Hochschule für Technik und Wirtschaft Dresden
Fakultät Informatik/Mathematik

Forschungsbericht

im Masterstudiengang Angewandte Informationstechnologien (IK)

Sensornetze Gruppe 1

eingereicht von: Alexander Elchlepp (Mat.-Nr.: 30044)
eingereicht am: 17. Februar 2015
Betreuer: Prof. Dr. Jörg Vogt

Inhaltsverzeichnis

Abkürzungsverzeichnis III

Glossar IV

Abbildungsverzeichnis V

Listings VI

1 Vorwort 1

2 Umstellung auf ContikiMAC 3
- 2.1 Was ist ContikiMAC . . . 3
- 2.2 Prinzip von ContikiMAC . . . 4
- 2.3 Timings . . . 5
- 2.4 Packet Detection und Fast Sleep . . . 6
- 2.5 Phasenoptimierung . . . 7
- 2.6 Aktivierung und Konfiguration von ContikiMAC . . . 8
- 2.7 Probleme . . . 8

3 Integration weiterer CoAP Ressourcen 10
- 3.1 Batteriesensor . . . 10
- 3.2 Atmega128RFA1 Temperatursensor . . . 11

4 Reichweitentests 13
- 4.1 CCA mit RSSI . . . 13
- 4.2 CCA mit carrier sense . . . 14
- 4.3 Auswertung . . . 14

5 Strommessungen des Funkmoduls 16
- 5.1 Strommessung . . . 16
  - 5.1.1 Durchführung . . . 16
  - 5.1.2 Ergebnisse . . . 17
- 5.2 Untersuchungen der CCA-Phasen . . . 18

6 Änderungen am Contiki-Quellcode 20
- 6.1 CoAP-Ressourcen . . . 20
- 6.2 ADC . . . 20
- 6.3 CCA mit carrier sense . . . 20
- 6.4 Bugfixing . . . 22

7 Zusammenfassung und Ausblick 24

8 Literaturverzeichnis **25**

A Anhang **26**
A.1 Zyklische Aufwachphasen alle 8 Hz (125ms) 26
A.2 Wachphase des Radio-Moduls = 3,375ms (t_r) 27
A.3 Dauer einer CCA-Abfrage = 0,125ms (t_r) 28
A.4 Dauer der gesamten CCA-Routine = 0,250 ms 29
A.5 Dauer zwischen zwei CCA-Abfragen = 1 ms (t_c) 30

Abkürzungsverzeichnis

ADC Analog Digital Converter.

CCA Clear Channel Assessment.

RDC Radio Duty Cycle.

RSSI Received Signal Strength Indicator.

SoC System-on-a-Chip.

Glossar

6LoWPAN

Steht für IPv6 over Low power Wireless Personal Area Network. Dabei handelt es sich um IPv6 mit (u.a) einer Komprimierung des Headers, wodurch die Dauer einer Übertragung besonders bei batteriebetriebenen Geräten verkürzt werden kann.

IEEE802.15.4

Hierbei handelt es sich um ein Übertragungsprotokoll auf der Frequenz 2,4 GHz. Der Standard beschreibt dabei Schicht eins und zwei des OSI-Modells und ist besonders für energiesparende Einheiten vorgesehen.

Abbildungsverzeichnis

1 Eingesetzte Hardware 1
2 Schichten in Contiki 3
3 Prinzip ContikiMAC (Unicast) 4
4 ContikiMAC Timings 5
5 Prinzip der Phasenoptimierung 7
6 Messung der Batteriespannung 10
7 Reichweitentest 14
8 Messung der Stromstärke 16

Listings

1 ContikiMAC aktivieren 8
2 Frequenz der Aufwachphasen 8
3 Einstellung der Phasenoptimierung 8
4 CCA Threshold 9
5 Berechnung der Batteriespannung 11
6 Bits zur Termperaturermittlung setzen 11
7 Ermittelung der internen Temperatur 12
8 Messung einer CCA-Abfrage 18
9 Messung der gesamten CCA-Routine 18
10 t_c mit 0,5 ms definiert 19
11 Schalter für RSSI bzw. carrier sense 21
12 Implementierung carrier sense 21
13 Bugfixes am Contiki-Code 22

1 Vorwort

Im Forschungsseminar Sensornetze geht allgemein um das Thema Hausautomatisierung.
Gruppe 1 des Forschungsseminars betreut die eingesetzte Hardware und nimmt Optimierungen bzw. Veränderungen am eingesetzten Betriebssystem Contiki (für μC) vor.
Als Hardware kamen zum einen deRFbreakout Board *(siehe Abbildung 1a)* sowie deRFnodes *(siehe Abbildung 1b)* zum Einsatz. Ersteres dient als Border-Router und wird mit Hilfe einer USB-UART-Bridge mit dem PC/Server verbunden und auch mit Strom versorgt. Die deRFNodes dienen als Sensorknoten. Sie eignen sich besonders gut dafür, da sie bereits über ein Batteriefach (3 x AA), diverse vorinstallierte Sensoren (u.a. Licht- und Beschleunigungssensor) sowie Aktoren (Buttons) besitzen.
Auf beiden Boards befindet sich jeweils ein System-on-a-Chip (SoC) vom Typ Atmega128RFA1 *(siehe Abbildung 1c)*. Dabei handelt es sich um einen μC und Funkmodul zusammen. Das Funkmodul arbeitet auf der Funkfrequenz von 2,4 GHz (IEEE802.15.4).

(a) deRFbreakout Board als Border-Router

(b) deRFnode als Sensorknoten

(c) deRFmega128

Abbildung 1: Eingesetzte Hardware

In den Nachfolgenden Kapiteln werden jeweils einzelne Aufgaben, die während des Forschungsseminars abgearbeitet wurden, näher erläutert.

2 Umstellung auf ContikiMAC

2.1 Was ist ContikiMAC

ContikiMAC ist ein proprietärer Mechanismus, um beispielsweise in Sensornetzwerken eine möglichst geringe Stromaufnahme von Sensorknoten zu gewährleisten. Sind diese Knoten kabellos über Funk miteinander verbunden, so kostet es in der Regel viel Energie Daten zu senden oder zu empfangen. Aus diesem Grund muss die Sende-/Empfangseinheit möglichst oft ausgeschaltet sein und nur aktiviert werden, wenn es notwendig ist.

Dieser Mechanismus wird Radio Duty Cycle (RDC) genannt und ContikiMAC ist ein Mechanismus, um dies zu realisieren.

In Contiki wurde für RDC eine neue Schicht eingefügt, die sich zwischen der regulären Schicht 1 und 2 des OSI-Modelles ansiedelt. Da es in Contiki verschiedene RDC-Verfahren gibt, ist dies durchaus sinnvoll.

In Abbildung 2: Schichten in Contiki, ist die Einordnung des RDC-Layers in Contiki zu sehen:

ContikiMAC nutzt hierbei keine zusätzlichen Pakete zum signalisieren und fügt auch keine Informationen dem Header hinzu. Das ganze wird durch Einhaltung genau festgelegter Zeitvorgaben erreicht.

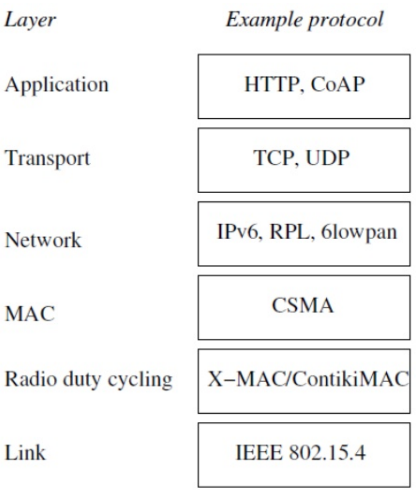


Abbildung 2: Schichten in Contiki

2.2 Prinzip von ContikiMAC

ContikiMAC versucht die Sende-/Empfangseinheit möglichst lange ausgeschaltet zu lassen. Periodisch wird geschaut, ob die Empfangseinheit eine Aktivität verzeichnet. Wenn dem so ist, bleibt der Empfänger so lange aktiv, bis ein Paket vollständig empfangen werden kann. Der Empfänger sendet zum Abschluss der Übertragung ein Acknowledgement an den Sender.

Der Sender wiederum sendet ein Packet so oft, bis er ein Acknowledgement des Empfängers erhält oder die maximale Anzahl Wiederholungen erreicht wurde.

Das Prinzip wird in dem folgendem Bild veranschaulicht:

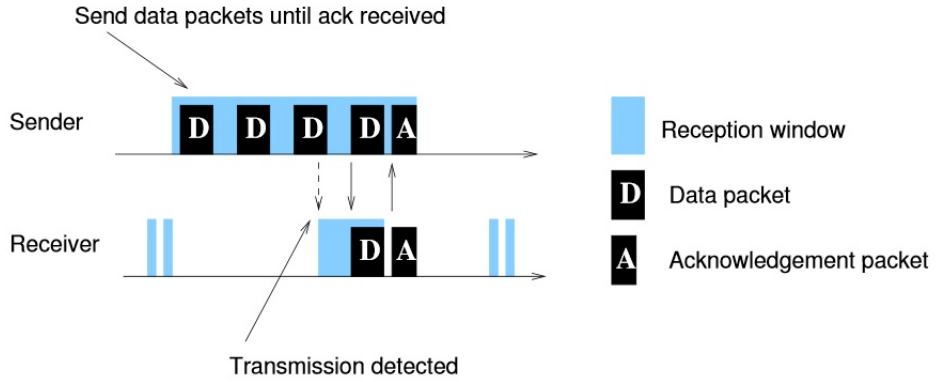


Abbildung 3: Prinzip ContikiMAC (Unicast)

Bei einem Broadcast sendet der Sender ein Packet ebenfalls periodisch. Der Unterschied zum Unicast ist, dass ein Empfänger kein Acknowledgement sendet. Somit ist gewährleistet, dass alle Empfänger das Packet empfangen können.

Um herauszufinden, ob eine Übertragung stattfindet, nutzt ContikiMAC einen kostengünstigen Mechanismus, um zu prüfen, ob der Kanal frei ist oder nicht (Clear Channel Assessment (CCA)). Dabei wird ein Indikator für die Empfangsstärke (Received Signal Strength Indicator (RSSI)) von der Empfangseinheit ausgelesen und mit einem vorher definierten Schwellwert verglichen.

Wenn der RSSI-Wert kleiner dem Schwellwert ist, ist der Kanal frei (es wird nichts empfangen). Liegt er darüber, ist der Kanal belegt (es wird etwas empfangen).

2.3 Timings

In der folgenden Abbildung wird gezeigt, wie die Timings ineinander greifen:

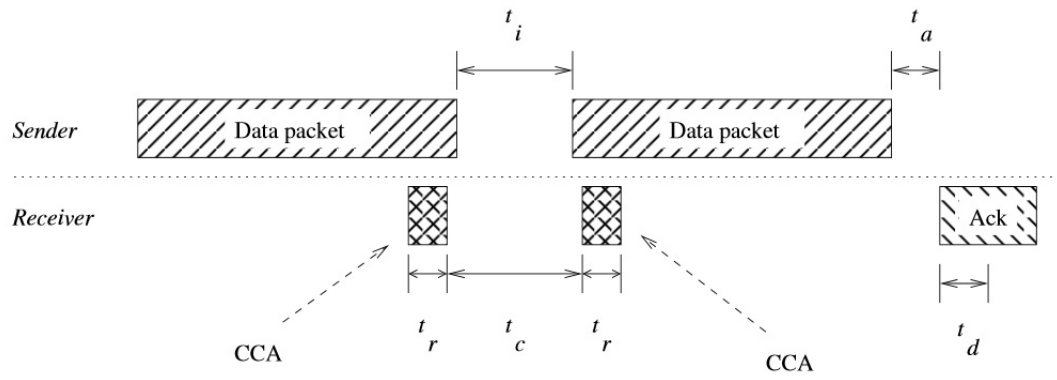


Abbildung 4: ContikiMAC Timings

- t_i : Zeit zwischen zwei Paketübertragungen
- t_r: die Zeit für eine verlässliche Aussage, ob ein Paket empfangen werden soll (CCA)
- t_c: die Zeit zwischen zwei CCA-Phasen
- t_a: die Zeit nachdem ein Acknowledgement nach Erhalt eines Paket gesendet wird
- t_d: die Zeit, um mit Sicherheit sagen zu können, dass es sich um ein Acknowledgement handelt (seitens des Senders)

Um ein Paket erfolgreich zu detektieren und anschließend zu empfangen, gibt es einige Bedingungen, die bei der Festlegung der Timings eingehalten werden müssen.

Die Zeit zwischen zwei Paketen (t_i) muss kleiner sein, als die Zeit zwischen den Aufwachphasen (t_c) des Empfängers, andernfalls kann nicht sichergestellt werden, dass mindestens in einer der CCA-Phasen ein Paket detektiert werden kann.

Mit dieser Einschränkung geht auch die minimale Paketgröße einher. Damit die Übertragung eines Paketes nicht zwischen zwei CCA-Phasen fällt, muss die Übertragungsdauer eines Paketes also mehr als $2\ t_r + t_c$ betragen.

Weiterhin muss der Sender bis zum versenden des nächsten Paketes warten, bis ein mögliches Acknowledgement eines Empfängers bei ihm ankommt und detektiert werden kann. Damit muss $t_a + t_d$ kleiner sein als t_i. Das ganze veranschaulicht dargestellt ergibt somit folgende Bedingungen für die Timings:

$$t_a + t_d < t_i < t_c < t_c + 2 * t_r < t_s$$

Wobei t_s die Übertragungsdauer eines Paketes darstellt.

Da auf Schicht 1 das Übertragungsprotokoll IEEE802.15.4 genutzt wird, schreibt dieser Standard bereits einige der Timings vor. Somit ist t_a mit einer Zeitdauer von 0,192 Millisekunden fest vorgegeben. Weiterhin wird der Wert für t_d mit 0,16 Millisekunden ausgewiesen.

Die Zeit für eine CCA-Phase ist wiederum abhängig von der eingesetzten Hardware. In Contiki wird der Wert für den CC2420 Radio Transceiver zugrunde gelegt, welcher 0,192 Millisekunden beträgt. Es ergibt sich somit:

$$0,352 < t_i < t_c < t_c + 0,384 < t_s$$

Die übrigen Werte müssen selbst festgelegt werden und sehen in ContikiMAC wie folgt aus:

$$0,352 < 0,4 < 0,5 < 0,5 + 0,384 < 0,884 \text{ (Millisekunden)}$$

Wie bereits erwähnt, existiert eine minimale Paketgröße, damit alle zuvor genannten Bedingungen eingehalten werden können. Es kann jedoch der Fall eintreten, dass bei Verwendung von 6LoWPAN (aufgrund der Komprimierung des Headers), ein Paket kleiner als der vorgegebene Minimalwert ist. In ContikiMAC wird in so einem Fall einfach die Komprimierung deaktiviert, somit ist gewährleistet, dass Pakete immer lang genug sind.

2.4 Packet Detection und Fast Sleep

Während der CCA-Phasen werden Pakete nicht direkt erkannt, vielmehr wird nur erkannt, ob das Empfangssignal (RSSI) einen bestimmten Schwellwert überschreitet. Dies kann nun mehrere Ursachen haben:

1. ein anderer Knoten sendet etwas an den aktuellen Sensor
2. ein anderer Knoten sendet etwas, dass nicht für den aktuellen Sensor bestimmt ist
3. andere Geräte senden etwas

Wenn ein Paket erkannt wurde, dieses jedoch nicht für den aktuellen Knoten bestimmt ist, ist es sinnvoll diesen Knoten möglichst schnell wieder schlafen zu legen. Aus diesem Grund wurde in ContikiMAC ein weiterer Mechanismus zum Energiesparen umgesetzt. Der fast sleep Mechanismus.

Wenn ein Paket erkannt wird, bleibt die Empfangseinheit strikt für eine Dauer von $2 * t_l + t_i$ an. Wobei t_l die Zeit für das längst mögliche Paket entspricht.

Der Empfänger legt sich vorzeitig wieder schlafen wenn:

1. die Übertragung eines Paketes länger als t_l dauert (Bsp: wenn ein anderes Gerät Daten überträgt)

2. das Ende eines Paketes erkannt wurde aber die Ruhezeit (t_i) bis zum nächsten Paket zu lange dauert (Bsp: das Paket war für einen anderen Knoten bestimmt und dieser hat bereits ein ACK gesendet)

3. eine korrekte Dauer der Ruhezeit (t_i) erkannt, danach jedoch kein gültiger Paketanfang detektiert werden konnte

2.5 Phasenoptimierung

Die Phasenoptimierung ist wieder ein Mechanismus, um energieeffizienter senden zu können. Dieser Mechanismus ist abgekapselt von ContikiMAC, um ihn auch anderen RDC-Protokollen zugänglich zu machen.

Hierbei führt der Sender eine Liste mit Empfängern, in der steht, zu welchen Zeitpunkten sie ihm bei einer der letzten Übertragungen das Acknowledgement gesendet haben. Daraus kann der Sender berechnen, zu welchem Zeitpunkt er ein Paket senden muss, damit es genau zu dem Zeitpunkt ankommt, dass der Empfänger dieses erhalten kann.

Kurz bevor der Empfänger aufwacht, sendet der Sender das Paket, somit muss er im besten Fall nur noch zwei Mal ein Paket senden, bis der Empfänger es erhalten kann. Die folgende Abbildung zeigt dies exemplarisch:

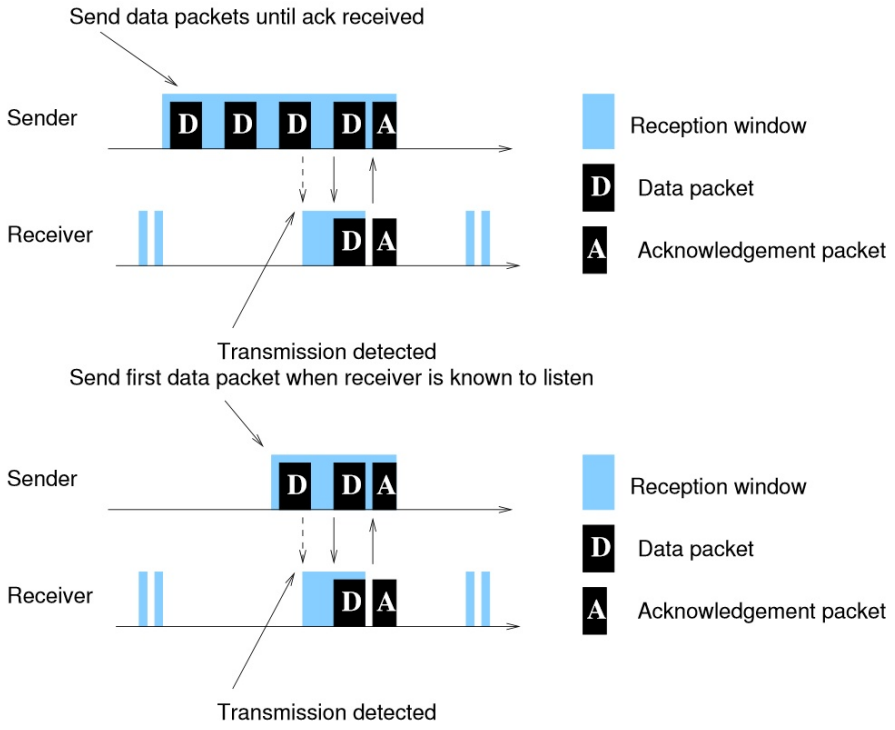


Abbildung 5: Prinzip der Phasenoptimierung

2.6 Aktivierung und Konfiguration von ContikiMAC

Alle notwendigen Einstellungen, bezüglich der plattformabhängigen Aktivierung und Konfiguration von ContikiMAC, werden in der Datei

platform/avr-atmega128rfa1/contiki-conf.h

vorgenommen.
In dem aktuell verwendeten Code ist ContikiMAC standardmäßig aktiviert worden. Unter anderem ist es notwendig, den RDC-Mechanismus korrekt auf ContikiMAC einzustellen:

Listing 1: ContikiMAC aktivieren

```
244 #define NETSTACK_CONF_RDC contikimac_driver
```

Mit der nachfolgenden Zeile stellt man die Frequenz, der Aufwachphasen für ContikiMAC ein. Standardmäßig ist dieser Wert mit 8 Hz gesetzt. Dies bedeutet, dass alle 125 ms geschaut wird, ob ein Paket empfangen werden soll.

Listing 2: Frequenz der Aufwachphasen

```
246 #define NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE 8
```

Um die Kanallast weiter zu reduzieren, kann man die Phasenoptimierung aktivieren *(siehe Abschnitt 2.5)*. Aufgrund von Problemen mit auseinanderdriftenden Timern, wurde diese Option deaktiviert *(siehe Abschnitt 2.7)*.

Listing 3: Einstellung der Phasenoptimierung

```
252 #define CONTIKIMAC_CONF_WITH_PHASE_OPTIMIZATION 0
```

2.7 Probleme

Nach der Aktivierung von ContikiMAC (und dem Upgrade von Contiki 2.6 auf 2.7) traten zwei schwerwiegende Probleme auf:

1. sehr hoher Paketverlust auf der Funkstrecke (> 80 %)
2. sehr geringe Reichweite zwischen Border-Router und Sensorknoten (wenige Meter)

Das erste Problem hatte seinen Ursprung in dem Versionsupgrade von Contiki 2.6 zu 2.7. Die aktuell genutzte Plattform atmega128rfa1 im Ordner */platform* wird nicht mehr gepflegt. Im Zuge der neuen Contiki-Version wurde ein entscheidendes #define umgenannt und eben nicht für die im Forschungsseminar genutzten Plattform nachgetragen.

Im speziellen ging es um das #define von Listing *3 Einstellung der Phasenoptimierung.*
Hinzu kommt, dass die Quarze bzw. RC-Oszillatoren der genutzten SoCs nicht sehr zuverlässig laufen. Die Timer laufen auseinander und der Border-Router sendet seine Pakete zu den Zeiten, wo er vermeintlich denkt, ein Sensorknoten wäre wach, um das Paket zu empfangen. Da sich die Knoten jedoch nicht synchronisieren, kommt es zu einem Zeitversatz und Pakete können nicht zugestellt werden.
Mit der Deaktivierung der Phasenoptimierung konnte das Problem des hohen Paketverlustes beseitigt werden.

Die Untersuchungen für das zweite Problemen erwiesen sich als schwieriger. Hierbei wurde auch Kontakt mit der Contiki-Community aufgenommen, von der einige hilfreiche Hinweise kamen, die das Problem letztlich zu Tage brachten.
In der Standardeinstellung einer CCA-Phase wird von dem Funkmodul ein Indikator für die gemessene Signalfeldstärke (RSSI) abgerufen und mit einem vorher festgelegten Schwellwert (threshold) verglichen. Liegt der ermittelte RSSI-Wert nun höher als der Schwellwert, so wurde kein Signal detektiert. Befindet sich der gemessene Wert jedoch unterhalb des Schwellwertes, so wird davon ausgegangen, dass ein Paket empfangen werden soll.
Standardmäßig war der Schwellwert mit -77 dbm vorgegeben. Dieser Wert war für die eingesetzte Hardware noch nicht niedrig genug. Es wurde der niedrigst mögliche Wert von -90 dbm eingestellt und somit konnten auch höhere Reichweiten erzielt werden *(siehe auch Abschnitt 4 Reichweitentests).*

Listing 4: CCA Threshold

```
285 #define RF230_CONF_CCA_THRES -90
```

Trotz dieser Maßnahme ist die Indoor-Reichweite noch nicht zufrieden stellend. Über mehr als zwei Räume (Ziegelwände) ist kein stabiler Verbindungsaufbau möglich.

3 Integration weiterer CoAP Ressourcen

Zum gegenwärtigen Zeitpunkt können 7 Sensoren als CoAP-Ressource abgefragt werden:

- Temperatur des SHT21 *(sht21_temperature)*
- Luftfeuchtigkeit des SHT21 *(sht21_humidity)*
- Temperatur des Atmega128RFA1 *(intern_temp)*
- Luftdruck des BMP085 *(bmp085)*
- Status Button 1 auf deRFNode *(button1)*
- Batteriespannung auf deRFNode *(battery)*
- letzter RSSI-Wert des Funkmoduls *(rssi)*

Die Sensoren sind unter der URL *sensors/<name>* erreichbar.

Im Folgenden wird die Integration des Batteriesensor sowie internen Temperatursensors des Atmega128RFA1 näher beschrieben.

3.1 Batteriesensor

Die Ermittlung der Batteriespannung ist in der Regel nur über separate Sensoren oder über andere Umwege möglich. Die einfachste Methode ist die Nutzung eines Spannungsteilers *(siehe Abbildung 6)*.

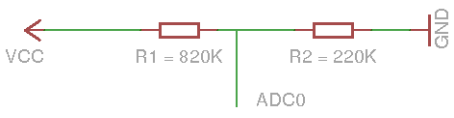


Abbildung 6: Messung der Batteriespannung

Hierbei ist der Wert für *VCC* von Interesse, den es nun gilt zu Berechnen. Der Sensorknoten deRFNode besitzt genau den Schaltkreis aus Abbildung 6. Somit sind 3 Werte Bekannt und über die umgestellte Spannungsteilerformel

$$VCC = \frac{V_{in} * (R1 + R2)}{R2}$$

kann die Batteriespannung ermittelt werden.
Die Werte für R1 und R2 sind durch die Beschaltung fest vorgegeben. Zur Ermittelung von V_{in} wird der 10-bit Analog Digital Converter (ADC) genutzt. Der ermittelte Wert muss unter Zuhilfenahme einer internen Referenzspannung ($V_{ref} = 1,6V$) in eine Spannung umgerechnet werden:

$$V_{in} = \frac{ADC0 * V_{ref}}{1023}$$

(Vgl. ATMEL 2014, 27.8 ADC Conversion Result)

Zur Bestimmung des Wertes für ADC0 wird wiederum die bereitgestellte API aus *Abschnitt 6.2 ADC* genutzt.
In der Datei *project/coap-sensornode/coap-sensornode.c* wird nun eine CoAP-Ressource *sensors/battery* bereitgestellt und liefert auf Anfrage (GET bzw. Observe) den berechneten Wert der Batteriespannung *(siehe Listing 5)*.

Listing 5: Berechnung der Batteriespannung

```
655 // Wert von ADC0
656 int adcVal = readADC(PF0);
657 // gemessene Spannung an ADC0 in V auf 2 nachkommastellen genau (
    festkomma arithmetik), ref Spannung = 1.6V
658 unsigned long int adc_in_v = ((uint32_t)adcVal * 160) / 1023;
659 // Spannungsteilerformel nach Vin umgestellt mit R1=820kOhm, R2=220
    kOhm
660 unsigned int voltage = (adc_in_v * 1040) / 220;
```

Ein Observe auf diese Ressource, liefert alle 60 Sekunden die aktuelle Spannung.

3.2 Atmega128RFA1 Temperatursensor

Der eingesetzte μC Atmega128RFA1 verfügt über einen integrierten Temperatursensor, auf dessen Wert man als CoAP-Ressource zugreifen kann (inkl. Observe). Eine Referenzimplementierung sowie Berechnungsvorschriften können zudem dem Datenblatt des Controllers entnommen werden *(Vergleich ATMEL 2014, 27.9 Internal Temperature Measurement)*.
Um die Temperatur zu ermitteln, wird eine bestimmte Beschaltung des ADC genutzt. Hierfür müssen die im Datenblatt angegebenen Bits im Register ADMUX korrekt gesetzt werden.
Siehe dazu die Datei *platform/avr-atmega128rfa1/dev/adc.c*:

Listing 6: Bits zur Termperaturermittlung setzen

```
71 ADCSRB |= _BV(MUX5);
72 ADMUX = _BV(REFS1) | _BV(REFS0) | 0b1001 ;
73 ADCSRA = _BV(ADEN) | _BV(ADPS0) | _BV(ADPS2) ;
```

Zur Ermittlung des Wertes wird die bereitgestellte API aus *Abschnitt 6.2 ADC* genutzt.
Der ermittelte Wert nach der AD-Wandlung muss nun noch entsprechend der gegebenen Formeln (siehe Datenblatt) umgerechnet werden.
Da das Rechnen mit float-Werten auf μControllern oftmals nicht ad-hoc funktioniert bzw. hierfür extra Bibliotheken eingebunden werden müssen und somit auch mehr Speicher benötigt wird, nutzt man stattdessen die Festkomma-Arithmetik.
Anstatt mit float-Werten zu arbeiten:

$$°C = ADC_{TEMP} * 1.13 - 272.8$$

rechnet man die gegebenen Konstanten * 100 und beseitigt somit das Komma

$$°C = ADC_{TEMP} * 113 - 27280$$

Die Genauigkeit wird dadurch in keinster Weise beeinträchtigt. Um nun bei dem Senden der Antwort an einen Client trotzdem eine Ausgabe mit Komma zu erhalten, nutzt man den Modulo-Operator:

$$°C \mod 100$$

Damit werden die zwei Nachkommastellen der Temperatur ausgegeben *(siehe Listing 7)*.

Listing 7: Ermittelung der internen Temperatur

```
573 // Temperatur auf 2 Nachkommastellen genau
574 int t = readInternalTemp();
575 snprintf(tempstr, 20, "%3d.%02d [C]", t/100, t%100);
```

4 Reichweitentests

Wie bereits in *Abschnitt 2.7 Probleme* erwähnt, gab es anfängliche Schwierigkeiten, die sich in einer viel zu geringen Reichweite äußerten. Die genannten Probleme konnten bereits beseitigt werden, jedoch war die Reichweite immer noch nicht zufriedenstellend.
Aus diesem Grund wurde geschaut, wie die Reichweite weiter verbessert werden kann.

Die verwendeten SoCs tragen den Namen deRFmega128 und haben laut Produktbeschreiben von Dresden Elektronik eine Reichweite von 200 m (Freifeld). Es galt nun zu evaluieren, ob auch im Foschungsseminar eine solche Reichweite erzielt werden kann. Indoortests zeigten jedoch bereits, dass eine stabile Verbindung meist nur über 2 - 3 Räume möglich war.

Da im Forschungsseminar ContikiMAC aktiviert ist, ist es wichtig sich besonders die CCA-Phasen anzuschauen, da hierbei entschieden wird, ob Daten empfangen werden sollen oder nicht.

Die eingesetzte Hardware führt ein CCA selbst durch. ContikiMAC fragt bei der Hardware nur nach, ob der Kanal frei ist oder nicht (CCA). Wie die Hardware diese Überprüfung vornimmt, dafür gibt es mehrere Möglichkeiten. Zwei davon werden in den nachfolgenden Abschnitten näher untersucht.

4.1 CCA mit RSSI

Standardmäßig wird während einer CCA-Phase geschaut, ob ein von der Hardware bereitgestellter Indikator für die gemessene Signalfeldstärke (RSSI) größer oder kleiner einem in Contiki definiertem Schwellwert (threshold) ist.
Das benutzte Funkmodul besitzt eine Empfindlichkeit von -100 dbm. Das Problem hierbei ist, dass die Bestimmung des RSSI-Wertes nur von -10 dbm bis -90 dbm funktioniert *(Vgl. ATMEL 2014, Figure 9-17. Mapping between RSSI Value and Received Input Power).*
Die Differenz von 10 dbm, die bei diesem Verfahren auftaucht, könnte bereits für eine geringe Reichweite bei aktiviertem ContikiMAC sprechen. Signale, die genau in diese Lücke fallen, können schlichtweg nicht mehr detektiert werden, obwohl sie durch das Funkmodul noch empfangen werden könnten.
Um die größtmögliche Reichweite bei diesem Verfahren zu erhalten, wurde der konfigurierbare Schwellwert auf den niedrigstmöglichen Wert von -90 dbm gestellt *(siehe Listung 4)*. Das heißt, alle Signale von -10 dbm bis -90 dbm werden detektiert.

4.2 CCA mit carrier sense

Im vorherigen Abschnitt wurde eine große Schwäche durch die Nutzung des RSSI-Wertes deutlich. Die Hardware bietet jedoch noch eine weitere Möglichkeit. Anstatt nur die reine Signalfeldstärke zu messen, wird geschaut, ob das was empfangen wird einem gültigen IEEE802.15.4 Signal zuzuordnen ist. Diese Überprüfung funktioniert laut Datenblatt auch bis zur Empfindlichkeitsgrenze des Funkmoduls von -100 dbm *(Vgl. ATMEL 2014, 9.5.5.3 Data Interpretation)*.

Dieser neue Modus ist in Contiki nicht implementiert und musste nachträglich hinzugefügt werden *(siehe Abschnitt 6.3 CCA mit carrier sense)*.

4.3 Auswertung

Alle Reichweitentests wurden unter freiem Himmel (Freiluft) bei einer Außentemperatur von 8 $°C$ durchgeführt. Die Module wurden dabei in einer Höhe von 1,20 m zueinander ausgerichtet.

Gemessen wurde mit 3 Einstellungen:

- ContikiMAC deaktiviert
- ContikiMAC (CCA mit RSSI)
- ContikiMAC (CCA mit carrier sense)

Das Ergebnis sah wie folgt aus:

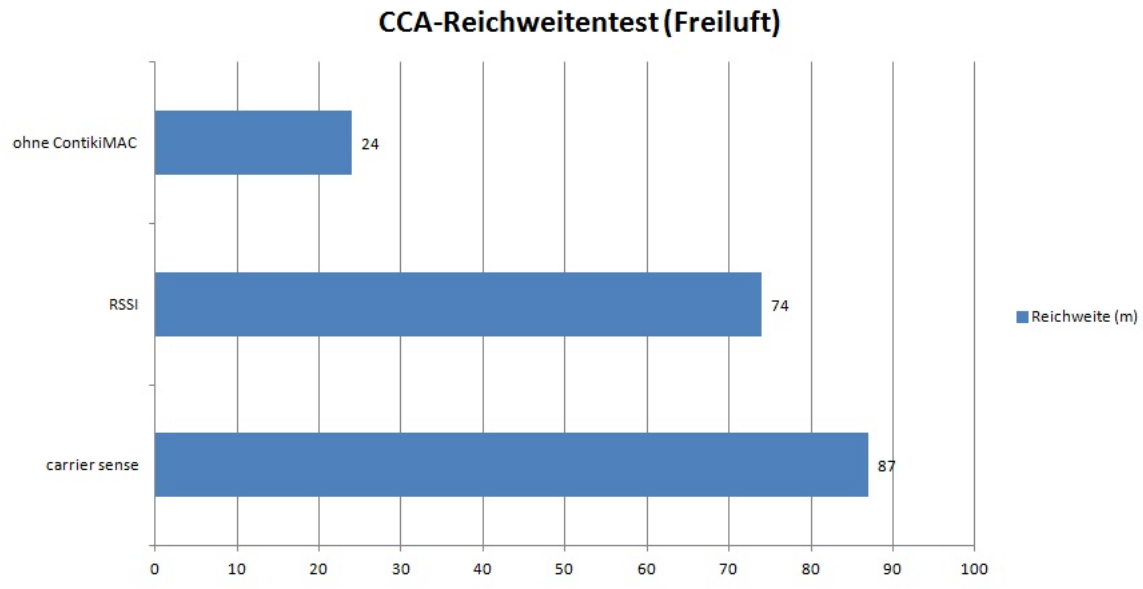


Abbildung 7: Reichweitentest

Eine Anomalie bei der Durchführung der Tests sticht sofort ins Auge. Mit deaktiviertem ContikiMAC hätte man die größtmögliche Reichweite erwartet. Im Test

trat genau das Gegenteil ein. Eine Erklärung hierfür konnte bislang nicht gefunden werden. Die Tests wurden mehrere Male durchgeführt und zeigten immer das gleiche Ergebnis.

Zufriedenstellend war jedoch, dass mit der neuen Methode (carrier sense) tatsächlich eine Verbesserung der Reichweite zu verzeichnen war. Insgesamt gesehen ist das Ergebnis aber immer noch unzureichend, da nach wie vor nicht der von Dresden Elektronik genannte Wert von 200 m erreicht werden konnte.

5 Strommessungen des Funkmoduls

Um die Laufzeit der im Forschungsseminar benutzten Sensorknoten deRFNode im Batteriebetrieb bestimmen zu können, wurden mit Hilfe eines Oszilloskops Strommessungen durchgeführt.
Dazu wurde auf dem Sensorknoten deRFNode der Jumper JP5 entfernt und ein Shunt-Widerstand eingesetzt, bei dem der Spannungsabfall mit dem Oszilloskop gemessen wurde *(siehe Abbildung 8)*.

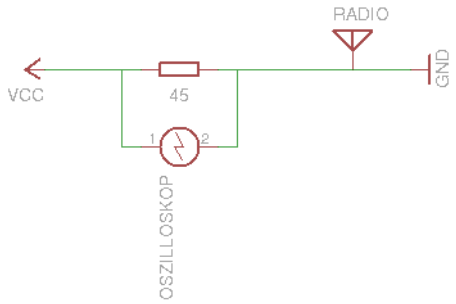


Abbildung 8: Messung der Stromstärke

Als Oszilloskop kam dabei das XMEGA Xprotolab von Gabotronics zum Einsatz *(Vgl. GABOTRONICS 2015, XMEGA Xprotolab)*.
Weiterhin wurden die Messungen mit dem RDC-Mechanismus ContikiMAC durchgeführt.
Ziel des Ganzen sollte es sein, zu evaluieren, weshalb die Lebensdauer der Sensorknoten im praktischen Einsatz so gering ist. Bei praktischen versuchen konnten bisher nur Laufzeiten von 1-2 Monaten mit 3 x AA 1900 mAh Akkus (NiMH, 1,2V) erzielt werden.

5.1 Strommessung

5.1.1 Durchführung

Wie in *A.1 Zyklische Aufwachphasen alle 8 Hz (125ms)* sehr gut zu sehen ist, wurde ContikiMAC so konfiguriert, dass alle 125 ms (8 Hz) geprüft wird, ob ein Signal für den Sensorknoten vorliegt (CCA-Phasen).
Weiterhin ist in *A.2 Wachphase des Radio-Moduls = 3,375ms* (t_r) erkennbar, dass eine CCA-Phase im Schnitt ca. 3,375 ms andauert und die Peaks bei einem 45 Ohm Widerstand einen Spannungsabfall von 320 mV verursachen.
Mit Hilfe des Ohmschen-Gesetzes:

$$R = \frac{U}{I} => I = \frac{U}{R}$$

kommt man auf eine Stromstärke von ca. 7 mA während der CCA-Phasen. In der übrigen Zeit liegt die Spannung bei ca. 10 mV, welche 0,22 mA entsprechen.
Bei den Messungen im Idle sind durchaus Messfehler zu erwarten, da normalerweise kein „zittern“, wie in den Bildern zu sehen ist, auftreten sollte. Hierbei scheint das benutzte Oszilloskop nicht sensibel genug zu sein.

5.1.2 Ergebnisse

Wenn man wieder einen Blick zu der Theorie der Timings von ContikiMAC wirft *(siehe Abschnitt 2.3 Timings)*, erkennt man eine große Diskrepanz in der Dauer der CCA-Phasen.
Wie lange die Messung der Energiestärke dauert, ist abhängig von der Hardware. Bei dem im Forschungsseminar benutzten Modul sollte solch eine Messung nur 0,192 ms betragen. Mit den gemessenen 3,375 ms überschreiten wir hier also sehr deutlich diesen Wert, was wiederum ein Indiz für eine geringe Batterielebensdauer ist.

Rechnet man die gemessenen Ströme auf 1 Stunde hoch so kommt man auf

7 mA für 97,2 s pro Stunde -> 2,7 % der Zeit
0,22 mA für 3502,8 s pro Stunde -> 97,3 % der Zeit

Somit ergibt sich eine durchschnittliche Stromaufnahme von

$$7\ \text{mA} * 0{,}027 + 0{,}22\ \text{mA} * 0{,}973 = 0{,}40306\ \text{mA}$$

Legt man nun eine Batterie mit 1900 mAh zugrunde, kommt man auf eine Lebenszeit von ca.

196 Tagen -> 6 Monate 16 Tage

Wichtig zu erwähnen sei noch, dass diese Messungen lediglich die Stromaufnahme des μControllers mitsamt des Funkmoduls berücksichtigen. Die reale Stromaufnahme sollte höher sein, da das eingesetzte Board deRFNode nicht vernachlässigt werden darf.
Weiterhin wird bei den berechneten Werten nur der Idle Betrieb mit ContikiMAC betrachtet.
Die tatsächliche Lebensdauer dürfte somit deutlich niedriger sein.
Dafür, dass der Sensorknoten keine Arbeit verrichtet und bei diesen Werten auch nicht auf Anfragen von Clients reagiert, sind 6 Monate für den praktischen Einsatz viel zu gering.

Im nachfolgenden Abschnitt wird genauer untersucht, wie es zu den Abweichungen der theoretischen CCA-Dauer und der praktischen CCA-Dauer *(t_r)* kommen kann.

5.2 Untersuchungen der CCA-Phasen

Im vorhergehenden Abschnitt wurde deutlich, dass einer der Gründe für eine erhöhte Stromaufnahme darin begründet ist, dass in der praktischen CCA-Phase das Radio-Modul viel zu lange aktiviert bleibt.

Um diesem Phänomen nachzugehen, wurde direkt vor und nach der CCA-Anfrage an die Hardware ein Ausgabepin des μController ein- bzw. ausgeschaltet und anschließend wieder mit einem Oszilloskop der Spannungsabfall über einem Widerstand gemessen *(Vgl. Abbildung 8)*.
Dazu wurde die Datei

cpu/avr/radio/rf230bb/rf230bb.c

entsprechend angepasst.

Listing 8: Messung einer CCA-Abfrage

```
1670 uint8_t volatile saved_sreg = SREG; // vorher: PIN HIGH
1671 sei( );
1672 hal_register_write(PHY_ED_LEVEL,0);
1673 while (rf230_ccawait) {}
1674 SREG = saved_sreg; // nachher: PIN LOW
```

Das Ergebnis dieser Messung kann dem *Anhang A.3 Dauer einer CCA-Abfrage = 0,125ms (t_r)* entnommen werden.
Dabei ist zu erkennen, dass die Hardware hierbei nicht das Problem darstellt, da die gemessene Zeit von 0,125 ms sogar besser als der theoretische Wert von 0,192 ms ist.
Weiterhin wurde die gesamte Dauer einer CCA-Routine gemessen. Dazu wurde, wie weiter oben schon beschrieben, an einem Ausgabepin die Spannungsspitzen gemessen. Um dies zu machen, geht man am besten direkt in die Implementierung von ContikiMAC und setzt jeweils vor und nach einer CCA-Abfrage den PIN auf HIGH bzw. LOW

core/net/mac/contikimac.c

Listing 9: Messung der gesamten CCA-Routine

```
420 if(NETSTACK_RADIO.channel_clear() == 0) {
421 packet_seen = 1;
422 break;
423 }
```

Dabei sieht man *(siehe A.4 Dauer der gesamten CCA-Routine = 0,250 ms)*, dass alleine durch die Abarbeitung der Routine, das Funkmodul für die doppelte Dauer (0,250 ms) der eigentlichen CCA-Abfrage (0,125 ms) eingeschaltet bleibt. Das

Funkmodul wird nämlich vor dem Aufruf der CCA-Routine eingeschaltet und danach wieder deaktiviert.
Theoretisch könnte das Funkmodul also viel länger deaktiviert bleiben. Dieser Umstand weist somit schon einmal auf einen erhöhten Stromverbrauch hin.

Zu guter Letzt wurde auch noch die Zeit zwischen zwei aufeinanderfolgenden CCA-Phasen gemessen *(siehe A.5 Dauer zwischen zwei CCA-Abfragen = 1 ms (t_c)).*
Eher durch Zufall kam dabei heraus, dass auch dieser Wert *(t_c)* nicht mit dem theoretischen Wert von 0,5 ms übereinstimmt. Er ist genau doppelt so hoch (1 ms).

Wie in dem folgenden Listing *(contikimac.c)* zu sehen ist, wird der Wert jedoch korrekt mit 2000 Hz (= 0,5 ms) definiert.

Listing 10: t_c mit 0,5 ms definiert

```
155 #define CCA_SLEEP_TIME RTIMER_ARCH_SECOND / 2000
```

Es liegt somit die Vermutung nahe, dass der genutzte Timer ein Problem darstellt.

Weitere Untersuchungen wurden aufgrund des Ende des Forschungsseminars nicht durchgeführt. Nachfolgende Gruppen sollten an dieser Stelle ansetzen, um das Phänomen des hohe Stromverbrauchs auf die Schliche zu kommen.

Des weiteren ist in *Anhang A.2* bei der Messung der Stromstärke nur ein langer Peak alle 125 ms aufgefallen. Erwartet hätte man ein ähnliches Verhalten wie in *Anhang A.3* mit zwei kurzen Peaks direkt hinter einander. Ein Indiz dafür, dass die Hardware hier vermutlich gar nicht ausgeschaltet (SLEEP) wird.

6 Änderungen am Contiki-Quellcode

6.1 CoAP-Ressourcen

Der Datei

project/coap-sensornode/coap-sensornode.c

wurden ein Batterie- sowie Temperatursensor als Ressourcen hinzugefügt (GET sowie Observe möglich).
(Siehe auch Abschnitt 3.1 und 3.2)

Alle Änderungen können den folgenden Commits entnommen werden:

https://github.com/HTWDD-SN/contiki/commit/39f0c8b5e705ceb588e69f27c67dd62b9e70feb0
https://github.com/HTWDD-SN/contiki/commit/6a909ff2626304da1cc885e7928eb4ea11f868b1

6.2 ADC

Zur einfachen Arbeit mit dem ADC und zum Lesen des internen Temperatursensors des Atmega128RFA1, wurde eine API zur Arbeit des ADC bereit gestellt.
Unter:

platform/avr-atmega128rfa1/dev/adc.c
platform/avr-atmega128rfa1/dev/adc.h

befindet sich die Implementierung, welche von einem Mitglied der Community bereits durchgeführt wurde und von hier stammt:

https://github.com/Louro/contiki-mirror/tree/contikimac128rfa1/platform/avr-atmega128rfa1/dev

Zur Nutzung muss lediglich die .h Datei eingebunden werden und der Methode *readADC(uint8_t pin)* der Pin des AD-Wandlers mitgeteilt werden.

Alle Änderungen können dem folgenden Commit entnommen werden:

https://github.com/HTWDD-SN/contiki/commit/6a909ff2626304da1cc885e7928eb4ea11f868b1

6.3 CCA mit carrier sense

Die Nutzung der alternativen Methode zur Bestimmung, ob ein Kanal frei ist oder nicht, musste in Contiki erst noch implementiert werden. Sie wurde für die Reichweitentests genutzt *(siehe Abschnitt 4.2)* und kann nun im Normalbetrieb verwendet werden.
Notwendige Änderungen dafür wurden in der Konfigurationsdatei für die eingesetzte Hardware

platform/avr-atmega128rfa1/contiki-conf.h

sowie in der Datei für das genutzte Funkmodul

cpu/avr/radio/rf230bb/rf230bb.c

vorgenommen.

In der Konfigurationsdatei wurde lediglich ein Schalter eingebaut, der es dem Anwender erlaubt zu entscheiden, welche Methode (RSSI oder carrier sense) bei einem CCA genutzt werden soll.

Listing 11: Schalter für RSSI bzw. carrier sense

```
286 /* whether we should use a treshhold in cca or if we use carrier
        sense mode (detect a valid 802.15.4 signal) */
287 //#define RF230_CONF_CCA_CARRIER_SENSE_ONLY 1
```

Weiterhin wurde die CCA-Routine *(rf230_cca)* des Radio-Treibers erweitert:

Listing 12: Implementierung carrier sense

```
1643 #ifdef RF230_CONF_CCA_CARRIER_SENSE_ONLY
1644 hal_subregister_write(SR_CCA_MODE,2);
1645 #endif
```

```
1662 #ifdef RF230_CONF_CCA_CARRIER_SENSE_ONLY
1663 // start manual cca
1664 hal_subregister_write(SR_CCA_REQUEST,1);
1665 #endif
```

```
1667 #ifndef RF230_CONF_CCA_CARRIER_SENSE_ONLY
```

```
1683 #endif
1684 #ifdef RF230_CONF_CCA_CARRIER_SENSE_ONLY
1685 while(hal_subregister_read(SR_CCA_DONE) == 0) {}
1686 // 1 = channel clear, 0 = channel bussy
1687 if(hal_subregister_read(SR_CCA_STATUS)) cca=0xff;
1688 #endif
```

```
1793 #ifdef RF230_CONF_CCA_CARRIER_SENSE_ONLY
1794 hal_subregister_write(SR_CCA_MODE,2); //carrier sense only
1795 #else
1796 hal_subregister_write(SR_CCA_MODE,1); //leave channel unchanged
1797 #endif
```

Alle Änderungen können dem folgenden Commit entnommen werden:

https://github.com/HTWDD-SN/contiki/commit/f43928a68b54041e614800fef0acaf3c9b881413

6.4 Bugfixing

Im fogenden werden einige Listings gezeigt, die Codeänderungen beinhalten, die das Beseitigen von compile Errors bzw. allgemeine Bugfixes am Contiki-Code beinhalten.
Fehler traten unter anderem nach dem Upgrade von Contiki 2.6 auf 2.7 auf, da die eingesetzte Hardwareplattform avr-atmega128rfa1 nicht mehr gepflegt wurde.

platform/avr-atmega128rfa1/Makefile.avr-atmega128rfa1

Listing 13: Bugfixes am Contiki-Code

```
31 +ifeq ($(UIP_CONF_IPV6),1)
32 + CFLAGS += -DUIP_CONF_IPV6=1
33 +endif
```

https://github.com/HTWDD-SN/contiki/commit/180196a365ac722d6557747f64142498f60b3e82
https://github.com/HTWDD-SN/contiki/commit/8114c26c4ad82487ba9e33c9c03dba46a51ba760

Mit aktiviertem *RF230_CONF_CCA_THRES* Wert gab es einen Fehler beim Compilieren (Klammer fehlte):

cpu/avr/radio/rf230bb/rf230bb.c

```
1674 + if (hal_register_read(RG_PHY_ED_LEVEL)<(91+RF230_CONF_CCA_THRES))
         cca=0xff;
```

https://github.com/HTWDD-SN/contiki/commit/c40a48472fbff9d4b2b8d7fff1bbc646d7250616

Beseitigen der Fehler, für Paketverlust und zu kurze Reichweite:

platform/avr-atmega128rfa1/contiki-conf.h

```
251 +#define CONTIKIMAC_CONF_WITH_PHASE_OPTIMIZATION 0
```

https://github.com/HTWDD-SN/contiki/commit/7e904fd0b0bbac72433efe8e006edbfd2c4a7261

```
282 +/* CCA theshold energy -91 to -61 dBm (default -77). Set this
        smaller than the expected minimum rssi to avoid packet
        collisions */
283 +/* The Jackdaw menu 'm' command is helpful for determining the
        smallest ever received rssi */
284 +#define RF230_CONF_CCA_THRES -90
```

https://github.com/HTWDD-SN/contiki/commit/eefb7c52c858f9c86ecb0ecf31a93607f7d2324c

Des weiteren mussten die Pins für die LEDs invertiert werden, damit bei einer CoAP-Anfrage zum Einschalten einer LED, diese auch eingeschaltet wird und nicht aus:

platform/avr-atmega128rfa1/contiki-main.c

```
259 + leds_invert(LEDS_YELLOW);
260 + leds_invert(LEDS_RED);
261 + leds_invert(LEDS_GREEN);
```

https://github.com/HTWDD-SN/contiki/commit/387edf36f177cffefb85d16bee4e78864f471121

7 Zusammenfassung und Ausblick

Während des Forschungsseminars wurde das Thema der Hausautomatisierung durch den Einsatz batteriebetriebener Sensoren/Aktoren näher beleuchtet. Auf diesem Gebiet gibt es noch eine Menge zu erledigen. Ein wichtiger Punkt ist die Sicherheit (Security), die bisher nicht berücksichtigt wurde. Auch die Entwickler des zugrunde liegende Betriebssystem Contiki nehmen sich diesem Thema mit Contiki 3.0 an. Diesbezüglich sollte man den Entwicklungsfortschritt genau verfolgen und zu gegebener Zeit von Contiki 2.7 auf 3.0 wechseln.

Des weiteren traten immer wieder Probleme mit der Reichweite der eingesetzten Hardware auf. Auch die Untersuchungen zu der hohen Stromaufnahme der Module wurden noch nicht vollends abgeschlossen. An diesen beiden Punkten muss nach wie vor gearbeitet werden.
Man sollte, was die Reichweite angeht, Tests durchführen, die unabhängig von Contiki ablaufen. Dresden Elektronik bietet Beispielsoftware an, mit der man unabhängig von Contiki testen kann.
Sollte man bei diesen Tests auch keine Erfolge verzeichnen, ist es ratsam über einen Wechsel der zugrunde liegenden Hardware nachzudenken.

8 Literaturverzeichnis

Adam Dunkels (2013): The ContikiMAC Radio Duty Cycling Protocol (Präsentation).
http://www.wsnmagazine.com/ContikiMAC_F13.pdf [Eingesehen 30.01.2015]

Adam Dunkels (2011): The ContikiMAC Radio Duty Cycling Protocol (Paper).
http://soda.swedish-ict.se/5128/1/contikimac-report.pdf [Eingesehen 30.01.2015]

Atmel (2014): ATmega128RFA.
http://www.atmel.com/Images/Atmel-8266-MCU_Wireless-ATmega128RFA1_Datasheet.pdf [Eingesehen 30.01.2015]

Contiki (2014): Change mac or radio duty cycling protocols.
https://github.com/contiki-os/contiki/wiki/Change-mac-or-radio-duty-cycling-protocols [Eingesehen 30.01.2015]

Dresden Elektronik (2014): deRFbreakout Board.
http://www.dresden-elektronik.de/funktechnik/products/boards-and-kits/development-boards/derfbreakout-board/ [Eingesehen 30.01.2015]

Dresden Elektronik (2014): deRFnode.
http://www.dresden-elektronik.de/funktechnik/produkte/boards-und-kits/development-boards/derfnode/ [Eingesehen 30.01.2015]

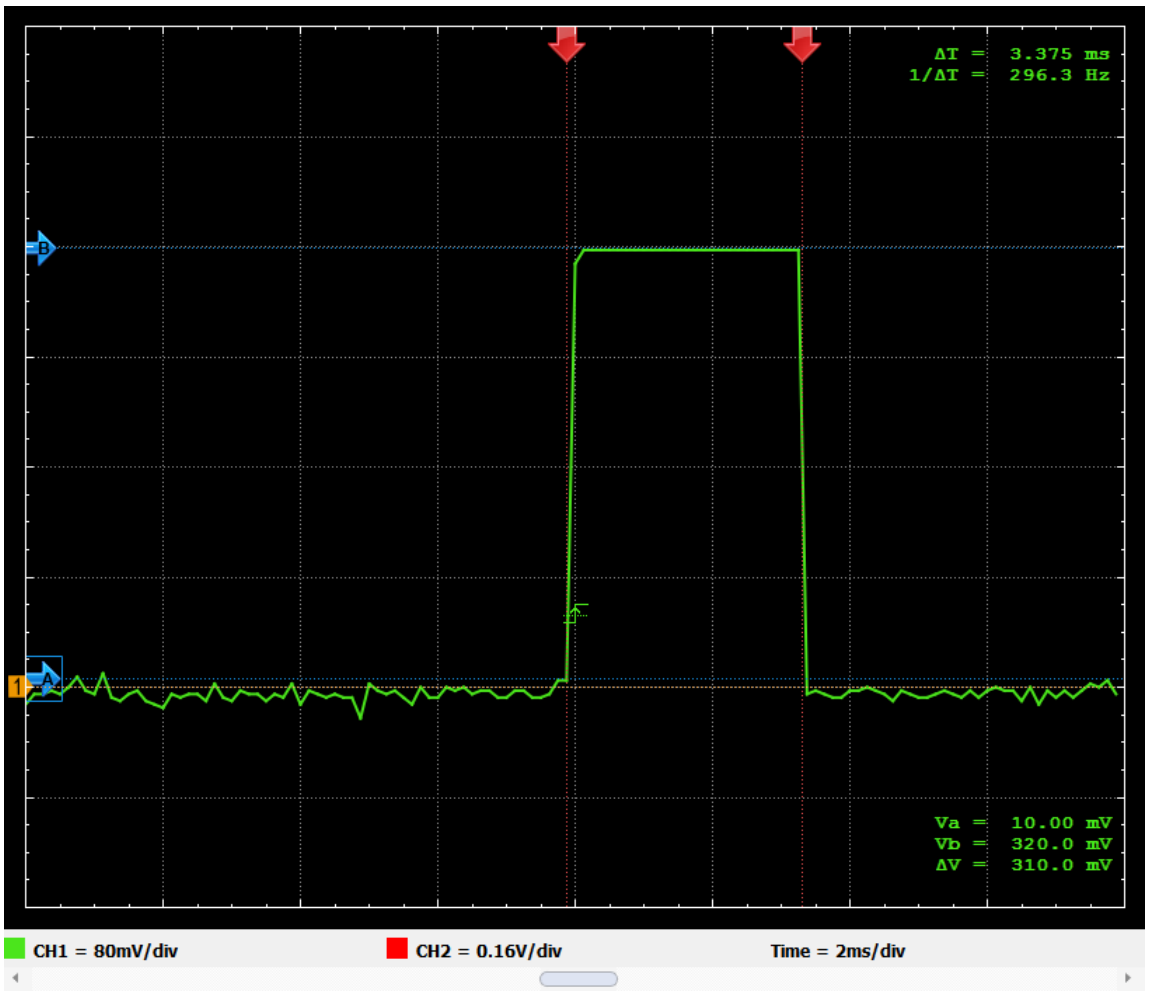
Dresden Elektronik (2014): Evaluations-Module deRFmega128.
http://www.dresden-elektronik.de/funktechnik/products/radio-modules/avr-single-chip-modules/ [Eingesehen 30.01.2015]

Gabotronics (2015): XMEGA Xprotolab.
http://www.gabotronics.com/development-boards/xmega-xprotolab.htm [Eingesehen 05.02.2015]

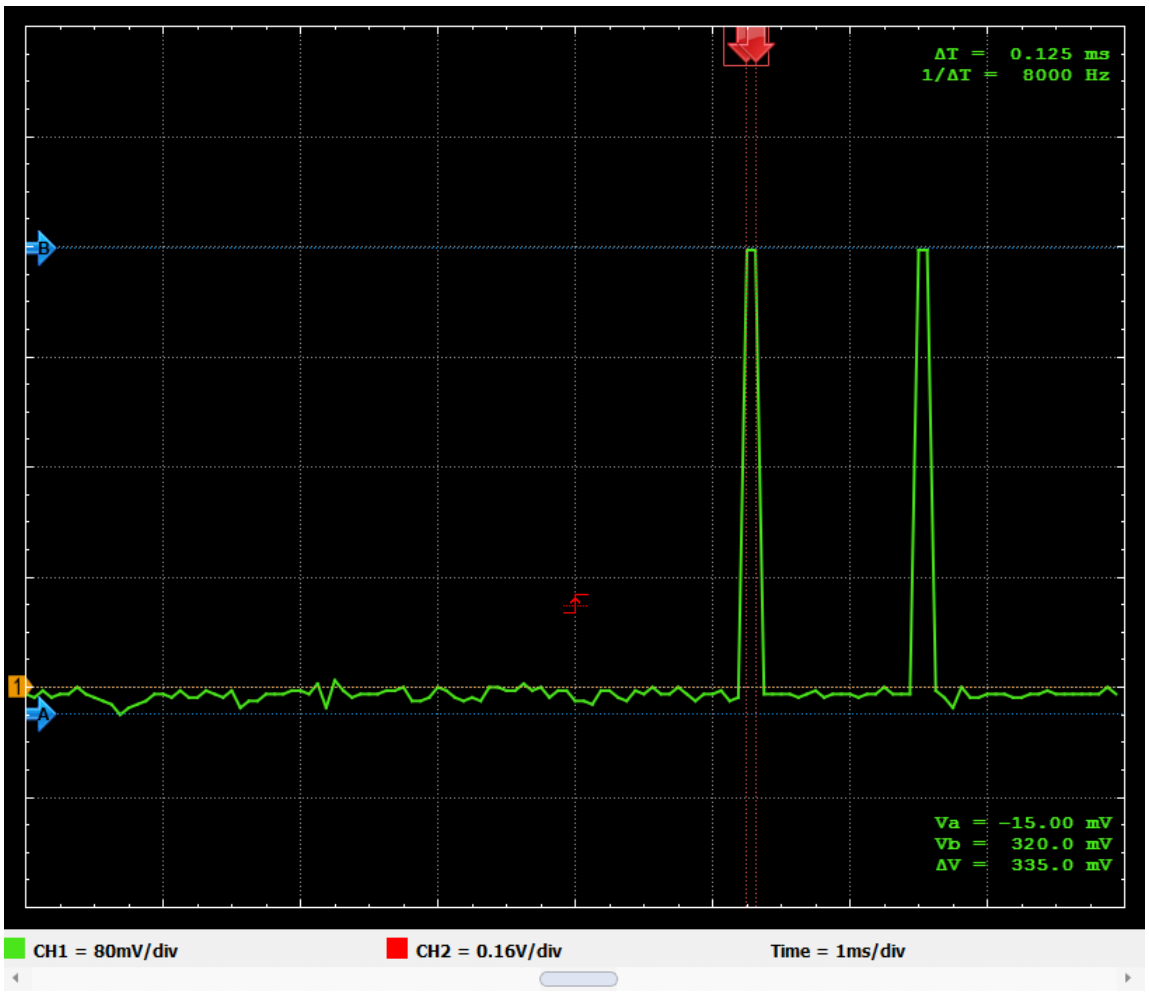
A Anhang

A.1 Zyklische Aufwachphasen alle 8 Hz (125ms)

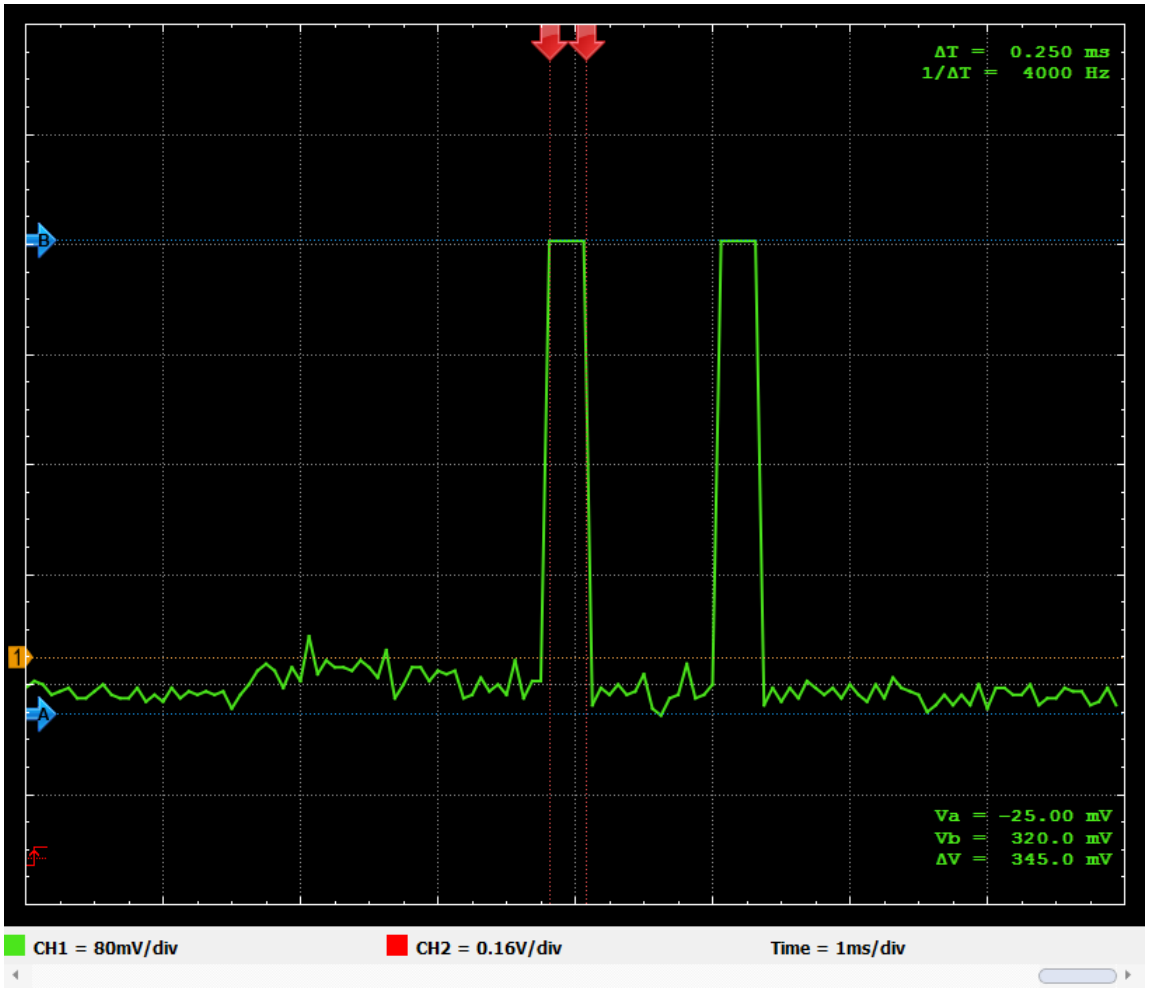
A.2 Wachphase des Radio-Moduls = 3,375ms (t_r)



A.3 Dauer einer CCA-Abfrage = 0,125ms (t_r)



A.4 Dauer der gesamten CCA-Routine = 0,250 ms



A.5 Dauer zwischen zwei CCA-Abfragen = 1 ms (t_c)