitumgebung beschrieben. Für die Anwendungsentwicklung ist es in vielen Fällen nötig, Abschätzungen zur Geschwindigkeit des Systems zu machen. Es werden dazu einige Tests der Laufzeit von Signalen im System angegeben, die gerade für mögliche Kommunikationssysteme wichtig sind. Danach werden einige Betrachtungen zum Stromverbrauch der Komponenten des MICAz gemacht. Hierbei wird eine Worst-Case-Abschätzung getroffen und einige Anwendungsfälle exemplarisch durchgerechnet.

4.1 Timing

Häufig kann der Zeitverbauch von Transitionen oder die Dauer von Signalweiterleitungen innerhalb des SDL-Systems außer acht gelassen werden. Handelt es sich bei der Zielplattform um einen aktuellen PC, sollte dieser in der Lage sein, alle Transitionen rechtzeitig auszuführen, bevor ein Timer abläuft. Aber bei zeitkritischen Systemen oder Zielhardware mit beschränkten Ressourcen werden diese Zeiten signifikant. Sobald die Ressourcen so stark eingeschränkt sind, wie es bei einem Mikrocontroller der Fall ist, müssen Zeitabschätzungen getroffen werden. Erst durch die Messung kann geklärt werden, für welche Einsatzgebiete sich die entwickelte Laufzeitumgebung in Verbindung mit einem SDL-System eignet. Die verwendeten Testsysteme und die Meßmethode sind in Anhang E auf Seite 63 aufgeführt. Alle Werte in diesem Kapitel sind in Ticks angegeben. Ein Tick entspricht 8 Takten. Dies liegt daran, daß für die Systemzeit der Takt durch den Prescaler durch 8 geteilt wurde. Bei einer Taktfrequenz von 7.372.800 Hz läßt sich die Zeit für eine Einheit mit ungefähr 1,1 μs angeben.

4.1.1 Interne Signallaufzeit zwischen SDL-Prozessen

Signale spielen in SDL eine wichtige Rolle. Nur mit ihnen ist die Kommunikation zwischen Prozessen möglich. Außerdem dienen sie als Ereignisse, die weitere Aktionen auslösen. Es ist deshalb interessant zu ermitteln, wie lange ein Signal benötigt,

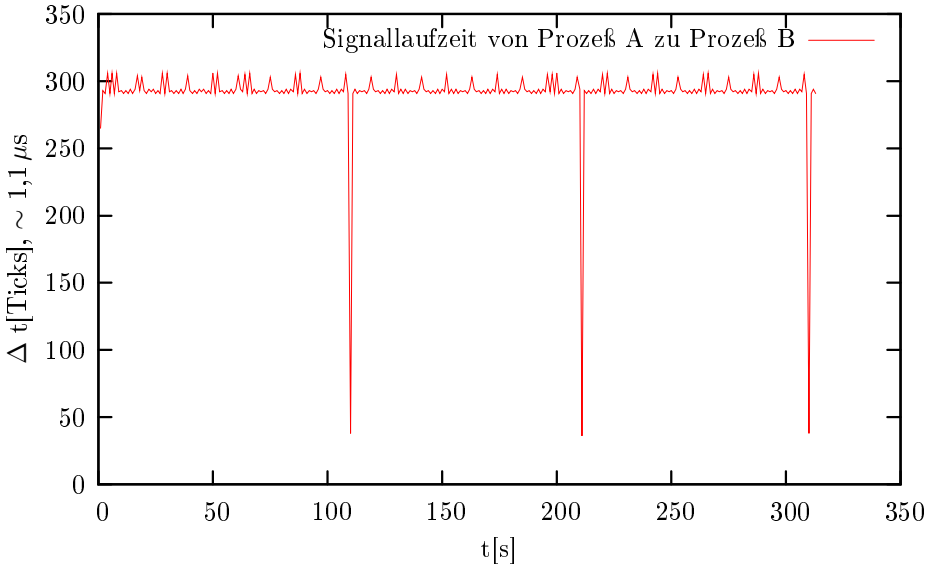

Abbildung 4.1: Signallaufzeit interner Signale eines lokalen Prozesses A zu einem lokalen Prozeß B

um von einem Prozeß A zu einem Prozeß B übertragen zu werden. Die Meßwerte zu diesem Test und auch das verwendete Beispielsystem sind im Abschnitt E.4 auf Seite 66 zu finden.

In diesem Test wurde periodisch ein Signal von Prozeß A an Prozeß B gesendet. In einer Variablen wurde der Sendezeitpunkt festgehalten und nach Empfang durch Prozeß B die Laufzeit berechnet. Die Ergebnisse dieser Messung sind in Abbildung 4.1 graphisch dargestellt.

Nach einer kurzen Einschwingphase benötigt SDL auf dem Mikrocontroller zwischen 291 und 306 Einheiten, um das Signal an Prozeß B auszuliefern. Dies entspricht einer Realzeit von $\sim$316 μs und $\sim$332 μs. Man erkennt aber auch innerhalb des Meßintervalls von 313 Sekunden drei Einbrüche der Signallaufzeit. An diesen Stellen erfolgte die Weiterleitung des Signals schon in 36 bzw. 38 Einheiten, was 39 μs bzw. 41 μs entspricht. Beim Scheduling zur Laufzeit handelt es sich zwar um einen indeterministischen Vorgang. Aufgrund der überschaubaren Prozeßmenge wäre hier eine konstante Laufzeit ohne Ausreißer zu erwarten gewesen. Denkbar wäre allerdings auch, daß im Quelltext der Laufzeitumgebung oder des SDL-Systems noch kritische Abschnitte verborgen sind. Innerhalb von kritischen Abschnitten werden alle Interrupts unterdrückt, jedoch das Auftreten gespeichert. Tritt innerhalb von kritischen Abschnitten ein Interrupt mehrfach auf, wird dieser nach Ende des Abschnitts nur einmal ausgeführt. Für den Timer bedeutet dies, läuft ein kirtischer Abschnitt länger als 2048 Takte (256 Ticks), wird nur ein Timerinterrupt statt zwei registriert. Die Zeit ist dementsprechend 256 Ticks zu langsam gelaufen. Berücksichtigt man diesen Sachverhalt bei den Ausreißern, und addiert zu ihnen 256 hinzu, liegen die Werte mit 292 und 294 im gleichen Bereich wie die anderen.

Interessant ist insgesamt die recht hohe Laufzeit von mindestens 291 Einheiten, wenn man die Ausreißer unberücksichtigt läßt. Dies entspricht immerhin 2.328 Takten ($\sim$316 μs), die der Controller benötigt, um die Daten von einem Prozeß zu einem

anderen zu verschicken. Diese Zeiten sind zu hoch. Der Scheduler muß aufgrund dieser Messungen zusätzlich noch auf Effizienz untersucht werden, um die Gründe hierfür zu finden.

4.1.2 Daten aus SDL an Umgebung senden

Der Controller muß nicht nur intern Daten an Prozesse schicken, sondern auch mit der Laufzeitumgebung interagieren. Es wurde somit ermittelt, wieviel Zeit vergeht vom Zeitpunkt der Sendeanforderung bis zur tatsächlichen Übertragung auf dem Medium. Zu diesem Zweck wurde in einem einfachen Prozeß periodisch die aktuelle Systemzeit in ein Signal für den Transceiver verwandelt. Die Schnittstelle in der Laufzeitumgebung protokollierte die Zeit, zu der das Signal eintraf. Eine weitere Zeit wurde ermittelt, sobald der Transceiver den *Start of Frame Delimiter* gesendet hat.

Die Hardware selbst benötigt für die Umschaltung in den Sendemodus 192 μs. Zusätzlich kommen noch 128 μs für die Präambel sowie 32 μs für die Übertragung des SFD selbst dazu. Es dauert somit mindestens 352 μs, bis der SFD übertragen ist. Zusätzlich kommen noch die Verzögerungen durch den Scheduler hinzu. Anschließend wurde der Sendezeitpunkt davon abgezogen und die Werte auf den beiden seriellen Schnittstellen ausgegeben. Eine zweite Messung zeigt auch das Ende der Übertragung. Hierzu erfolgte die Berechnung erst, sobald die Übertragung beendet wurde. Die Daten und das Meßsystem hierzu sind in Abschnitt E.1 auf Seite 63 zu finden.

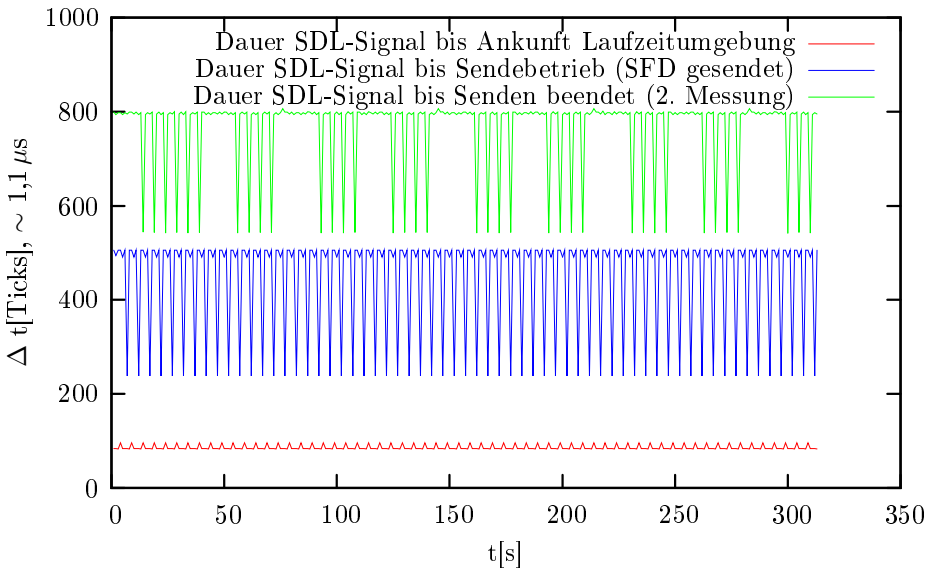

Abbildung 4.2: Signallaufzeit von SDL in Laufzeitumgebung bzw. bis Sendebeginn des Transceiver-Chips.

In Abbildung 4.2 sind beide Meßwerte graphisch dargestellt. Überraschend ist hier, daß Signale an die Laufzeitumgebung deutlich schneller verarbeitet werden, als dies durch Signale innerhalb des SDL-Systems geschehen ist. Zwischen 83 und 96 Einheiten (~90,0 μs und ~104,2 μs) benötigt ein Signal demnach, um aus SDL in die

Laufzeitumgebung zu gelangen. Die Laufzeit schwankt nur zwischen den beiden angegebenen Schranken und enthält keine Ausreißer wie in Abbildung 4.1. Die Zeit, die vergeht, bis der *Start of Frame Delimiter* gesendet wurde, ist als mittlerer Meßwert gezeigt. Hier sind deutliche Schwankungen zu erkennen. Die Werte schwanken zwischen 238 und 506 Einheiten ($\sim$258,2 μs und $\sim$549,0 μs), dies entspricht einer Differenz von 268 Einheiten. Hier könnte es sich um das gleiche Problem mit einem Timer-Interrupt, wie in Abschnitt 4.1.1, handeln. Werden die gemessenen Werte, die stark abweichen um den Wert von 256 Einheiten korrigiert, fügen sich die Werte in das normale Spektrum ein und die Schwankung liegt nur noch zwischen 494 und 506.

Damit benötigt der Transceiver 444,8 μs, bis der SFD gesendet wird. Rein rechnerich ergibt sich mit den Zahlen aus Abbildung 3.4 auf Seite 22 ein Wert von 352 μs, bis der SFD gesendet wird. Berücksichtigt man nun die Berechnug der CRC und der Paketlänge sowie das Schreiben des FIFOs, bevor das Paket gesendet werden kann, scheint der ermittelte Wert realistisch. Die obere Kurve repräsentiert die Gesamtzeit der Übertragung seitdem das Signal von SDL gesendet wurde und zeigt eine ähnliche Schwankung wie die untere Kurve. Die Meßwerte sind aus einer weiteren Messung entstanden, so daß sich hier die Schwankungen nicht direkt miteinander vergleichen lassen. Der Meßwert von maximal 807 Einheiten (Ausreißer unberücksichtigt), was einer Zeit von 874 μs entspricht, paßt zu den berechneten Werten aus der gleichen Abbildung. Die übertragenen Daten sind mit 4 Byte anzusetzen, hinzu kommen weitere 6 Byte für Länge, FCF, DSN und CRC, so daß insgesamt 10 Byte übertragen werden müssen. Die Zeit läßt sich damit als $10 \cdot 32\,\mu s = 320\,\mu s$ berechnen. Addiert man den Wert zu der Zeit für das SFD, ergibt sich $444,8\,\mu s + 320\,\mu s = 764,8\,\mu s$. Nicht enthalten ist die Zeit, die das Signal aus dem SDL System benötigt. Diese maximal 104,2 μs, verrechnet mit den 764,8 μs, ergeben 873 μs, was dem gemessenen Wert entspricht.

4.1.3 Signallaufzeit aus Laufzeitumgebung bis zur Ankunft im Prozeß

Signale werden nicht nur innerhalb von SDL, sondern auch in die Umgebung verschickt. Die Messung dieses Verhaltens wurde bereits im letzten Abschnitt beschrieben. Zusätzlich dazu werden aus der Umgebung auch Signale empfangen. Für diese Signale ist die Zeit, die bis zur Verarbeitung in SDL benötigt wird, interessant. Zu diesem Zweck wurde ein einfacher Prozeß modelliert, der aufgrund eines ankommenden Signals die aktuelle Systemzeit abzüglich der Zeit, die im Signal übergeben wurde, an die serielle Schnittstelle schickt. In der Laufzeitumgebung wurde dazu in der Empfangsroutine des Transceivers nicht das empfangene Paket, sondern die aktuelle Systemzeit an SDL geschickt. Damit ist der Empfangszeitpunkt des Pakets in SDL bekannt. Als Sender kommt dazu ein weiterer Knoten mit dem Programm aus dem vorherigen Abschnitt zum Einsatz. Alle benutzten Komponenten, der Quelltext und die Meßwerte sind im Anhang E.2 auf Seite 65 zu finden.

In Abbildung 4.3 auf der nächsten Seite sind die Daten dieser Messung aufgetragen. Exemplarisch ist in diesem Diagramm als zweite Kurve ein Korrekturwert eingezeichnet, der das Problem in den Meßdaten durch eine Korrektur von 256 Einheiten auf

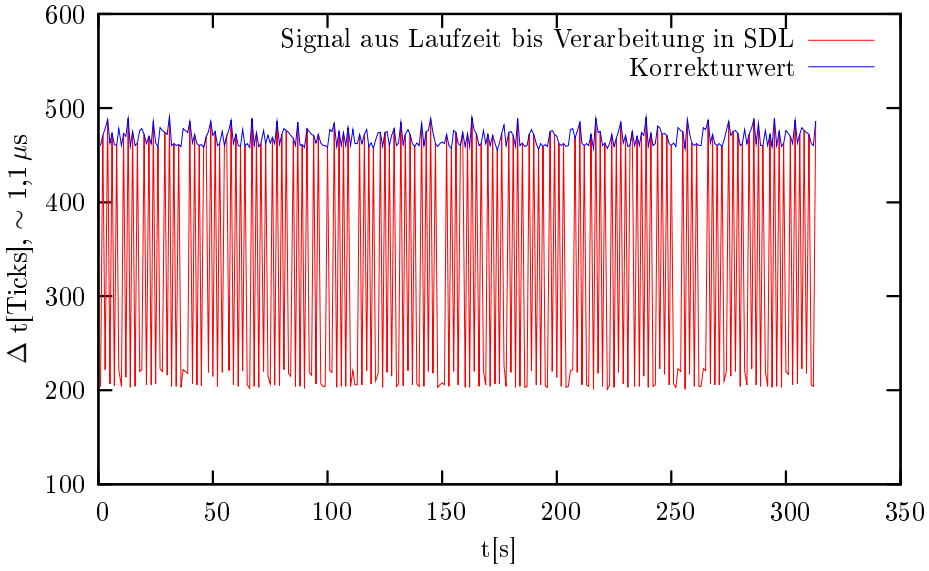

Abbildung 4.3: Signallaufzeit von Laufzeitumgebung bis Verarbeitung in SDL

ein „normales“ Niveau verschiebt (siehe Abschnitt 4.1.1). Der Wert entspricht dem Wert, der bei Überlaufen des Interrupts auf die interne Variable addiert wird. Betrachtet man also die so korrigierten Werte, schwanken diese nur zwischen 458 und 491 Einheiten (∼496,6 µs und ∼532,4 µs). Innerhalb dieser Zeit werden durch die `memlib.c` vier Byte Speicher für den aktuellen Zeitstempel reserviert. Die restliche Zeit wird danach wohl durch den SDL-Scheduler verbraucht.

4.1.4 Uhrendrift und Timergenauigkeit

Daten wie die Uhrendrift zu untersuchen, scheint aufgrund des Fehlers, der bereits vorher aufgetreten ist, nicht weiter interessant zu sein. Unter Uhrendrift versteht man die Abweichung der Zeit verschiedener Knoten. Doch diese Untersuchung kann, selbst wenn sie diesen Fehler nicht aufdeckt, zumindest einen Beleg für das Auftreten geben. Aus diesem Grund wurde auch diese Messung durchgeführt. Generell ist es wichtig zu untersuchen, wie stark die Uhren von verschiedenen unabhängigen Systemen voneinander abweichen. Ist ein Zeitpunkt auf beiden Systemen festgelegt, zu dem das eine System empfängt, während das andere sendet, sollten beide Systeme auch rechtzeitig bereit sein. Für diese Steuerung ist eine periodische Synchronisation nötig, dessen Intervall sich nach der Genauigkeit der einzelnen Systeme richtet. Zu diesem Zweck ist das in Abbildung E.3 auf Seite 67 gezeigte SDL-System spezifiziert worden, das zusammen mit dem System aus 4.1.2 auf Seite 35 verwendet wird. Das System empfängt dabei lediglich die Zeit und speichert die aktuelle Zeit als weiteren Wert. Beide Daten werden unverändert über die beiden seriellen Schnittstellen ausgegeben.

Zunächst wurde aus den Daten die Abweichung der Normsekunde ausgerechnet. Eine Normsekunde beschreibt hierbei die Anzahl an Ticks, die vom Mikrocontroller in einer Sekunde gezählt wurden. Bei einem Takt von 7.372.800 Hz sind dies dann

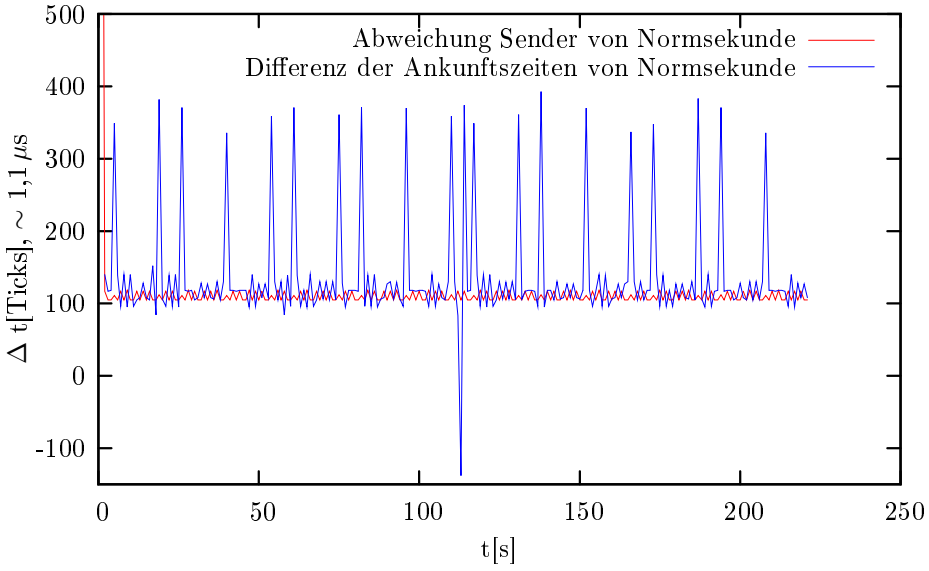

Abbildung 4.4: Kontinuität der Uhr des Senders und des Empfängers

genau 921.600 Ticks. Der Sender sollte jede Sekunde durch das Setzen eines Timers ein Signal mit seiner Zeit schicken. Es ist somit zu erwarten, daß die Differenz zwischen zwei ankommenden Zeitsignalen der Normsekunde entspricht. Es läßt sich damit die Genauigkeit der Timer von SDL bestimmen. Auf Seiten des Senders wird, wie in Abbildung 4.4 zu sehen ist, der Timer ca. 100 Einheiten zu spät bearbeitet. Auf Seiten des Empfängers ist nun nicht die Zeit eines Timers aufgetragen, sondern die Ankunftszeit der jeweiligen Pakete. Da die Signale auf Senderseite im Sekundenrhythmus gesendet werden, müßten sie demnach auch im Sekundenrhythmus ankommen. Die Differenz zur Normsekunde ist als blaue Kurve aufgetragen. An dieser Stelle tritt vermutlich wieder der Effekt der aus Abschnitt 4.1.3 bekannten Signallaufzeit-Probleme auf. Die Verzögerung, bis ein Signal gesendet ist sowie die eigentliche Signalübertragung kann als konstant angenommen werden, so daß hier für die blaue Kurve (mit den Ausreißern nach oben) zwar eine höhere Verzögerung gilt, diese aber in ähnlichem Maß schwanken sollte wie die rote.

Abbildung 4.5 zeigt nun die Zeitdifferenz zwischen dem Sender und dem Empfänger. Bestünde das Problem mit den Interrupts nicht, wäre hier eine konstante Linie zu erwarten, aus der sich, sofern beide Geräte gleichzeitig gestartet wurden, die Übertragungszeit für das Paket von Sender zu Empfänger bestimmen ließe. Da genau dies die Intention dieser Messung war, wurde über eine Steckerleiste mit Schalter bei beiden Knoten die Stromversorgung gleichzeitig eingeschaltet. Es ist daher davon auszugehen, daß beide Knoten im selben Moment gestartet wurden, und beide Timer gleichzeitig bei 0 angefangen haben zu zählen.

Die Abbildung zeigt stattdessen eine Treppenfunktion, die die bereits getroffene Annahme bestätigt. Eine andere mögliche Erklärung wäre eine sehr starke Drift zwischen beiden Knoten, die aber durch die vorherigen Messungen mit den Schwankungen widerlegt werden kann. Da beide Knoten außerdem Quarzbausteine zur Versorgung des Timers benutzen, die in der gleichen Umgebung betrieben werden, ist eine Schwankung in dieser Größenordnung auch nicht zu erklären.

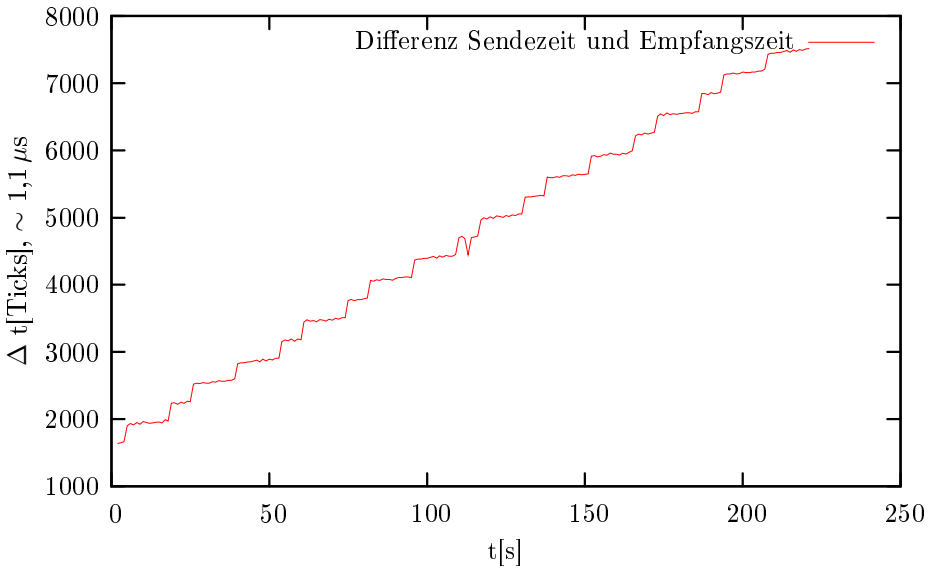

Abbildung 4.5: Uhrendrift

Ein weiterer Test, der eine bessere Genauigkeit der Uhrendrift erreicht, wäre obiges Programm auf zwei Knoten laufen zu lassen und einen weiteren Knoten im Sekundentakt ein Signal senden zu lassen. Durch diesen Test ließe sich die Drift sehr genau bestimmen. Bevor dies allerdings durchgeführt wird, müßte das Interrupt-Problem, das sich auf zu lange laufende kritische Abschnitte zurückführen läßt, erst behoben werden.

4.2 Energieverbrauchsbetrachtung

An dieser Stelle wird die Laufzeit eines MICAz unter Betrachtung einiger Szenarien aufgezeigt.
Da der ATMega 128L mindestens 2,7 V benötigt, ist es ausgeschlossen, Standard-Akkumulatoren zu verwenden. Diese liefern in der Bauform AA nach kurzer Zeit nur noch 1,2 V pro Stück. Hersteller von Alkali-Batterien geben nicht an, wie groß die Kapazität ihrer Batterien ist, oder wie die Entladekurve aussieht. In der Zeitschrift c't [Fab02] wurden Ergebnisse veröffentlicht, die als Anhaltspunkt dienen können. In diesem Test wurden die Batterien bis auf eine Spannung von 0,9 V bei einem konstanten Strom von 150 mA entladen. Die Kapazität der getesteten Batterien schwankte zwischen 1,85 mAh und 2,09 mAh. Da die minimale Spannung pro Zelle 1,35 V beträgt und der Stromverbrauch des Knotens kleiner ist als 150 mA, sind diese Zahlen nicht brauchbar.

Im Internet findet sich auf den Seiten der ETH Zürich [Zin] eine Entladekurve (siehe Abbildung 4.6 auf der nächsten Seite), die ich hier als Anhaltspunkt verwende. In der Abbildung läßt sich für 1,35 V ein Wert von ca. 5 Stunden für Alkaline ablesen, was bei einer Entladung von 120 mA einer nutzbaren Kapazität von $5\,\mathrm{h} \cdot 120\,\mathrm{mA} = 600\,\mathrm{mAh}$ entspricht. Für die e2-Lithium Batterie von Energizer [Ene] ergibt sich analog eine nutzbare Kapazität von $23\,\mathrm{h} \cdot 120\,\mathrm{mA} = 2.760\,\mathrm{mAh}$.

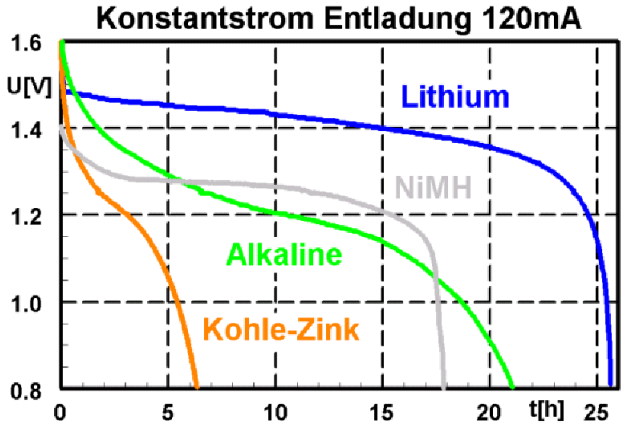

Abbildung 4.6: Entladekurve verschiedener Batteriesorten aus [Zin].

Bei einer Worst-Case-Abschätzung aller Komponenten, wie sie in Kapitel 2 beschrieben sind, ergibt sich ein Stromverbrauch von

$$7{,}5\,\mathrm{mA} + 19{,}7\,\mathrm{mA} + 35\,\mathrm{mA} = 62{,}2\,\mathrm{mA}.$$

Um kleinere Komponenten und sonstige Stromverluste mitzuberücksichtigen, über die es keine genaueren Angaben gibt, schlagen wir hier noch weitere 5 % auf und kommen damit auf einen Stromverbrauch von 65,31 mA. Der Aufschlag von 5 % ist bewußt hoch für die verbleibenden Komponenten getroffen. Für die Alkaline-Batterien ergibt sich somit eine maximale Laufzeit von

$$\frac{2 \cdot 600\,\mathrm{mAh}}{65{,}31\,\mathrm{mA}} = 18{,}4\ \mathrm{h},$$

für die e2-Lithium-Batterie hingegen eine maximale Laufzeit von

$$\frac{2 \cdot 2760\,\mathrm{mAh}}{65{,}31\,\mathrm{mA}} = 84{,}5\ \mathrm{h}.$$

In dieser Rechnung ist nun allerdings der serielle Datenspeicher mit durchgehendem Lesebetrieb und der Transceiver mit durchgehendem Sendebetrieb betrachtet. Beides sind unrealistische Annahmen für den Anwendungsfall. Für einen realistischen Anwendungsfall, der keine Energiespar-Modi berücksichtigt, können wir von einem durchgehenden Empfangsbetrieb und 5 % Sendebetrieb ausgehen. Damit ergibt sich ein Stromverbrauch von

$$7{,}5\,\mathrm{mA} + 5\,\% \cdot 17{,}4\,\mathrm{mA} + 95\,\% \cdot 19{,}7\,\mathrm{mA} \approx 26{,}1\,\mathrm{mA},$$

zuzüglich der oben erwähnten 5 % Aufschlag ergibt sich

$$26{,}1\,\mathrm{mA} \cdot 105\,\% = 27{,}4\,\mathrm{mA}.$$

Für Alkaline-Batterien bedeutet dies eine maximale Laufzeit von 43,7 h bzw. 201,2 h für die e2-Lithium Batterie.

Die Laufzeit des Knotens bei aktiviertem Stromsparmodus läßt sich kaum abschätzen. Hierfür ist die genaue Kenntnis eines Protokolls nötig. Soll beispielsweise der Transceiver ausgeschaltet werden, um so die Laufzeit zu erhöhen, sind Synchronisationsmechanismen nötig, um Sender und Empfänger gleichzeitig schlafen und wieder aufwachen zu lassen. Die Laufzeit hängt schließlich sehr stark von den gewählten Ruhezeiten des Chips ab.

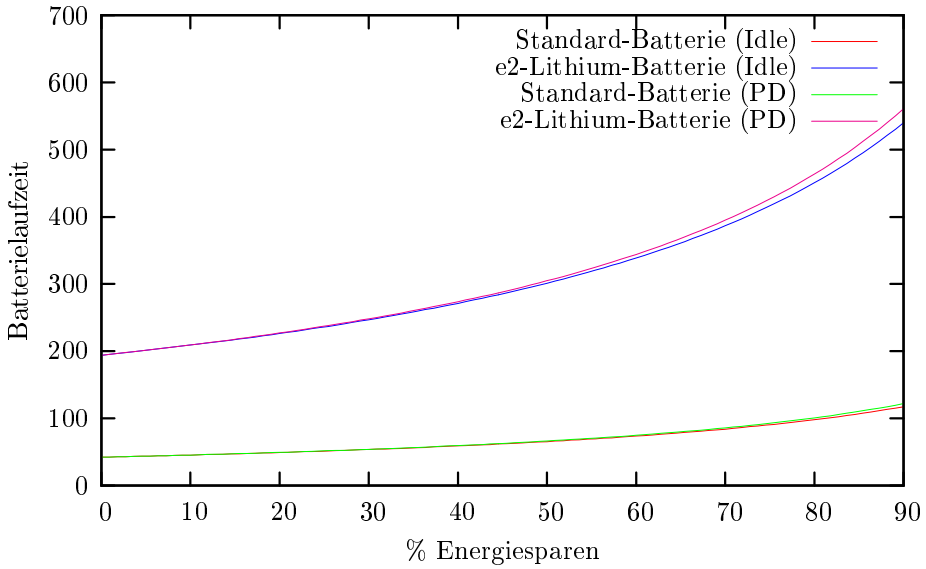

Abbildung 4.7: Batterielaufzeit bei 5% Sende- und 5% Empfangsbetrieb

In Abbildung 4.7 ist die Batterielaufzeit beider Batteriezellen exemplarisch für 5% Sende- und mindestens 5% Empfangsbetrieb angegeben. Dargestellt ist hierbei die Zeit, die der Chip im *Idle*-Modus ist. Im Falle von 0% Energiesparen bedeutet dies, der Chip befindet sich zu 5% der Zeit im Sendemodus und weitere 95% im Empfangsmodus. Für den Fall 90% sind es nur noch 5% Empfangs- und 5% Sendebetrieb, der Rest der Zeit ist er *Idle* oder im *Power-Down*-Modus. Diese Abbildung soll hierbei veranschaulichen, welche Laufzeiten mit Batterien möglich sind, sofern der Transceiver in einen Stromsparmodus geschaltet wird. Bei den Standard-Zellen ist fast kein Unterschied zwischen dem *Power-Down*-Modus und dem *Idle*-Modus festzustellen. Bei den e2-Lithium-Zellen ist der Unterschied zwar nicht groß, aber durchaus festzustellen. Generell erhöht sich durch das Stromsparen die Laufzeit der e2-Lithium-Zellen von etwa 200h auf über 600h!

Kapitel 5

Entwicklung einer MAC-Schicht

MAC-Schichten gibt es für verschiedenste Zugriffsmedien und Anforderungen. Einschränkungen können sich durch domänenspezifische Anforderungen wie Energieeffizienz oder Größe des Systems ergeben. MAC-Protokolle für mobile Knoten müssen im Gegensatz zu kabelgebundenen Systemen auf Energieeffizienz achten. Um die Funktion der Laufzeitumgebung zu zeigen, soll eine minimale Anforderung genügen. Das hier vorgestellte Protokoll ist nur ein einfaches Beispiel für eine MAC-Schicht, das keine spezifischen Anforderungen von mobilen Knoten berücksichtigt.

5.1 Anforderungen

Das Beispielprotokoll für den MICAz muß folgende Anforderungen erfüllen:

- Medienzugriff mit Kollisionsvermeidung
- Prioritätsbasiertes Senden
- Fairneß, kein Aushungern von Knoten

Der Medienzugriff mit Kollisionsvermeidung sorgt dafür, daß ein bereits belegter Kanal nicht erneut belegt wird und, sofern er frei ist, nicht alle Stationen gleichzeitig mit dem Senden beginnen. Prioritätsbasiertes Senden ist eine Vorstufe des *Quality of Service* und beschreibt, daß Pakete mit höherer Priorität auf dem Medium zu bevorzugen sind. Durch diese Bevorzugung kann es zur Aushungerung von Knoten kommen, wenn diese nur Nachrichten mit niedriger Priorität versenden wollen, während ein anderer Knoten mit Nachrichten höherer Priorität den Kanal blockiert.

5.2 Analyse

Die Steuerung des Medienzugriffs mit Kollisionsvermeidung erledigt die Hardware beim CC2420 teilweise eigenständig. Der Zustand des Mediums kann über Signale an das SDL-System weitergeleitet werden. Problematisch ist das gleichzeitige Senden von zwei oder mehr Stationen. Während des Sendens kann der Transceiver-Chip

das Medium nicht abhören und deshalb eine Kollision nicht feststellen. Dies ist nicht für den Transceiver des MICAz spezifisch, sondern stellt ein generelles Problem von drahtlosen Netzwerkkomponenten dar. Ein weiteres Problem ergibt sich aus der Umschaltzeit zwischen Empfangen und Senden. In dieser Zeit ist der Transceiver in einem Zwischenzustand, in dem er nicht sendet, aber auch nichts empfangen kann. Zur Lösung des Problems werden zufällige *Backoff*-Zeiten definiert, die ein Knoten wartet, bevor er versucht, das Medium zu belegen, was auch als *Carrier Sense Multiple Access / Collision Avoidance* (CSMA/CA) [IEE03] bezeichnet wird. Damit wird sichergestellt, daß nicht alle sendewilligen Knoten gleichzeitig senden. Ist das Medium belegt, wird die eigene Backoff-Zeit auf den vorherigen Wert zurückgesetzt. Kollisionen lassen sich zwar hiermit nicht gänzlich vermeiden, die Wahrscheinlicheit des Auftretens wird jedoch reduziert.

Die Anforderung des prioritätsbasierten Sendens läßt sich durch die Verwendung von unterschiedlich langen Backoff-Zeiten regeln. Dazu definiert man einen Offset, auf den die zufällige Wartezeit aufsummiert wird. Nachrichten mit hoher Priorität haben einen kleinen, Nachrichten mit niedriger Priorität einen hohen Offset. Damit der Kanal möglichst gut ausgenutzt wird, entfällt bei freiem Medium die zufällige Wartezeit und nur bei belegtem Kanal wird diese hinzugenommen.

Für die Fairneß ist das gerade dargestellte Verfahren ein Problem. Ein Knoten kann ausgehungert werden, sofern er nur Nachrichten mit niedriger Priorität versenden möchte, ein anderer Knoten währenddessen den Kanal mit Nachrichten höherer Prioriät belegt. Mit jeder Nachricht wird die Backoff-Zeit des ersten Knotens auf seinen Ausgangswert gesetzt, und dieser kann nie senden. Eine Lösung des Problem ist, die Backoff-Zeit nicht jedesmal auf den Offset inklusive der Zufallszahl zurückzusetzen. Läßt man die Zeit allerdings einfach nur weiterlaufen, erhöht sich die Gefahr einer Kollision auf dem Medium. Da sich eine Kollision nie ganz vermeiden läßt, genügt dieser Ansatz der Forderung von Fairneß.

5.3 Design

Die Struktur der MAC-Schicht ist in Abbildung 5.1 gezeigt. Es handelt sich hierbei um den Block, der für die MAC-Schicht zuständig ist. Diese Implementierung benutzt den CC2420Driver, der Signale der Umgebung auf interne SDL-Signale umsetzt. Anstelle der Bezeichnung ZIGBEE vor dem Signalnamen ist hier die Bezeichnung PHY zu finden. An dem Signal selbst ändert sich sonst nichts. Die Anwendungsschicht kennt nur zwei Signale, mac_recv und mac_send. Das Signal mac_recv wird von der MAC-Schicht geschickt, sobald ein Paket mit gültiger CRC empfangen wurde. Das Signal mac_send wird von der Anwendung an die MAC-Schicht gesendet. Es beinhaltet die zu sendenden Daten sowie einen Prioritätswert. Der Prozeß MacProc stellt die zentrale Komponente dar. Sie übernimmt die Steuerung der zu sendenden Pakete und reicht ankommende Pakete, sofern die CRC-Prüfung positiv ausfiel, an die Anwendungsschicht weiter. Der Prozeß MacQueue dient als FIFO-Warteschlange für Sendewünsche. Der Beginn einer Backoff-Phase wird dem Prozeß Backoff mitgeteilt. Nach Ablauf der Backoff-Zeit schickt dieser ein Signal an den Prozeß MacProc.

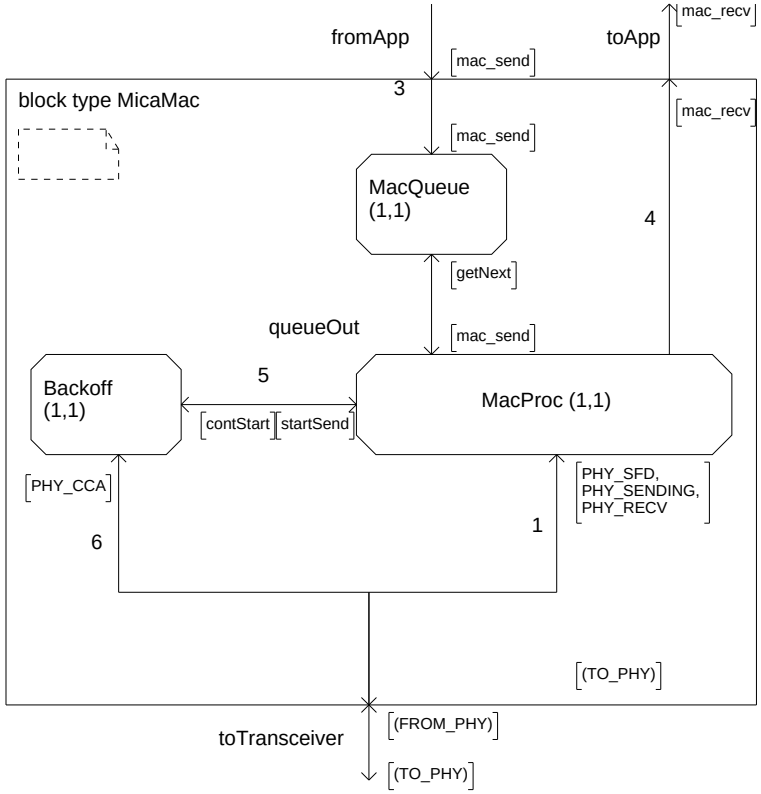

Abbildung 5.1: Block der Beispiel-MAC-Schicht

In Listing 5.1 auf der nächsten Seite ist das Verhalten von Prozeß MacQueue in Pseudocode gezeigt. Die SDL-Darstellung ist in Abbildung H.3 auf Seite 79 dargestellt. Wird von der Anwendung ein Sendewunsch geschickt und ist der Prozeß nicht beschäftigt (busy), so wird er direkt an den MacProc gesendet. Geschieht dies aber während der Prozeß beschäftigt ist, wird das Signal in eine FIFO-Queue gespeichert. Sobald der MacProc mit einem Signal getNext anzeigt, daß er neue Aufträge entgegennehmen kann, wird, falls vorhanden, ein Sendewunsch aus dem FIFO herausgenommen und an den Prozeß MacProc geschickt (Zeile 11).

Der Prozeß MacProc ist als Pseudocode in Listing 5.2 auf der nächsten Seite dargestellt. Die SDL-Modellierung ist in Abbildung H.2 auf Seite 78 zu finden. Jedes Signal PHY_RECV wird als mac_recv an die Anwendung weitergegeben, sofern die CRC-Prüfung erfolgreich war. Sobald ein Signal mac_send von dem Queue-Prozeß eintrifft, wird der Prioritätswert an den Backoff-Prozeß weitergeleitet. Trifft das Signal startSend ein, werden die zu sendenden Daten mittels PHY_SEND_CCA an den Transceiver weitergeleitet und von der Queue neue Daten mit getNext angefordert.

Der Backoff-Prozeß (Listing 5.3 auf Seite 47) ist vom Zustand des Mediums, der ihm mit dem Signal PHY_CCA mitgeteilt wird abhängig. Ist das Medium frei, enthält die Variable busy den Wert **false**, andernfalls **true**. Abhängig vom Zustand des Mediums bzw. dieser Variablen erhält ein Sendewunsch eine unterschiedliche Wartezeit. Ist der Kanal frei, wird die Anzahl an Warteslots auf die Priorität des Sendewunsches gesetzt. Es wird bei freiem Medium somit nicht unnötig lange gewartet. Die Wahrscheinlichkeit einer Kollision steigt an dieser Stelle jedoch. Ist das Medium be-

```
while(true){
        if(fromApp.mac_send & !busy){
                signal queueOut.mac_send
                busy:=true
        }else
                FIFO.queue(fromApp.mac_send)
        if(MacProc.getNext}{
                if(FIFO.isempty)
                        busy:=false
                else{
                        signal FIFO.dequeue(queueOut.mac_send)
                        busy:=true
                }
        }
}
```

Listing 5.1: Pseudocode für Prozeß MacQueue.

```
while(true){
        if(toTransceiver.PHY_RECV & (CRC = OK) )
                signal toApp.mac_recv
        if(queueOut.mac_send)
                signal contStart(prio)
        if(startSend){
                signal PHY_SEND_CCA
                signal getNext
        }
}
```

Listing 5.2: Pseudocode für Prozeß MacProc.

legt, während die Backoff-Wartezeit gesetzt wird, so ergibt sich die Wartezeit aus der Priorität und einer zusätzlichen zufälligen Zeit. Ist das Medium frei, wird bei Beginn des nächsten Slots die Variable counter, in der die Backoff-Zeit gespeichert ist, heruntergezählt. Hat sie den Wert 0 erreicht, wird das Signal startSend an den Prozeß MacProc gesendet, der nun die wartenden Daten sendet.

5.4 Test des Systems

Nach der Spezifikation des Systems in SDL wurde der erzeugte Code zusammen mit der Laufzeitplattform auf den MICAz geladen. Die Funktion konnte damit getestet werden. Leider stürzte das System nach kurzer Zeit ab, was sich nach einiger Debug-Arbeit als größeres Problem herausstellte. Manche Funktionen werden mit falschen Parametern aufgerufen. Der Fehler kann sich beispielsweise durch eine eingeschaltete rote LED äußern, obwohl diese gar nicht im Programm eingeschaltet wurde. Die Ursache dafür ist bis jetzt noch nicht lokalisiert. Mögliche Fehlerquellen können hier überschriebene Rücksprungadressen oder die fehlerhafte Anbindung einer Funktion an das SDL-System sein. Sonderbar ist hier vor allem, daß die Interruptbehandlung nicht weiterläuft. Ohne weitere Hardware-Debugger wird es wahrscheinlich schwer, den Fehler einzugrenzen. Denn nur ein solcher Debugger ermöglicht zum Zeitpunkt

```
1  while(true){
2          if(PHY_CCA)
3                  busy:=false
4          else
5                  busy:=true
6          if(contStart){
7                  if(busy)
8                          counter:= prio * CW + RANDOM
9                  else
10                         counter:= prio
11         }
12         if(counter>0 & nextSlot & !busy){
13                 counter:=counter -1
14                 if(counter = 0)
15                         signal startSend
16         }
17 }
```

Listing 5.3: Pseudocode für Prozeß Backoff.

des Fehlers die Ausgabe des kompletten Inhalts des Arbeitsspeichers und der Register.

Kapitel 6

Fazit und Ausblick

In dieser Arbeit wurde für den MICAz eine Laufzeitplattform erstellt, um in SDL spezifizierte Applikationen auszuführen. Die Spezifikation mit Hilfe von SDL kann die Entwicklung deutlich beschleunigen. Zusätzlich erhält man ein formales Modell, das sich mit eingebauten Werkzeugen auf Fehler untersuchen läßt. Durch die Verwendung der Schnittstellen des *SDL Environment Frameworks* ist eine Anbindung des in SDL spezifizierten Systems an den *ns*+SDL-Simulator möglich. Dadurch läßt sich das System ohne MICAz testen. Aus SDL können Daten an andere Knoten über den Transceiver verschickr und empfangen werden. Die Kommunikation erfolgt transparent mit Signalen aus SDL. Die Steuerung von transceiverspezifischen Funktionen wie Sende- und Empfangskanal oder Sendestärke ist ebenfalls möglich. Um den hohen Stromverbrauch des Transceivers zu senken, steht ein weiteres Signal bereit. Außerdem sind zwei serielle Schnittstellen aus SDL erreichbar, die einen direkten Datenaustausch mit angeschlossenen Geräten ermöglichen. Die drei Leuchtdioden, die als einfache Statusanzeige dienen, stehen ebenfalls über Signale zur Verfügung.
Kapitel 4.1 hat gezeigt, daß in einfachen Systemen die Laufzeit von Signalen relativ konstant ist. Dies ist eine Grundvoraussetzung für die Vorhersagbarkeit und Simulation des Verhaltens. Für eine lange Laufzeit des mobilen Knotens wurden einige Abschätzungen in Kapitel 4.2 angesprochen. Das Beispielsystem aus Kapitel 5 zeigt, wie einfach sich die Programmierung mit Hilfe der Laufzeitumgebung gestaltet.

An diese Arbeit müssen anschließend noch zwei Fehler gefunden und behoben werden. Der eine besteht in sporadischen Programmabstürzen, wobei die Ursache noch nicht genau geklärt ist. Der Fehler äußert sich beispielsweise durch eine eingeschaltete rote LED, obwohl das Programm diese an keiner Stelle einschaltet. Der zweite Fehler betrifft die Systemzeit. In Kapitel 4.1 wurde bereits angesprochen, daß ein Timer-Interrupt nicht auslöst und dadurch der Makrotakt nicht erhöht. Hierbei handelt es sich vermutlich um eine Funktion, in der alle Interrupts deaktiviert sind, und deren Laufzeit länger ist als die Zeit zwischen zwei Makrotakten. Sobald die Systemzeit wieder zuverlässig arbeitet, sollte eine Messung der Uhrendrifft von verschiedenen MICAz-Knoten durchgeführt werden. Diese Drifft wird benötigt, um Synchronisationszeitpunkte für die Knoten zu bestimmen. Ein einzelner Knoten als Sender kann periodisch Daten senden, während alle anderen dieses Signal empfangen. Jeder Emp-

fänger gibt seine Systemzeit auf der seriellen Schnittstelle aus. Würden alle Uhren synchron laufen, müßte die Zeitdifferenz zwischen zwei empfangenen Paketen auf allen Knoten gleich groß sein.

Weiterhin könnten die hohen Verzögerungen bei der Signalweiterleitung (Kapitel 4.1) genauer untersucht werden. Die verwendete Speicherbibliothek weist, je nach Auslastung und Fragmentierung, starke Schwankungen bei der Laufzeit aus. Ein Algorithmus mit nahezu konstanter Laufzeit wäre wichtig. Soll der MICAz im mobilen Umfeld und mit Batterien betrieben werden, sollten zusätzlich die Stromsparmodi des Mikrocontrollers genutzt werden. In Verbindung mit SDL wären dafür allerdings Eingriffe in den Scheduler nötig.

Einige Funktionen des Mikrocontrollers, die vornehmlich für Sensornetzwerke Verwendung finden, werden derzeit nicht genutzt. Hierzu zählt der EEPROM-Speicher, der es erlaubt, Daten dauerhaft abzulegen. Dieser Speicher ist direkt in den Mikrocontroller integriert und kann mit einem Takt Verzögerung angesprochen werden. Aufgrund der begrenzten Lebensdauer von 10.000 Schreiboperationen kann man diesen nicht als RAM-Erweiterung verwenden. Weiterhin fehlt die Ansteuerung der analogen und digitalen Meßeingänge für externe Signale. Mittels der *Capture-Compare*-Funktion läßt sich so das Überschreiten eines vorgegebenen Wertes prüfen. Mit den externen Ausgängen des Chips können zusätzlich Steuer- und Regelungsaufgaben erledigt werden. Zum Speichern der ermittelten Meßwerte können diese entweder über den Transceiver weitergeben oder in den noch nicht verwendeten statischen Speicher geschrieben werden.

Anhang A

Erstellen einer Arbeitsumgebung für den MICAz

Für die Arbeit mit dem MICAz ist die Einrichtung einer Arbeitsumgebung nötig. Standardmäßig geschieht dies auf einem Windows-PC. Da die UNIX-Umgebung Cygwin [cyg] zum Einsatz kommt, sollten sich die Pakete auch auf einem Linux-PC mit dem Paketverwaltungsprogramm `rpm` einrichten lassen.

A.1 Einrichten von TinyOS

1. Die neueste Version von TinyOS [Tin] besorgen.

2. Die Installation nach der Anleitung durchführen (normalerweise nur weiterklicken), bei Linux mit `rpm -ivh` installieren.

3. Häufig ist die Installationsversion nicht die neueste Version von TinyOS. Deshalb von der TinyOS Homepage [Tin] das Update Paket herunterladen und nach Anleitung installieren.

4. Sobald Cygwin gestartet ist, in das Verzeichnis `/opt/tinyos-1.x/apps/Blink` wechseln. Nach erfolgreicher Installation läßt sich mit Befehl `make micaz` das Programm kompilieren.

5. Nun den PC mit dem Programmierboard verbinden. Ein einfaches Board wird per paralleler Schnittstelle programmiert. Die evtl. vorhandenen seriellen Anschlüsse werden für die Kommunikation des MICAz, nicht aber zur Programmierung verwendet.

6. Wenn alles angeschlossen ist, sollte ein `make mica2 install` nach einigen Schritten den MICAz Knoten zum Blinken bringen. Ist dies nicht der Fall, sind Änderungen an der Datei `/opt/tinyos-1.x/apps/Makerules` erforderlich. Hier muß die Programmiermethode auf Parallel stehen. Das heißt, nach Aufrufen des Befehls `make micaz install` sollte als Ausgabe `uisp -dprog=dapa -wr_fuse_h=0xd9 -dpart=ATmega128 -wr_fuse_e=ff -erase` auftauchen.

7. Jetzt die Laufzeitumgebung der Diplomarbeit in das Verzeichnis `/opt/tinyos-1.x/apps/` kopieren oder unter TinyOS einen Link in diesem Verzeichnis anlegen.

8. SDL sollte jetzt für die Arbeit eingerichtet und konfiguriert werden. Dazu wird das Verzeichnis `kernel_support` in einen Ordner innerhalb von TinyOS kopiert, z. B. `/opt`.

9. In Tau SDL den Targeting Expert starten, und unter Target Library das Memory management einschalten. Speichern. In Compiler/Linker/Make unter *Source Files* die Einträge:

   ```
   /opt/kernel_support/env.c
   /opt/kernel_support/mk_cpu.c
   /opt/kernel_support/mk_stim.c
   /opt/kernel_support/mk_user.c
   ```

   hinzufügen.

10. Zum Ausführen ist es jetzt nur noch nötig, innerhalb des SDL-Verzeichnisses ein `make` und im Laufzeitumgebungs-Verzeichnis ein `make clean install` aufzurufen. Damit wird dann der Rest kompiliert, gelinkt und auf den Knoten hochgeladen.

Falls es Probleme bei der Einrichtung geben sollte, ist die komplette, bei der Arbeit verwendete, TinyOS/cygwin-Umgebung in `tinyOS\cygwin.zip` abgelegt. Da es schwer ist, alle Parameter im Targeting Expert einzustellen und bei der Beschreibung keinen auszulassen, ist im Verzeichnis `Targeting-Expert` die zur Entwicklung der Middleware verwendete Konfiguration abgelegt. Die Dateien sind in das Verzeichnis `SDL-Projekt\Target._NummerDesSystems` des zu entwickelnden SDL-Projekts zu schreiben. Target bezeichnet den Namen des Systems, das kompiliert werden soll. Die Nummer des Systems gibt die Position des Systems im Organzier (Angefangen bei 0) multipliziert mit 1000 wieder. Darin enthalten sind mindestens zwei Dateien: `Application_CM.uis` und `sdttaex.ini`. *Zu beachten*: auf dem verwendeten System liegen die Dateien auf Laufwerk `H:\src\sdl\micaz\kernel_support`, außerdem ist die Runtime von Tau aus Cygwin unter `/opt/tau` zugänglich. Die Pfade müssen entsprechend auf das verwendete System angepaßt werden.

A.2 Anpassungen am Makefile

Alle im vorherigen Abschnitt angesprochenen `make`-Kommandos werden mit Hilfe des im gleichen Verzeichnis befindlichen `Makefile` interpretiert. Für den Installationsprozeß ist die Datei `Makefile.initial` zuständig. In ihr ist als Pfad für die TinyOS-Tools `$(TOSROOT)/tools/make/` eingetragen, der auf allen Systemen nach

einer korrekten Einrichtung stimmen sollte. Sofern ein eventuell falscher Pfad korrigiert ist, sind hier keine Änderungen mehr nötig. An der Datei `Makefile` müssen je nach eingesetztem System weitere Änderungen durchgeführt werden. Die wichtigste Variable, die auf das SDL-System eingestellt werden muß, ist hierbei `SDL_DIR`. Als Parameter ist der vollständige Pfad anzugeben. Sollte cygwin eingesetzt werden, ist für ein anderes Laufwerk, z.B. `h:`, `/cygdrive/h/` anzugeben. In der Variablen `C_OBJECTS` können weitere C-Dateien angegeben werden, die kompiliert und zu dem Projekt gelinkt werden. Alle restlichen Einstellungen sollten keine weitere Anpassung erfordern.

Die Operationen, die mittels des Makefiles durchgeführt werden, sind:

- `all` bzw. keine Angabe: Es werden alle Quellen kompiliert (Laufzeitumgebung und SDL-System). Die Quellen der Runtime werden allerdings nicht vorher auf Veränderungen untersucht. Es kann vorher ein `cleanrun` nötig sein.
- `sdl`: Alle SDL-Sourcen werden kompiliert.
- `install`: Schließt `all` ein, danach wird das Programm auf den Mikrocontroller geladen.
- `clean`: Entfernt alle kompilierten Dateien.
- `cleanrun`: Entfernt alle kompilierten Dateien der Laufzeitumgebung.
- `cleansdl`: Entfernt alle kompilierten Dateien des SDL-Systems.

Anhang B

Einschränkungen des Cmicro - Compilers

Der Cmicro-Compiler von Tau SDL [TAU] besitzt einige Einschränkungen, die man beachten muß, wenn man damit ein SDL-System entwirft. Der Cmicro-Compiler wird benutzt, um Code für den Mikrocontroller zu erzeugen. Die folgenden Punkte dürfen in einem SDL-System, das mit Cmicro übersetzt werden soll, **nicht** verwendet werden.

- Prozeduren
  - Vererbung von Prozeduren
  - Prozeduren mit Zuständen
  - Remote Procedure Calls
  - bei verschachtelten Prozedur-Aufrufen (Rekursive Aufrufe) wird die Variablensichtbarkeit ignoriert
- Timer
  - mit mehr als einem Parameter
  - deren Parameter nicht **Integer** sind
  - mit einer Dauer als Kommazahl (Diese Einschränkung gilt für die hier entwickelte Laufzeitumgebung nicht. Die Zeit wird in Ticks übergeben. Um dies lesbar zu gestalten, existiert eine Prozedur, die Sekunden und Nanosekunden in die geforderten Ticks umrechnet.)
- Input/Output
  - Weglassen von Parametern im Signal-Eingang
  - Output **via all**
  - Enabling Condition
  - Continuous Signals (Signale, die solange erzeugt werden, bis die Bedingung wahr ist)

  - Prozesse mit nicht spezifizierter Anzahl von Instanzen, wie (x,) oder (,)
  - Export / Import (Variablen aus anderen Prozessen)
  - View / Reveal
  - FPARS (Formal Context Parameters)

- Datentypen und Operatoren
  - Datentypen, die nicht im Targeting Expert eingeschaltet wurden, stehen nicht zur Verfügung.
  - die Datentypen **Array**, **String**, **Powerset**, **Bag**, **Ref** sollten wegen des Verbrauchs an dynamischem Speicher vermieden werden.
  - Datentypen, die auf **Charstring** basieren, werden als **char*** abgebildet.
  - Read und Write Funktionen und der **Q**-Operator werden nicht unterstützt bzw. ignoriert.
- der **any** Ausdruck

Anhang C

Probleme mit der Hard-/Software

In diesem Kapitel gehe ich kurz auf einige Probleme ein, die bei der Arbeit mit der Hard- bzw. Software auftreten können und zeige mögliche Lösungen auf.

Nach dem Reboot des PC arbeitet der Knoten nicht mehr.

Problem:
Das Programmierboard ist an den PC über die parallele Schnittstelle angeschlossen. Der MICAz-Knoten ist auf dem Board angeschlossen. Nach dem Starten des PCs arbeitet der Knoten nicht mehr.
Lösung:
Der PC initialisiert während des Bootens seine Schnittstellen neu. Aufgrund der Spannungen an der parallelen Schnittstelle wird der MICAz angehalten. Um das Problem zu beheben, trennen Sie einfach das parallele Kabel von dem Anschluß des Programmierboards. Danach führt der Knoten das zuletzt geladene Programm wieder aus.
In manchen Fällen führt auch diese Vorgehensweise nicht zum Erfolg. Es kann durchaus helfen das Board ohne Stromversorgung zu programmieren und den Verify-Vorgang abzubrechen. Danach Stromversorgung wieder anschließen und erneut programmieren.

Das Board läßt sich nicht programmieren.

Problem:
Das Board ist korrekt an den PC angeschlossen und auch mit Strom versorgt, aber beim Programmieren erscheint die Meldung `flash error at address`.
Lösung:
Dieses Problem tritt besonders nach Neustarts des Rechners auf. Scheinbar gibt es auch hier Probleme mit der Initialisierung der Schnittstelle. Als Lösungsmöglichkeit hat sich bewährt, alle Komponenten noch einmal auseinanderzunehmen, also den

Knoten vom Board zu trennen, die Spannungsversorgung und auch das parallele Kabel neu einzustecken. Hilft auch dies nicht, sollte man probieren, erst einmal einen anderen Knoten zu programmieren. Anschließend läßt sich der ursprüngliche meist auch programmieren.

SDL erzeugt Code, der Mikrocontroller arbeitet nicht korrekt.

Problem:
Es wurde ein System in SDL spezifiziert, und der Targeting Expert erzeugt daraus Code. Nach dem Hochladen auf den Mikrocontroller passiert nicht das Erwartete.
Lösung:
Wurden in SDL Code-Blöcke verwendet? Wurden Pointer direkt an Methoden aus der Laufzeitumgebung übergeben? Sollte dies der Fall sein, überprüfen Sie, ob in der `memlib.h` der Schalter SAVE_FREE eingeschaltet ist. Dieser sorgt dafür, daß die *free*-Funktion auch für eSpeicher, der nie allokiert wurde, funktioniert.
Falls dies nicht der Fall ist, sollten in der Laufzeitumgebung `SDLM.nc` verschiedene Debug-Schalter ausprobiert werden, um so das Problem einzugrenzen. Es sollte zunächst mit dem Debuggen des Timers DEBUG_TIMER bzw. DEBUG_TIMER_PRINT begonnen werden.

Die serielle Schnittstelle gibt eigenartige Daten zurück.

Problem:
Sie debuggen gerade und geben z. B. die aktuelle Zeit mit DEBUG_TIMER_PRINT auf die serielle Schnittstelle aus. Die Daten, die dort ankommen, ergeben keinen Sinn.
Lösung:
Falls Sie z. B. die Funktion *sdl_main()* auskommentiert haben, wird die Funktion *sdl_init_env()* nicht aufgerufen. Deshalb ist die Übertragungsrate nur bei 57,6 kBaud und nicht bei 115 kBaud, wie dies nach der Initialisierung durch SDL üblich ist. Ein klares Indiz dafür ist: Sie senden aus dem System zwei Bytes, es kommen jedoch manchmal zwei und manchmal drei Bytes an Daten im Terminal-Programm an.

Anhang D

SDL-System

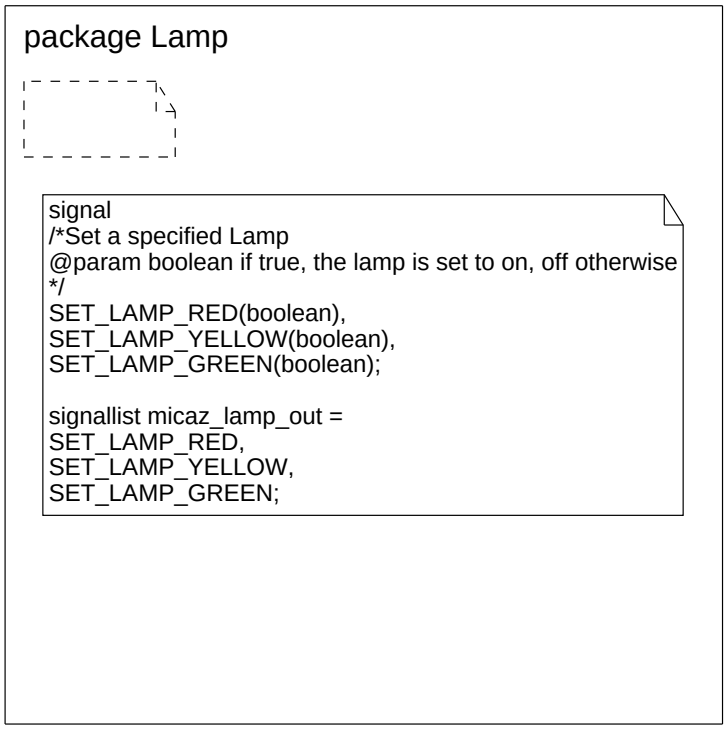

Abbildung D.1: LED-Interface aus SDL

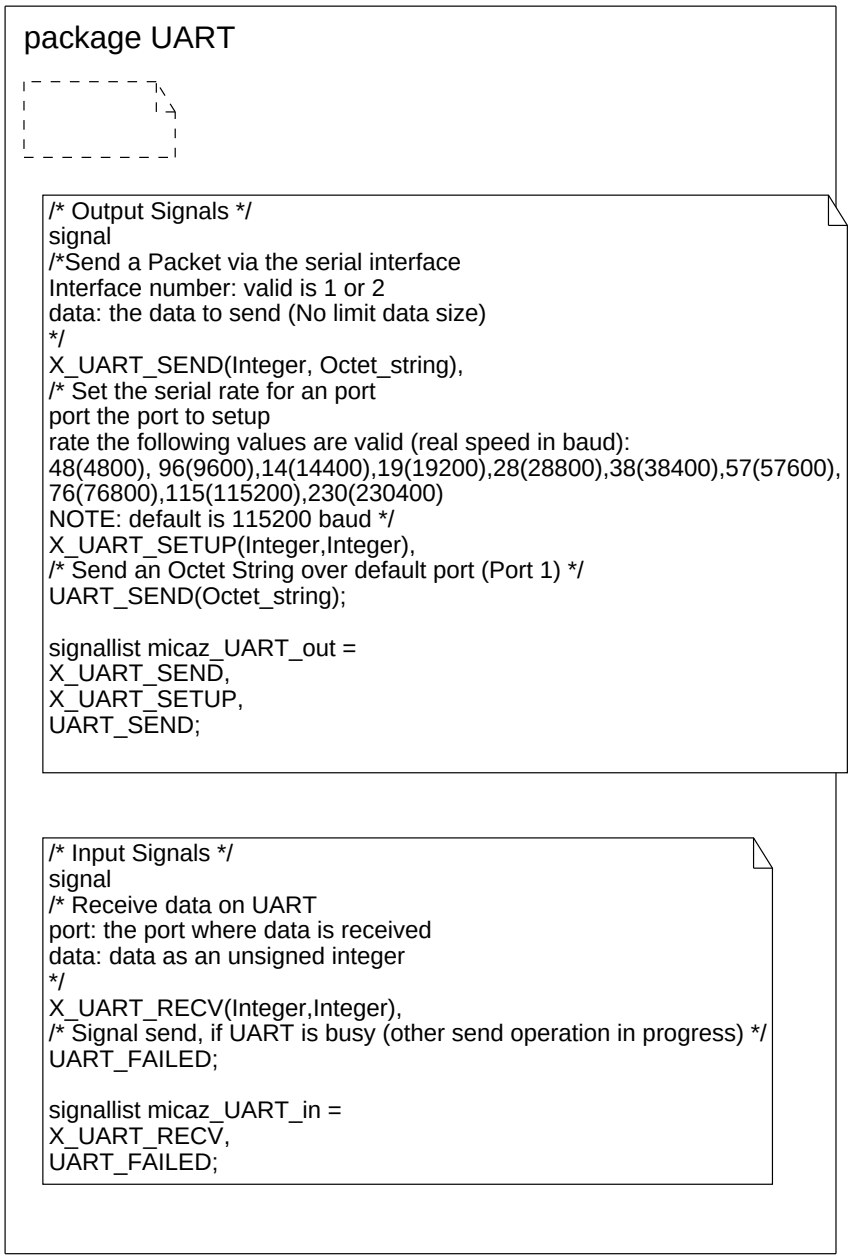

Abbildung D.2: UART-Interface aus SDL

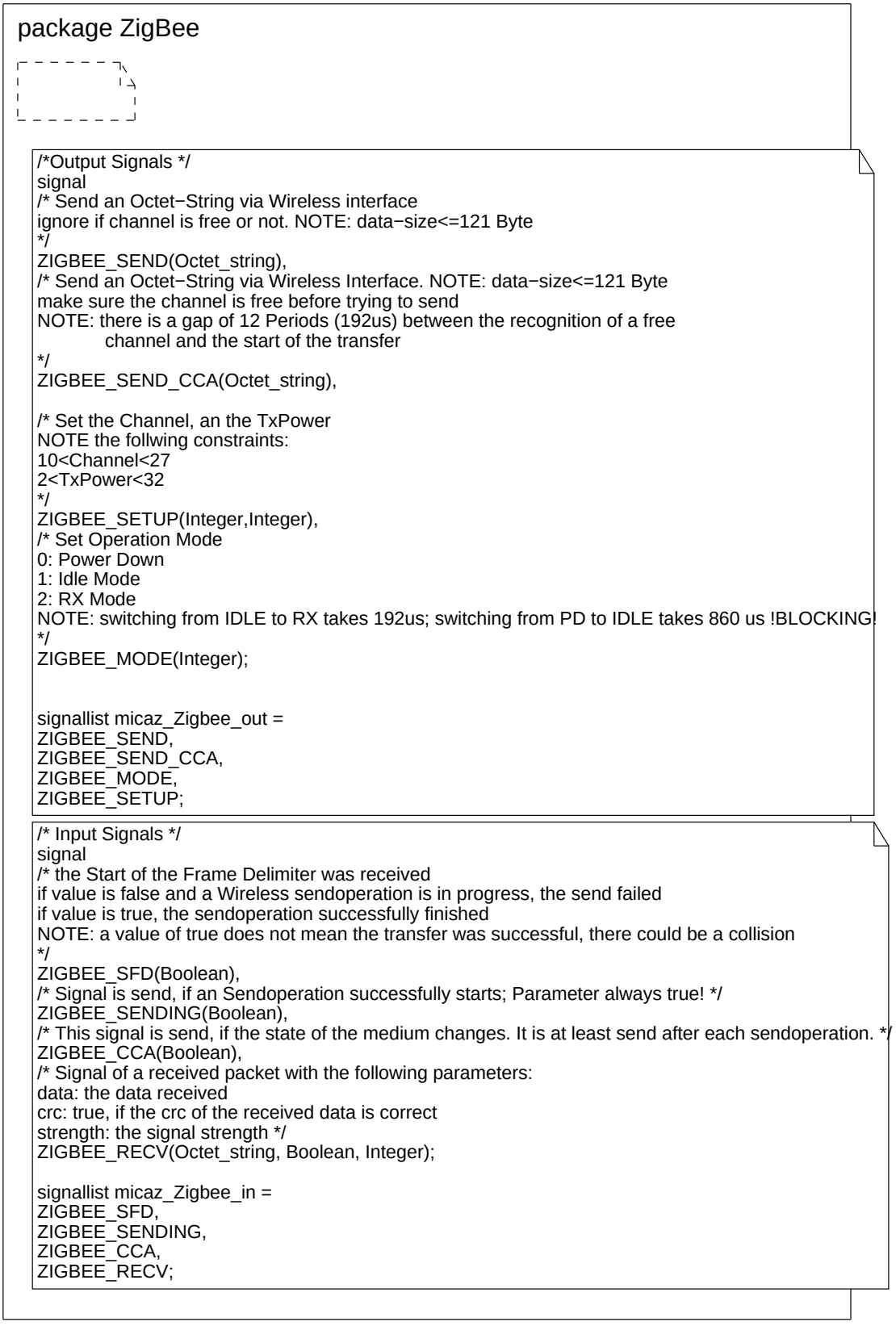

Abbildung D.3: Transceiver-Interface aus SDL

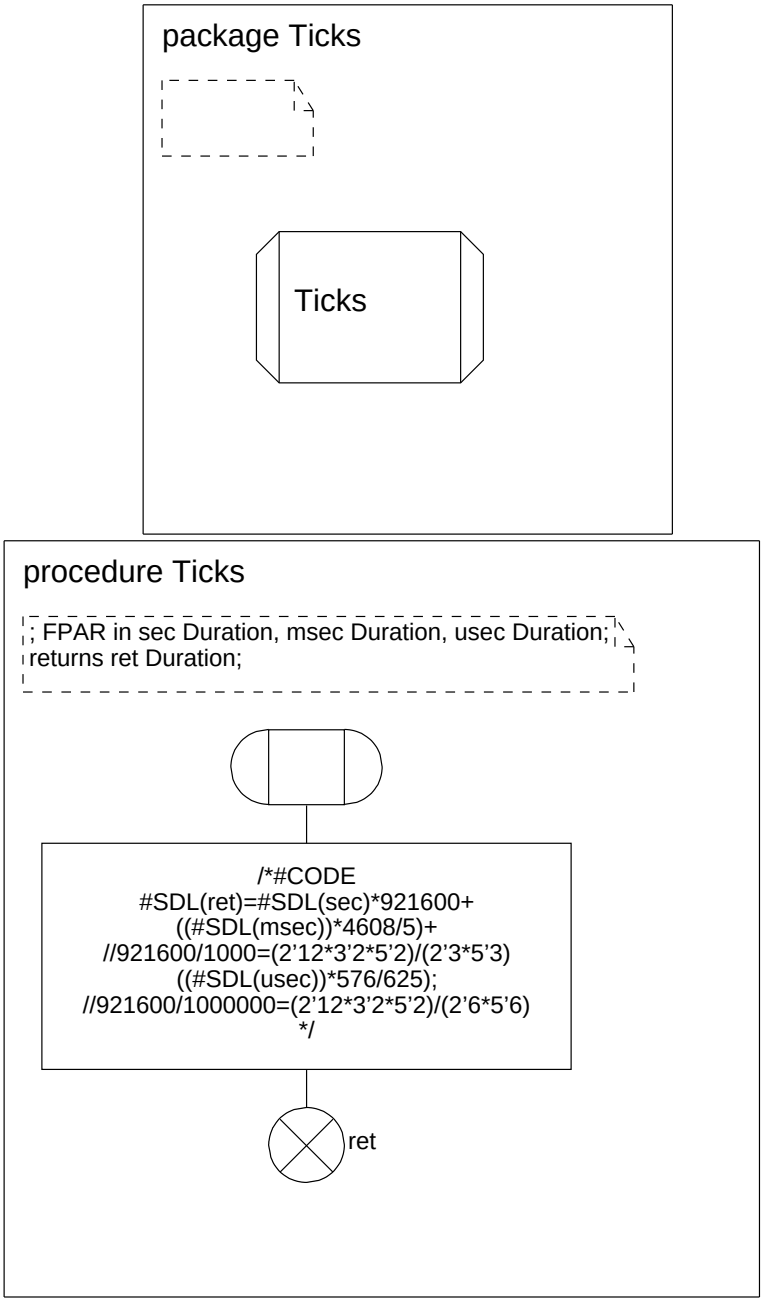

Abbildung D.4: Ticks-Interface aus SDL, zur Steuerung des Timers

Anhang E

Auswertungsdaten und Meßmethoden

Dieses Kapitel dient dazu, die in Kapitel 4.1 dargestellten Daten zu belegen. Es werden die jeweiligen Meßsysteme dargestellt. Auf der beigelegten CD finden sich die Meßdaten im Verzeichnis `TimingDaten`.

E.1 Daten aus SDL an Umgebung senden

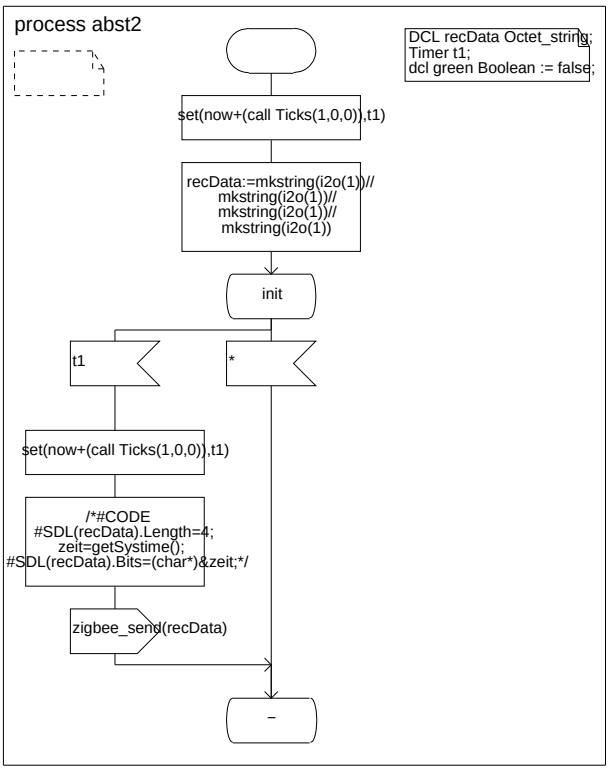

Abbildung E.1: Testsystem 1, Prozeß.

Diese Messung soll zeigen, wie lange ein Signal aus SDL benötigt, bis es in der Laufzeitumgebung ankommt. Alle Meßdaten sind in der Datei `test1.csv` zu finden. Diese Zeit in Ticks ist in der zweiten Spalte dargestellt. Die erste Spalte zeigt lediglich die Messung, die im Sekundenabstand durchgeführt wurde. In der dritten Spalte ist die Zeit gezeigt, die benötigt wird, bis ein Sendewunsch an den Transceiver ausgeführt wurde. Es handelt sich hierbei um die Zeit, die seit Absenden in SDL bis zum Auftreten des SFD vergangen ist. In Abbildung E.1 auf der vorherigen Seite ist der Prozeß, der für diese Messungen benutzt wurde gezeigt. Es handelt sich hierbei um einen einfachen Prozeß, der die aktuelle Zeit in einem **Octet_string** verpackt und über den Transceiver versendet. Es wurde hierbei bewußt der Aufruf *getSystime()* der Laufzeitumgebung und nicht **now** verwendet, um eine möglichst genaue Messung zu erhalten. Der für die Messung benötigte Quelltext ist in Listing E.1 angegeben. Während des Tests ist das Flag TIMING_TEST_SEND gesetzt.

Die zweite Messung wurde getrennt von der ersten durchgeführt und zeigt die Zeit, nach der das Paket vollständig übertragen wurde.

```
491     bool sendWirelessString(char* string, int length, bool CCA){
492         uint8_t i;
493 #ifdef TIMING_TEST_SEND
494         t1=getSystime();
495         sendTime=*((xmk_T_TIME*)string);
496 #endif
497         if(length>MAX_WIRELESS_LEN)return FALSE;
498 
499         atomic{
500             for(i=0;i<length;i++){
501                 sendMsg.data[i]=string[i];
502             }
503             sendMsg.length = length;
504         }
505 #ifdef DEBUG_WIRELESS
506         // on send toggle green
507         ledsCtrl(0,2);
508 #endif
509         return call WirelessSend.send(&sendMsg, CCA);
510     }
511 
512     event TOS_MsgPtr Receive.receive(TOS_MsgPtr m){
513          int len, i;
514          char *data;
515 #ifdef TIMING_TEST_SEND
516                  t2=getSystime();
517                  t1=t1-sendTime;
518                  t2=t2-sendTime;
519                  sendUARTString((char*)&t1,4,1,FALSE);
520                  sendUARTString((char*)&t2,4,2,FALSE);
521 #endif //end of TIMING_TEST_SEND
```

Listing E.1: Quelltext-Auszug für die Messung zu Testsystem 1 (aus `SDLM.nc`).

E.2 Signallaufzeit aus Laufzeitumgebung bis zur Ankunft im Prozeß

Die folgenden Meßwerte geben die Laufzeit eines in der Laufzeitumgebung erzeugten Signals bis zu deren Verarbeitung in SDL an. Die zugehörigen Meßwerte sind in der Datei `test3.csv` zu finden. Dazu wurden das in Abbildung E.2 auf der nächsten Seite spezifizierte System und der in Listing E.2 angegebene Quelltext verwendet, in dem das Flag TIMING_TEST_RECV definiert ist. Die Zeit t* gibt hierbei den korrigierten Wert für das Timing an. Es wurde bei jedem Wert, der „unrealistisch“ schien, also dessen Wert kleiner als 400 war, 256 zur Korrektur addiert.

```
512     event TOS_MsgPtr Receive.receive(TOS_MsgPtr m){
513           int len, i;
514           char *data;
515 #ifdef TIMING_TEST_SEND
516                   t2=getSystime();
517                   t1=t1-sendTime;
518                   t2=t2-sendTime;
519                   sendUARTString((char*)&t1,4,1,FALSE);
520                   sendUARTString((char*)&t2,4,2,FALSE);
521 #endif //end of TIMING_TEST_SEND
522
523 #ifdef DEBUG_WIRELESS
524           //on receive toggle green
525         ledsCtrl(0,2);
526 #endif
527
528 //FIXME: hier werden daten kopiert, anstatt pointer zu verbiegen
529
530 atomic{
531 #ifdef TIMING_TEST_RECV
532 //this is used for testing purpose only!
533           len=4;
534           data=memoryAlloc(4);
535           *(long*)data=getSystime();
536
537 #else
538           len=(m->length - MSG_HEADER_SIZE - MSG_FOOTER_SIZE);
539           data=memoryAlloc(len);
540           for(i=0;i<len;i++){
541                 data[i]=m->data[i];
542           }
543 #endif
544 }
545           ZIGBEE_Recv(data, len, m->crc, m->strength);
546         return m;
547     }
548 }
```

Listing E.2: Quelltext-Auszug für die Messung zu Testsystem 3 (aus `SDLM.nc`).

```
473  async event void SFDCapture.captured(uint16_t time) {
474      call SFDCapture.disableEvents();
475  #ifdef TIMING_TEST_SEND
476      signal Receive.receive(rxbufptr); //use this signal just temporary
477  #else
```

Listing E.3: Quelltext-Auszug für die Messung zu Testsystem 3 (aus `WirelessM.nc`).

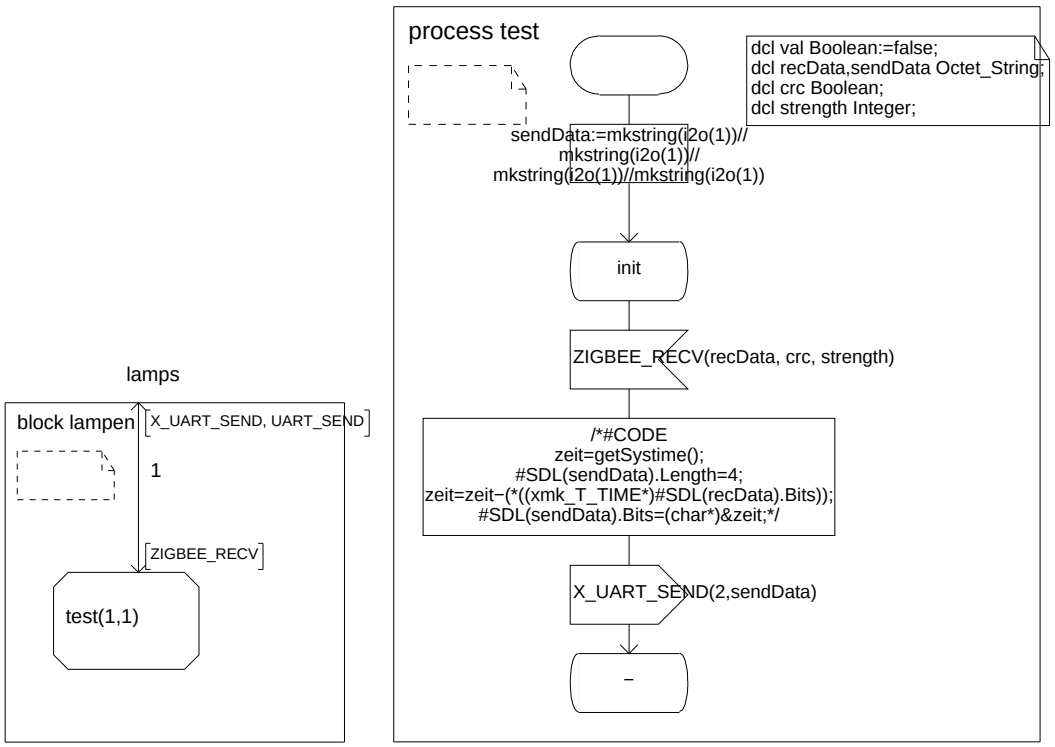

Abbildung E.2: Testsystem 3, Block und zugehöriger Prozeß.

E.3 Uhrendrift und Timergenauigkeit

Die folgenden Werte geben einen Anhaltspunkt zur Genauigkeit von Timern und der Drift der unabhängigen Uhren unterscheidlicher Knoten. Alle Meßwerte sind in Datei `test4.csv` zu finden. Dazu wurde auf einem Knoten das System aus E.1 eingesetzt und auf dem Meßsystem das SDL-System aus Abbildung E.3 eingesetzt. In der Tabelle sind beide Werte, sowie die Differenz zum vorherigen Wert angegeben. Zusätzlich ist für beide Werte die Abweichung zur Normsekunde von 921.600 aufgeführt.

E.4 Interne Signallaufzeit zwischen SDL-Prozessen

Die zwischen zwei Prozessen auftretenden Signallaufzeiten wurden mit dem in Abbildung E.4 auf der nächsten Seite und Abbidlung E.5 auf Seite 68 gezeigten System durchgeführt und hierbei die Meßwerte, die in der Datei `test2.cvs` zu finden sind, ermittelt.

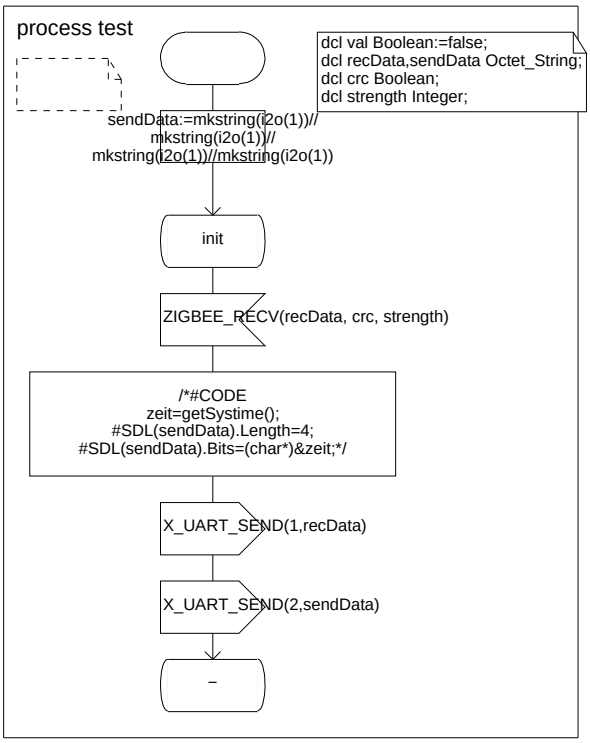

Abbildung E.3: Testsystem 4, Prozeß.

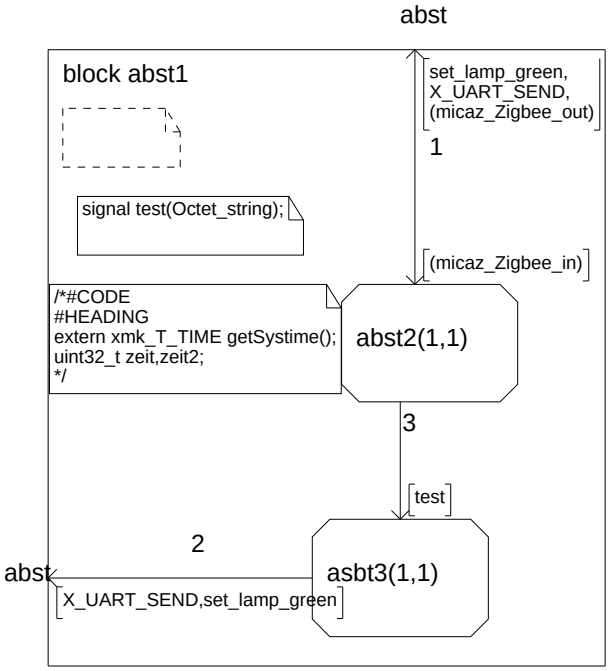

Abbildung E.4: Testsystem 2, Block.

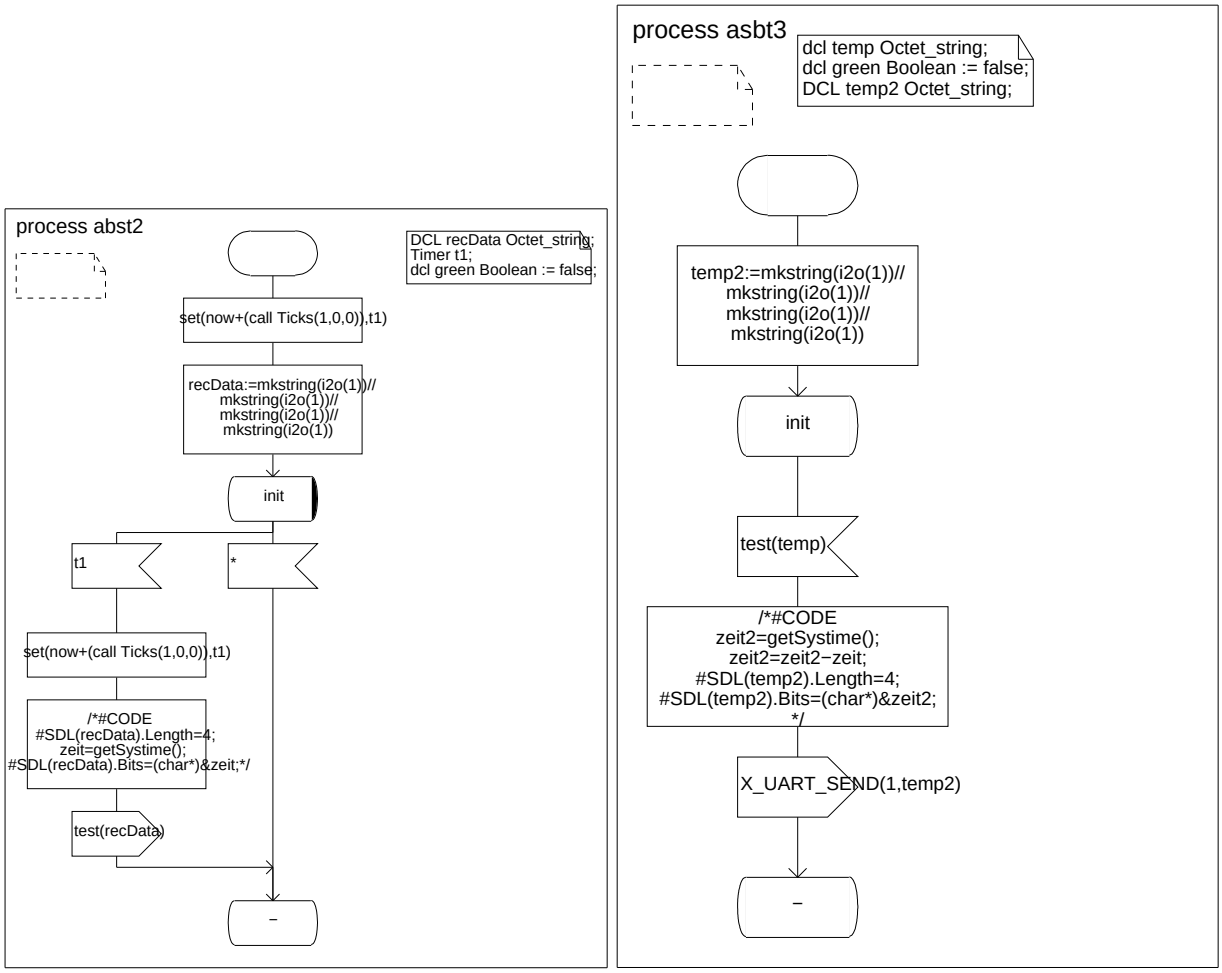

Abbildung E.5: Testsystem 2, Prozeß 1, Prozeß 2.

Anhang F

Alle technischen Daten im Überblick

Hier sind alle technischen Daten aus Kapitel 2 als Schnellreferenz noch einmal tabellarisch zusammengefaßt. Eine vollständige Aufstellung erfolgt an dieser Stelle nicht. Dafür sei auf die jeweiligen technischen Datenblätter verwiesen.

F.1 ATMEL ATMega 128L

Controller	8-Bit
RAM	4 KByte
ROM	128 KByte
EEPROM	4 KByte
Taktfrequenz	7,3728 MHz
Timer	Real Time Clock zwei 8-Bit Timer zwei 16-Bit Timer
serielle Schnittstelle	zwei UART-Schnittstellen (max. 256 kbps)
Betriebsspannung	2,7 V - 5,5 V
Stromaufnahme	Idle: 6,5 mA Last: 7,5 mA

F.2 Dallas DS2401

Seriennummer gesamt	64 Bit (s. Abb. 2.2, Seite 6
Family Code	fest `0x1h`, 8-Bit
Seriennummer	48 Bit
CRC	8 Bit

F.3 ATMEL AT45DB041B

Kapazität	528 KByte in 264 Seiten à 264 Byte
Betriebsspannung	2,5 V bis 3,6 V
Stromaufnahme	Lesen: 4 mA bis 10 mA Schreiben: 15 mA bis 35 mA
Zeit	Lesen: 300 μs Schreiben + Löschen: 20 ms

F.4 Chipcon CC2420

Übertragungsfrequenz	2,4 GHz bis 2,4835 GHz
Kanäle	11 bis 26
Übetragungsrate	2,4 kbps
Paketgröße	128 Byte
Puffer	ein Empfangspuffer (128 Byte) ein Sendepuffer (128 Byte)
Sicherheit	CRC-16 Einheit integriert eingebaute Encryption Unit
Betriebsspannung	2,1 V bis 3,6 V
Stromaufnahme	PowerDown: 20 μA Idle: 426 μA TX: 8,5 mA bis 17,4 mA RX: 19,7 mA
Zeit	PD $\rightarrow$ Idle: 860 μs Idle $\rightarrow$ RX/TX: 192 μs RX/TX $\rightarrow$ TX/RX: 192 μs

Anhang G

Schnittstellen

Wie bereits in Kapitel 3 erwähnt, gibt es drei Schnittstellen in der Laufzeitplattform. Die erste Schnittstelle ist zwischen dem TinyOS-Code und der Übergabe der Funktionspointer an die C-Datei `SDL2TOS.c` zu finden, die Zweite zwischen den SDL-Code-Dateien und der eben erwähnten C-Datei. Die letzte Schnittstelle findet sich schließlich innerhalb der SDL-Umgebung, die für die Anwendungsentwicklung gebraucht wird. Im folgenden werden alle Schnittstellen und ihre Parameter erklärt.

G.1 TinyOS-Schnittstelle

Allgemein werden alle Funktionspointer innerhalb der Datei `SDLM.nc` angelegt und auch initialisiert. Alle Zugriffe auf Funktionen innerhalb des TinyOS oder der Hardware werden über dieses Interface abgebildet. Alle Funktionspointer werden, wie in `SDL2TOS.h` definiert, angegeben:

*extern void (*SDLM$ledsPtr) (int op, uint8_t value)*
Kontrolliert die Leuchtdioden. Der Parameter **op** gibt den Modus an, wie die Lampen bzgl. des alten Wertes behandelt werden. **op**=0 bedeutet ein **XOR** mit dem vorherigen Wert, **op**=1 ein direktes Setzen des aktuellen Wertes, **op**=2 eine **OR**-Operation und 3 eine **AND** Verknüpfung. In **value** ist der Wert jeder Lampe kodiert. Bit 0 für rot, 1 für grün und 2 für gelb.

*extern void (*SDLM$setSystimePtr) (xmk_T_TIME value)*
Von SDL zwingend benötigte Funktion zum Setzen der Systemzeit, z. B. nach einem Überlauf der Variablen für die Zeit. **value** ist hier die neue Zeit, die gesetzt werden soll, wobei das letzte Byte ignoriert wird.

*extern void (*SDLM$initSystimePtr) ()*
Von SDL zwingend benötigte Funktion zum Initialisieren der Systemzeit. Die Zeit wird auf „0“ gesetzt.

*extern xmk_T_TIME (*SDLM$getSystimePtr) ()*
Von SDL zwingend benötigte Funktion zur Abfrage der aktuellen Zeit. Wird regelmäßig von der **now**-Funktion oder beim Setzen einer Timers von SDL aus aufgerufen.

*extern bool (*SDLM$sendUARTPtr) (char* data, int length, int port, bool freeMem)*
Kontrolle beider UART-Schnittstellen. **data** gibt die Daten, die verschickt werden sollen, an. **length** dient dazu, die Länge der Zeichenkette zu ermitteln, kann aber auch genutzt werden, falls nur Teile davon verschickt werden sollen. Falls andere Daten als Zeichen verschickt werden, ist darauf zu achten, daß der Mikrocontroller *Little-Endian* kodiert ist. Der Parameter **port** gibt an, an welchen Port die Daten gesendet werden. Port 1 wird mit 1, Port 2 mit 2 angesprochen. Beide Ports haben eine eigene Sendewarteschlange, so daß sie parallel genutzt werden können. Der letzte Parameter **freeMem** gibt an, ob der in **data** belegte Speicher nach dem Senden mittels der *memoryFree*-Methode (der Speicherbibliothek) freigegeben werden soll oder nicht. Falls kein Speicher reserviert wurde, dieser Parameter aber auf **true** gesetzt wurde, wird das Programm nicht weiterlaufen, sofern in der Datei `memlib.h` die Option SAVE_FREE deaktiviert ist.

*extern bool (*SDLM$sendWirelessPtr) (char* string, int length, bool CCA)*
Senden einer Zeichenkette über das Wireless-Interface. Hier gibt **string** die Zeichenkette an, die gesendet werden soll, **length** wie bei *sendUARTPtr* auch die Länge. **CCA** gibt hier zusätzlich an, ob das Senden unter Beachtung eines freien Kanals erfolgen soll.

*extern void (*SDLM$serialSetBaudPtr) (int port, int baud)*
Parameter der seriellen Schnittstelle einstellen. Pro Port kann, genau wie in *sendUARTPtr*, die Geschwindigkeit eingestellt werden. Um für den Parameter nicht unnötig Speicherplatz zu verschwenden, wurden für die Geschwindigkeiten 4800, 9600, 14400, 19200, 28800, 38400, 57600, 76800, 115200, 230400 Baud die Abkürzungen 48, 96, 14, 19, 28, 38, 57, 76, 115, 230 eingeführt.

*extern void (*SDLM$wirelessSetupPtr) (int channel, int txpower)*
Parameter für das Wireless-Interface setzen. Mittels **channel** wird der Kanal eingestellt, wobei $11 \leq$ **channel** ≤ 26 sein muß. **txpower** gibt die Sendeleistung an, mit der der Transceiver-Chip senden soll. Hier steht der Wert 31 für maximale und 3 für die minimale Sendeleistung, die einer Dämpfung des Signals von 25 dbm entspricht.

*extern bool (*SDLM$currentCCAPtr) ()*
Aktuellen Kanalstatus prüfen. Diese Funktion wird genutzt, um zu überprüfen, ob der Kanal belegt (Rückgabe **false**) oder nicht belegt (**true**) ist.

*extern void (*SDLM$setZigbeeModePtr) (int mode)*
Kontrolle des Transceiver-Zustands. Mit dieser Funktion kann der Transceiver in einen Stromsparmodus geschaltet und daraus auch wieder aufgeweckt werden. Der

Modus wird mit dem Parameter **mode** festgelegt. Der Wert 0 bedeutet, den Chip ganz auszuschalten, also soviel Strom wie möglich zu sparen. Die Aufwachzeit aus diesem Modus beträgt aber 860 μs und ist blockierend. Ein Wert von 1 schaltet ihn lediglich in den *Idle*-Modus, wobei hier die Stromersparnis nicht so groß ist, dafür aber auch die Aufwachzeit nur 192 μs beträgt. Jeder andere Wert schaltet den Transceiver wieder in den *Receive*-Modus. Die genauen Werte zur Ersparnis sind in Abschnitt 2.2.2, Seite 5, bereits angegeben.

extern uint8_t (*SDLM$getID_OR_CRCPtr) (bool ID)*
Eindeutige ID abrufen. Mit dieser Funktion lassen sich die eindeutige Seriennummer von 48 Bit oder die 8 Bit Prüfsumme (CRC) abrufen. Beide können zur Identifikation oder zum Initialisieren von Zufallszahlengeneratoren verwendet werden. Der Parameter **ID** gibt an, ob die ID (**true**) oder die CRC (**false**) zurückgegeben werden soll.

G.2 SDL-Environment-Schnittstelle

Bis auf die Receive-Funktionen *ZIGBEE_Recv* und *X_UART_Recv* sowie *ZIGBEE_Sfd* werden alle Funktionen, die mit dem SDL Environment kommunizieren, in der Datei `SDL2TOS.c` definiert. Hier eine kurze Beschreibung der Funktionen:

void UART_Setup(int port, unsigned int baud)
Diese Funktion ist genau wie *serialSetBaudPtr* aufgebaut.

void sdl_init_env()
Diese Funktion kann zum Initialisieren des Environments verwendet werden. Derzeit setzt diese Funktion die Übertragungsrate für beide seriellen Schnittstellen auf 115 kbps.

void setSystime(xmk_T_TIME value)
Diese Funktion ist genau wie *setSystimePtr* aufgebaut.

void initSystime()
Diese Funktion ist genau wie *initSystimePtr* aufgebaut.

xmk_T_TIME getSystime()
Diese Funktion ist genau wie *getSystimePtr* aufgebaut.

void Set_Lamps(int val)
Diese Funktion hat das gleiche Interface wie *ledsPtr*, stellt aber sicher, daß nur die letzten 3 Bit benutzt werden. Diese Funktion kann benutzt werden, um eine Zahl direkt auf den Lampen darzustellen, wobei dann natürlich nur die letzten 3 Bit angezeigt werden.

void Toggle_Lamps(int val)
Diese Funktion kann verwendet werden, um eine oder mehrere Leuchtdioden aufgrund eines Ereignisses blinken zu lassen. Bei der Programmierung könnte man den letzten Zustand der Leuchtdioden speichern und danach entsprechend neu setzen. Dieses Vorgehen ist nicht empfehlenswert, da durch Aktivieren eines Debug-Levels (z. B. in der Datei `SDLM.nc`) bei bestimmten Ereignissen die Leuchtdioden ein- bzw. ausgeschaltet werden. Setzt man nun kurz nach Auftreten eines Ereignisses einen anderen Wert, ist es meist nicht möglich, dieses zu erkennen.

void set_lamp(bool val, int bit)
Interne Funktion, die von *Set_Lamp_Red,Green,Yellow* benutzt wird.

void Set_Lamp_Red(bool val)
Schaltet die rote Lampe ein (**true**) oder aus (**false**).

void Set_Lamp_Green(bool val)
Schaltet die grüne Lampe ein (**true**) oder aus (**false**).

void Set_Lamp_Yellow(bool val)
Schaltet die gelbe Lampe ein (**true**) oder aus (**false**).

bool UARTprintf(char data)*
Funktion, die zu Debug-Zwecken verwendet werden kann. Gibt eine Zeichenkette auf UART-Port 1 aus.

bool UARTprintflen(char data, int len)*
Funktion, die zu Debug-Zwecken verwendet werden kann. Gibt eine Zeichenkette auf UART-Port 1 unter Berücksichtigung der Länge aus. Diese Funktion kann damit auch dazu verwendet werden, andere Datentypen als Zeichenketten auszugeben oder sich auf den relevanten Teil dabei zu beschränken.

bool X_UART_Send(int port, SDL_Bit_String data)*
Diese Funktion ist genau wie *sendUARTPtr* aufgebaut, wobei hier ein SDL-**Octet_string** übergeben werden kann.

void UART_Setup(int port, unsigned int baud)
Diese Funktion ist genau wie *serialSetBaudPtr* aufgebaut.

bool wirelessCCAStatus()
Diese Funktion gibt den aktuellen Kanalstatus zurück.

bool sendWireless(SDL_Bit_String data, bool CCA)*
Interne Funktion, die für SDL-**Octet_strings** die Funktion *sendWirelessPtr* kapselt. Die maximale Payload beträgt hierbei 121 Byte.

bool ZigBee_Send(SDL_Bit_String data, bool CCA)*
Verwendet direkt *sendWireless*.

bool TOSsendWireless(char data, bool CCA)*
Für Character-Strings, vergleichbar mit *sendWirelessPtr*.

bool TOSsendWirelessLen(char data, int len, bool CCA)*
Für Character-Strings, vergleichbar mit *sendWirelessPtr* unter Berücksichtung der Länge.

void ZigBee_Setup(int channel, int txpower)
Diese Funktion ist genau wie *wirelessSetupPtr* aufgebaut.

uint8_t getID_OR_CRC(bool ID)*
Diese Funktion ist genau wie *getID_OR_CRCPtr* aufgebaut.

void setZigbeeMode(int mode)
Diese Funktion ist genau wie *setZigbeeModePtr* aufgebaut.

G.3 Interface in SDL

Innerhalb von SDL werden normalerweise keine C-Funktionen außer zu Debug-Zwecken aufgerufen. Dies ist vor allem für eine plattformunabhängige Entwicklung hinderlich. Deshalb wurden in SDL vier Packages erstellt, die einzelne Funktionen, je nach Bedarf des MICAz, zur Verfügung stellen.

- **Ticks**
  Das Package Ticks ist für fast alle Projekte nötig, da hier die Prozedur Ticks definiert ist, die zur Steuerung von Timern benötigt wird.
  Soll ein Timer auf 1 Sekunde und 5 Millisekunden aufgezogen werden, schreibt man *set(now + (call Ticks (1, 5, 0)), Timer)*. Der erste Parameter gibt die Zeit in Sekunden, der zweite die Zeit in Millisekunden und der dritte in Mikrosekunden an. Aufgrund dieser Feingranularität sollten keine Timer aufgezogen werden, die länger als 19 Minuten laufen. Eine Sekunde entspricht 921.600 Ticks, in Cmicro läuft der Zeitzähler nach 2^{31} Ticks über, daraus ergeben sich

$$\frac{2^{30}}{921.600} \approx 1.165 \text{ Sekunden.} \approx 19,4 \text{ Minuten.}$$

- **Lamp**
  Dieses Package ist für eine simple Ein- und Ausgabe über die drei Leuchtdioden zuständig. In ihm sind die Signale SET_LAMP_RED, SET_LAMP_GREEN und SET_LAMP_YELLOW definiert, die alle in der Signalliste micaz_lamp_out zusammengefaßt sind. Jedes dieser Signale hat einen **Boolean** Parameter, der den Zustand der jeweiligen Lampe angibt.

- **UART**
  Das Package UART ist für alle Funktionen rund um die zwei seriellen Schnittstellen da. Hierin sind die Ausgabesignale X_UART_SEND, X_UART_SETUP und UART_SEND sowie die Signalliste micaz_UART_out, in der alle Ausgabesignale definiert sind. Die Parameter entsprechen den jeweiligen Parametern aus der SDL-Environment-Schnittstelle.
  Das Eingabesignal X_UART_RECV wird mit den Parametern Port und Daten jeweils als **Integer** an das System übergeben. Das Signal UART_FAILED wird immer dann vom Environment zurückgegeben, wenn die UART-Schnittstelle gerade mit einer anderen Sendeoperation noch belegt ist. Diese beiden Signale sind in der Signalliste micaz_UART_in zusammengefaßt.

- **ZigBee**
  Für die Übertragung über die Funkschnittstelle ist das Package ZigBee zuständig. Hier sind die Ausgabesignale ZIGBEE_SEND zum Senden eines **Octet_string**s, ZIGBEE_SEND_CCA zum Senden eines Octet_strings unter Beachtung eines freien Kanals, ZIGBEE_SETUP zum Einstellen des Kanals und der Sendestärke, wie in *wirelessSetupPtr* gezeigt, sowie das Signal ZIGBEE_MODE, wie in *setZigbeeModePtr*, definiert. Alle Eingabesignale finden sich in der Signal-Liste micaz_Zigbee_out wieder. Bei der Sendenoperation ist zu beachten, daß die maximale Payload 121 Byte und die Zeit, bis der Transceiver sich im Sendemodus befindet, 192μs beträgt. Es existiert also auch bei der Operation ZIGBEE_SEND_CCA diese Totzeit.
  Die Eingabesignalliste enthält das Signal ZIGBEE_SFD, das mit einem **Boolean** Parameter angibt, ob eine Sendeoperation fehlgeschlagen (**false**) oder erfolgreich beendet wurde (**true**). Als weiteres Signal wird ZIGBEE_SENDING geschickt, sobald die Sendeoperation begonnen hat. Das Signal ZIGBEE_CCA mit einem **Boolean** Parameter gibt den aktuellen Kanalstatus an und wird bei einer Kanaländerung, auf jeden Fall aber nach einer Sendeoperation, von der Umgebung geliefert. Das letzte Signal in der Liste ist das Empfangssignal ZIGBEE_RECV mit den Parametern **Octet_string**, **Boolean** und **Integer**. Hier steht der **Octet_string** für die Daten, **Boolean** für die Korrektheit der Prüfsumme (CRC) des Pakets und der letzte Parameter gibt die Signalstärke an.

Anhang H

SDL-MAC-Beispielsystem

Die Folgenden Abbildungen zeigen die Beispielimplementierung einer kleinen MAC Schicht in SDL.

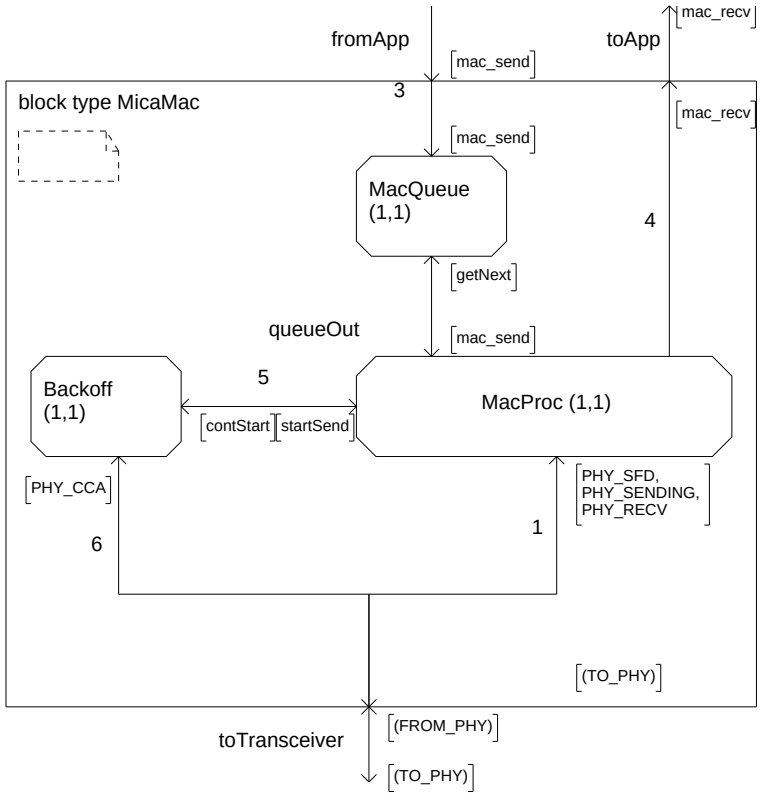

Abbildung H.1: Block der MAC-Schicht

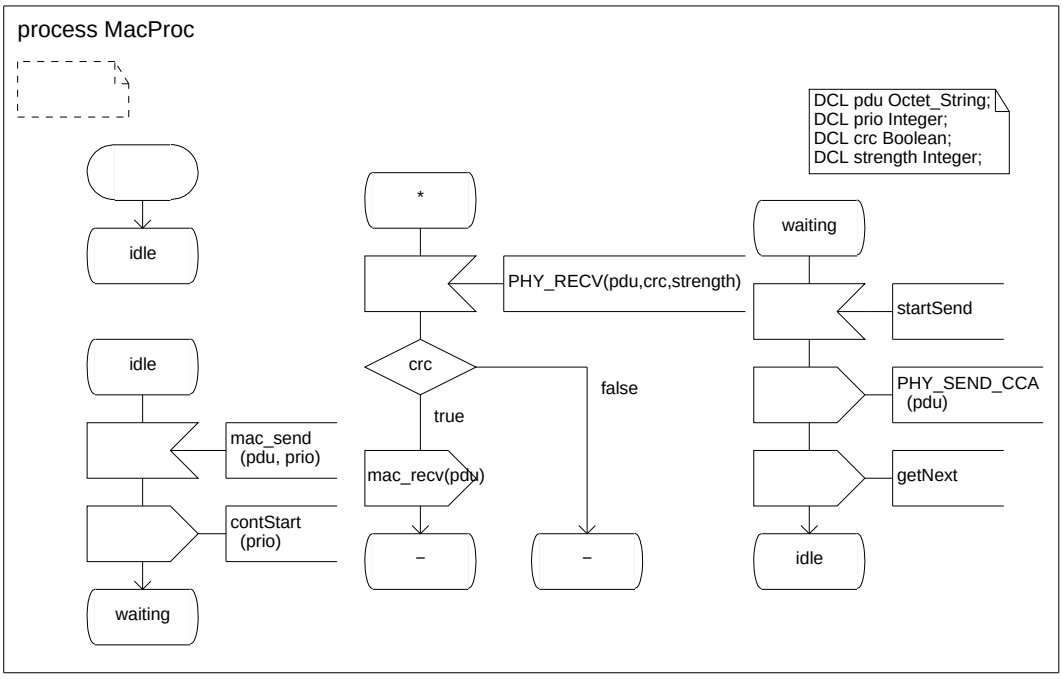

Abbildung H.2: Hauptprozeß der MAC-Schicht

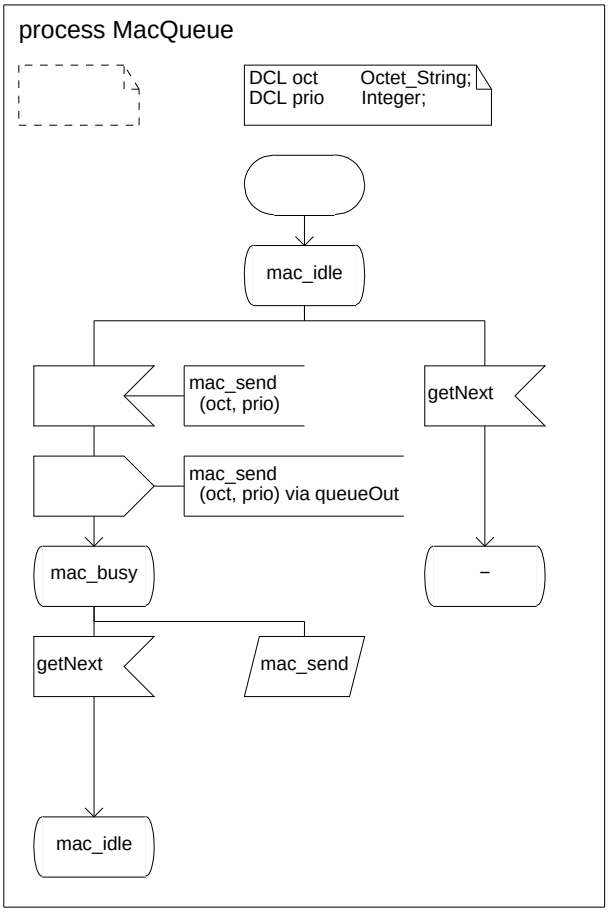

Abbildung H.3: Queue der MAC-Schicht

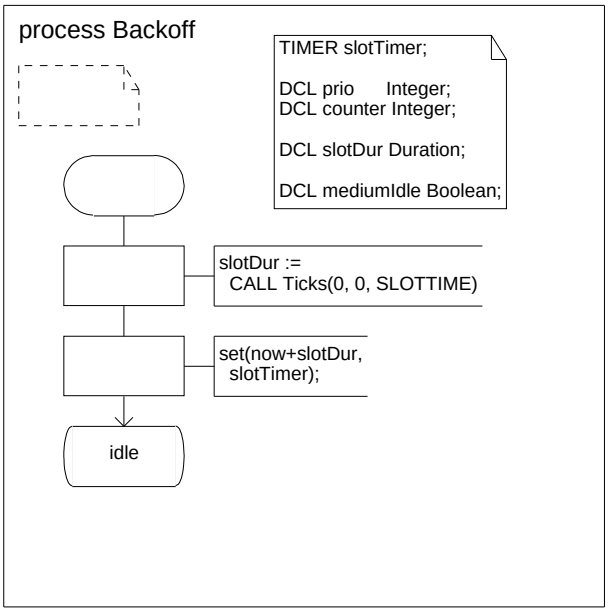

Abbildung H.4: Backoff-Prozeß der MAC-Schicht (1)

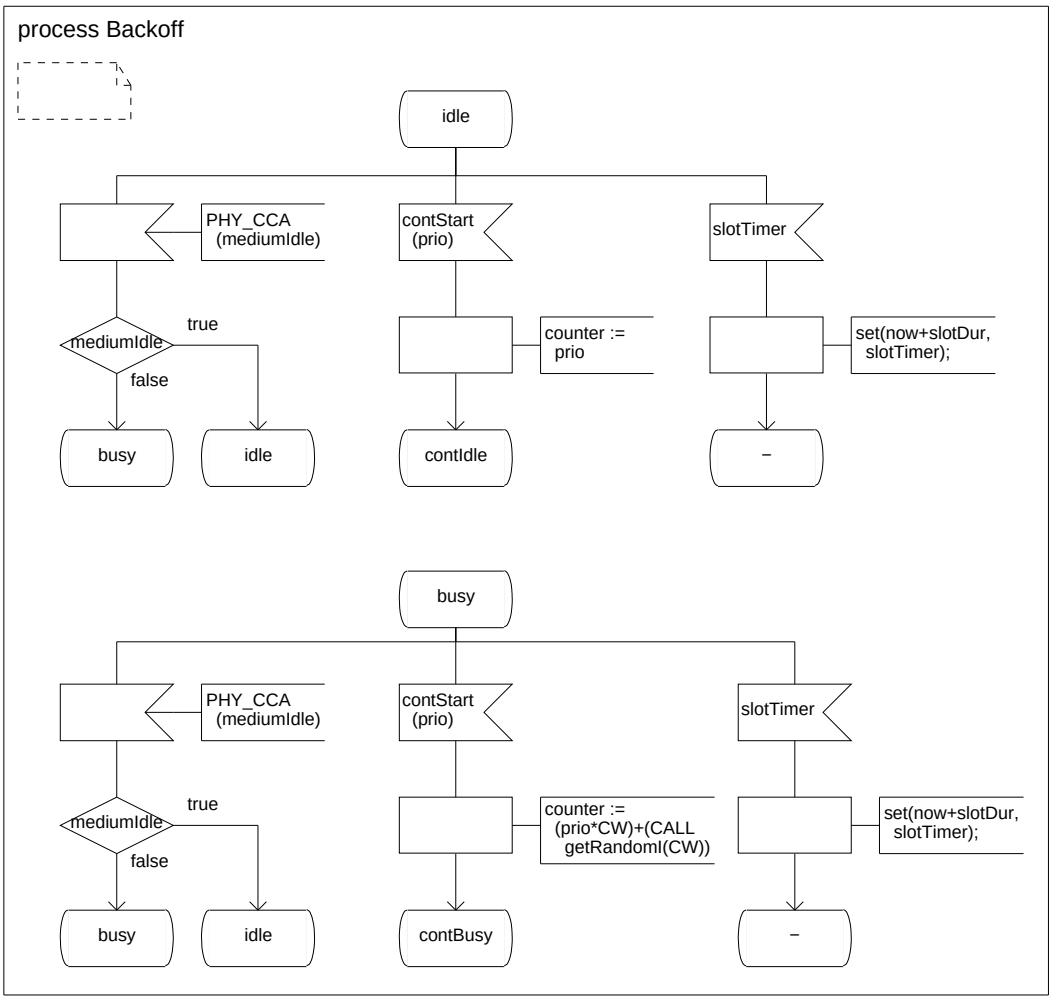

Abbildung H.5: Backoff-Prozeß der MAC-Schicht (2)

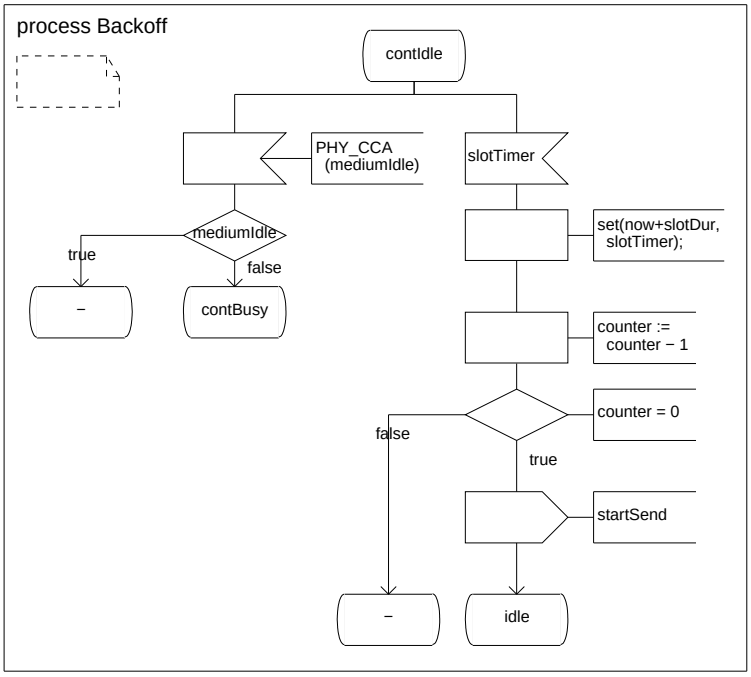

Abbildung H.6: Backoff-Prozeß der MAC-Schicht (3)

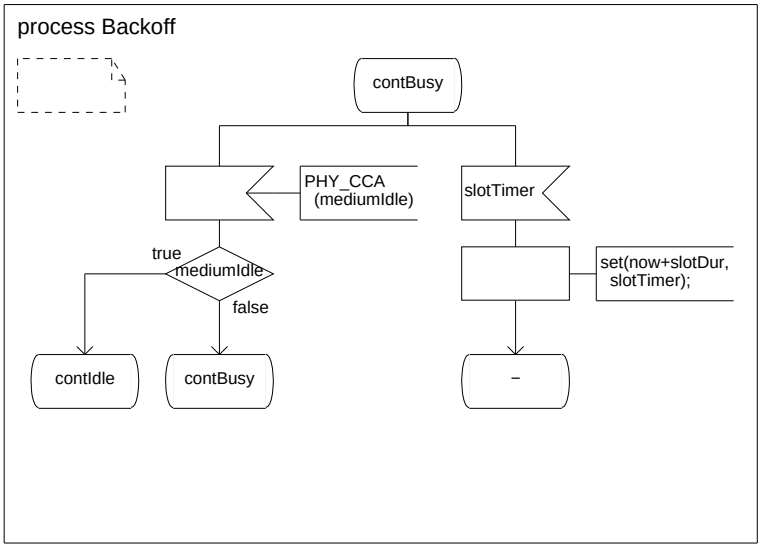

Abbildung H.7: Backoff-Prozeß der MAC-Schicht (4)

Anhang I

Abkürzungsverzeichnis

AMI	Ambient Intelligence
Bit	binary digit
Byte	Binary term (Backronym)
CCA	Clear Channel Assessment
CD	Compact Disc
CRC	Cyclic Redundancy Check
CSMA/CA	Carrier Sense Multiple Access / Collision Avoidance
Δ	Difference
DSN	Data Sequence Number
E/A	Eingabe / Ausgabe
EEPROM	Electrically Erasable Programmable Read-Only Memory
ETH	Eidgenössische Technische Hochschule
FCF	Frame Control Field
FIFO	First In First Out
FPARS	Formal Context Parameters
GCC	Gnu-C-Compiler
GHz	Gigahertz
h	Stunde
Hz	Hertz
IC	Integrated Circuit
I/O	Input / Output
kBaud	kilo (1000) Baud
kbps	kilo (1000) Baud
KByte	kilo (1024) Byte
LAN	Local Area Network
LED	Light-Emitting Diode
MAN	Metropolitan Area Network
MAC	Medium Access Control
MHz	Megahertz
μA	Mikroampère
μs	Mikrosekunde
mA	Milliampère
mAh	Milliampère Stunden

ms	Millisekunde
ns	Nanosekunde
PC	Personal Computer
PD	Power Down
PLL	Phases Lock Loop
QOS	Quality Of Service
RAM	Random Access Memory
ROM	Readonly Memory
RTC	Real Time Clock
RX	Receive
s	Sekunde
SDL	Specification and Description Language
sek	Sekunde
SEnF	SDL Environment Framework
SFD	Start of Frame Delimiter
SPI	Serial Peripheral Interface
SRAM	Serial Random Access Memory
t	time
TOS	TinyOS
Transceiver	Kombination der Wörter Transmitter und Receiver
TX	Transmit
UART	Universal Asynchronous Receiver Transmitter
V	Volt
WAN	Wide Area Network
WLAN	Wireless LAN

Literaturverzeichnis

[AMI] *Ambient Intelligence Homepage der TU-Kaiserslautern.* http://www.eit.uni-kl.de/AmI/.

[ATMa] *Data Sheet ATMEL ATmega128.* http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf. Revision 2467M-AVR-11/04.

[ATMb] *Data Sheet ATMEL AT45DB041B.* http://www.atmel.com/dyn/resources/prod_documents/doc3443.pdf. Revision 3443C-DFLSH-5/05.

[AVRa] *AVR ASM Tutorial.* http://www.avr-asm-tutorial.net/avr_de/index.html.

[AVRb] *AVR-GCC Tutorial.* http://www.mikrocontroller.net/articles/AVR-GCC-Tutorial.

[AVRc] *Installation des AVR-GCC.* http://gcc.gnu.org/install/specific.html#avr.

[CC204] *Spezifikation CC2420.* http://www.chipcon.com/files/CC2420_Data_Sheet_1_2.pdf, Juni 2004. Revision 1.2.

[cyg] *cygwin Homepage.* http://cygwin.com/.

[DS2] *Data Sheet DS2401.* http://pdfserv.maxim-ic.com/en/ds/DS2401.pdf. Revision 022102.

[Ene] *Energizer e2 lithium batteries.* http://www.energizer-eu.com/en/e2_lithium/default.htm.

[Fab02] Fabich, Carsten: *Einwegstrom: 27 Alkali-Mangan-Batterien LR6/Mignon im Vergleich.* c't, 23/02:192, 2002.

[IEE03] IEEE Computer Society Task Group 4: *802.15.4 Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specufications for Low-Rate Wireless Personal Area Networks (LP-WPANs).* http://standards.ieee.org/getieee802/download/802.15.4-2003.pdf, 2003.

[IT95] ITU-T, International Telecommunication Union: *CCITT Specification and Description Language (SDL)*, 1995. ITU-T Recommendation Z.100 (03/93) + (10/96) + (1998).

[KGGR05] Kuhn, Thomas, Alexander Geraldy, Reinhard Gotzhein und Florian Rothländer: *ns+SDL - The Network Simulator for SDL Systems*. In: Prinz, A., R. Reed und J. Reed (Herausgeber): *SDL 2005*, Lecture Notes in Computer Science (LNCS) 3530. Springer Hdb., 2005.

[LLB00] Luo, Haiyun, Songwu Lu und Vaduvur Bharghavan: *A New Model for Packet Scheduling in Multihop Wireless Networks*, 2000.

[MIC] *MICAz Datasheet*. `http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/MICAz_Datasheet.pdf`. Revision B.

[Moo56] Moore, Edward F.: *Automata Studies, Annals of Mathematical Studies*, Band no. 34, Kapitel Gedanken-experiments on Sequential Machines, Seiten 129 – 153. Princeton University Press, Princeton, N. J., 1956.

[nes] *nesC Homepage*. `http://nescc.sourceforge.net/`.

[SDLa] *SDL Tutorial*. `http://www.sintef.no/time/elb40/html/elb/sdl/sdl_t01.htm`.

[SDLb] *Tutorial on SDL*. `www.cs.auc.dk/~kgl/TOV05/SDL.pdf`.

[Sik04] Sikora, Prof. Dr. Axel: *ZigBee: Grundlagen und Applikation*. `http://www.elektroniknet.de/topics/kommunikation/fachthemen/2004/0002/`, 2004.

[TAU] *Tau SDL Hilfe*. Hilfe in der Tau SDL Suite [Tel] mitgeliefert.

[Tel] *Telelogic Tau SDL Suit Homepage*. `http://www.telelogic.com/products/tau/sdl/index.cfm`.

[Tin] *TinyOS Homepage*. `http://www.tinyos.net`.

[XBO] *CrossBow Homepage*. `http://www.xbow.com`.

[Zig] *ZigBee Alliance Homepage*. `http://www.zigbee.org`.

[Zin] Zinniker, Dr. Rolf: *Die ideale Batterie*. `http://www.ife.ee.ethz.ch/~zinniker/batak/ideal/`.