Abschlussbericht für das Forschungsseminar Sensornetze Wintersemester 16/17

im Master-Studiengang Angewandte Informationstechnologien

eingereicht von:

Alexander Gräb

Betreuer: Herr Prof. Dr.-Ing. Jörg Vogt

Eingereicht: 28.03.2017

Inhalt

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

CoAP Constrained Application Protocol

RSSI Received Signal Strength Indicator

CCR Channel Check Rate

RDC Radio Duty Cycling

CCA Clear Channel Assessment

REST Representational State Transfer

RPL Routing Protocol for Low power and Lossy Networks

1 Einleitung

Dies ist der Abschlussbericht zum Forschungsseminar *Sensornetze* an der HTW Dresden unter der Leitung von Herrn Prof. Dr.-Ing. Jörg Vogt. Ziel des Forschungsseminars ist es, herauszufinden, inwiefern sich Sensornetze auf Basis von offenen Standards und Internet-Technologien realisieren lassen. Die konkreten Anwendungsfälle für Sensornetze stehen dabei zunächst im Hintergrund. Dieser Bericht ist die Fortsetzung des Zwischenberichts[1] von Kai Richter und Alexander Gräb vom Sommersemester 2016 (siehe [11]).

Der Fokus des Autors im Wintersemester 16/17 lag stark auf dem Sparen von Energie bei hinreichender Qualität der Funkverbindung. In Kap. 2 wird auf die Änderungen im Contiki-Quellcode eingegangen, welche während des Wintersemesters 16/17 gemacht wurden. In Kap. 3 werden die Erkenntnisse beschrieben, welche aus Messungen und Experimenten resultiert sind. Kap. 4 spricht offene Probleme an, welche während des Semesters nicht gelöst wurden und Kap. 5 gibt einen Ausblick darauf, welche Probleme und Fragestellungen zukünftig im Forschungsseminar angegangen werden könnten.

[1] `https://www2.htw-dresden.de/~wiki_sn/images/8/84/ZwischenberichtSS16_RichterGraeb.pdf`

2 Modifikationen in den Quellen von Contiki 3.0 gegenüber der Original-Quellen

Im Zwischenbericht aus dem Sommersemester 2016 wurden bereits alle Änderungen im Quellcode von Contiki 3.0 gegenüber der Original-Quellen dokumentiert (siehe [11, S. 14 ff.]). Die HTW Dresden verwaltet diese Änderungen in einem eigenen GitHub-Repository (`https://github.com/HTWDD-SN/contiki`). Dieses Kapitel dokumentiert die Änderungen, welche im Wintersemester 16/17 hinzugekommen sind. Am Ende jedes Unterkapitels ist die ID des entsprechenden Commits auf GitHub, sowie ein Link, über welchen die Änderungen auf GitHub detailliert betrachtet werden können, aufgeführt.

2.1 Channel-Check-Raten von 4 und 2 Hz

Im Forschungsseminar wird das Radio Duty Cycling (RDC)-Protokoll *ContikiMAC* eingesetzt, um die Sensorknoten (Nodes) sparsamer zu betreiben, damit sie möglichst lange mit einem Satz Batterien bzw. einer Akku-Ladung auskommen. Dies wird erreicht, indem die Funkeinheit so lange wie möglich deaktiviert bleibt. Zyklisch wird die Funkeinheit aktiviert, um einen sogenannten Clear Channel Assessment (CCA) durchzuführen. Dabei wird festgestellt, ob Daten empfangen werden können. Mit anderen Worten, die Node überprüft, ob eine anderer Node im Sensornetzwerk Daten an sie sendet. Ist dies der Fall, bleibt die Funkeinheit so lange aktiv, bis die Daten empfangen wurden und eine Antwort gesendet wurde. Andernfalls wird die Funkeinheit unverzüglich wieder deaktiviert. Wie oft der CCA durchgeführt wird, wird über die sog. CCR in Hz angegeben. Der Standardwert von Contiki für die CCR beträgt 8 Hz. D. h. also, dass acht mal in der Sekunde ein CCA durchgeführt wird. Die CCR kann über die Konfigurations-Konstante `NETSTACK_CONF_RDC_CHANNEL_CHECK_RATE` in der Datei *platform/avr-atmega128rfa1/contiki-conf.h*[2] angepasst werden.

CCRs von unter acht Hz sind nicht ohne weiteres möglich. Zwar lassen sich solche Werte angeben, ohne dass es einen Fehler beim Kompilieren gibt, jedoch werden niedrigere Werte für die CCR einfach ignoriert. Dies liegt darin begründet, dass der Hardware-Timer der verwendeten Plattform (*ATmega128RFA1*) nur über ein 8-Bit-Zählregister verfügt. Somit muss ein Kompromiss zwischen der Granularität des Timers (also der kleinsten überbrückbaren Zeitspanne) und der längsten überbrückbaren Zeitspanne geschlossen werden. Mit den Änderungen sind nun Zeitspannen zwischen zwei Millisekunden und 0,5 Sekunden möglich.

Commit: f8fdd14
`https://github.com/HTWDD-SN/contiki/commit/f8fdd14659af8ab5bc010e4ef514f8655edd018e`

[2] Pfadangaben sind stets relativ zum Wurzelverzeichnis des Contiki-Quellcode-Baumes.

2.2 Wake-Up-Cycle entfernt

Der Durchschnittliche Stromverbrauch mit aktiviertem ContikiMAC war unter Contiki 3.0 mit ca. einem mA (vgl. [11, S. 28]) deutlich höher, als unter Contiki 2.7 (0,24 mA, vgl. [8]). Dies war einem programmierten Wake-Up-Cycle zu verschulden, welcher alle 16 CCA-Phasen verhinderte, dass die Node wieder in den energiesparenden Schlafmodus wechselt. Den Kommentaren an der betroffenen Stelle im Quellcode ist zu entnehmen, dass dies wohl getan wurde, um Fordergrundaktivitäten zu gestatten. Der Wake-Up-Cycle tritt nun nicht mehr ein, wenn die Konstante `CONTIKIMAC_CONF_NO_WAKE_CYCLE` definiert ist. Von der möglichkeit den Wake-Up-Cycle zu deaktivieren wird seit diesem Commit gebrauch gemacht. Mit deaktiviertem Wake-Up-Cycle beträgt der Stromverbrauch nur noch etwa 0,33 mA.

Commit: 65eeb7c
`https://github.com/HTWDD-SN/contiki/commit/65eeb7cb269f3a48c987651b466f33bbdc6ca6d0`

2.3 Sensoren und andere CoAP-Ressourcen

Der Mikrocontroller ATmega128RFA1 sowie das Board deRFnode verfügen über Sensoren. Die Werte dieser Sensoren und andere Informationen können über das Constrained Application Protocol (CoAP) abgerufen werden (vgl. [4]). Es gibt periodische und nicht periodische CoAP-Ressourcen.[3] Periodische CoAP-Ressourcen unterstützen eine sog. *Subscription*. D. h., ein Client kann sich bei einer Ressource einschreiben, um sich aktiv mit Informationen versorgen zu lassen. Dies ist z. B. sinnvoll, um sich von einem Sensorknoten über einen geänderten Sensorwert informieren zu lassen, anstatt immer wieder aktiv anzufragen (Polling). Dennoch bieten viele CoAP-Ressourcen, aus in den Unterkapiteln genannten Gründen, keine Subscription an. CoAP-Ressourcen stehen derzeit nur bei Sensor-Nodes, also dem Softwareprojekt *er-rest-example* (vgl. [11, S. 14]), zur Verfügung. Der Border-Router verfügt über keine CoAP-Ressourcen. Zum Abrufen von Informationen via CoAP kann z. B. das Programm *coap-client*, welches mit der Software-Bibliothek *libcoap*[4] mitgeliefert wird, verwendet werden (siehe [3]). Eine besondere CoAP-Ressource verbirgt sich hinter der URL */.well-known/core*. Sie liefert Informationen über alle verfügbaren CoAP-Ressourcen. Listing 1 zeigt die Verwendung des Programms coap-client, um diese Ressource abzufragen. Man beachte, dass die erste Zeile umbrochen ist. An der Ausgabe kann man erkennen, dass auch noch andere, als die hier beschriebenen Ressourcen, vorhanden sind. Z. B. die Ressource */test/hello*, welche bereits implementiert war. Alle Ressourcen werden über das Representational State Transfer (REST)-Paradigma bereitgestellt (siehe [15]). Es gibt die Operationen *GET*, *POST*, *PUT*, *DELETE* usw. Um Informationen abzurufen, wird die Operation GET (in Listing 1 über den Parameter `-m get` angegeben) verwendet.

[3] Jede CoAP-URL wird als Ressource bezeichnet.
[4] `https://sourceforge.net/projects/libcoap/files/`

```
root@OpenWrt:~# /root/bin/coap-client -m get coap://[2002:8d38:831a:aaaa:221:2eff
    :ff00:225d]/.well-known/core
v:1 t:0 tkl:0 c:1 id:64713
</.well-known/core>;ct=40,</testv:1 t:0 tkl:0 c:1 id:64714
/hello>;title="Hello world: ?lenv:1 t:0 tkl:0 c:1 id:64715
=0..";rt="Text",</test/push>;titv:1 t:0 tkl:0 c:1 id:64716
le="Periodic demo";obs,</sensorsv:1 t:0 tkl:0 c:1 id:64717
/tmp102_temp>;title="Temperaturev:1 t:0 tkl:0 c:1 id:64718
";rt="Temperature";obs,</sensorsv:1 t:0 tkl:0 c:1 id:64719
/battery>;title="Battery voltagev:1 t:0 tkl:0 c:1 id:64720
";rt="Battery";obs,</sensors/lumv:1 t:0 tkl:0 c:1 id:64721
inosity>;title="ISL29020 luminosv:1 t:0 tkl:0 c:1 id:64722
ity in [Lux]";rt="Luminosity-Luxv:1 t:0 tkl:0 c:1 id:64723
";obs,</sensors/acceleration>;tiv:1 t:0 tkl:0 c:1 id:64724
tle="BMA150 acceleration in [g]"v:1 t:0 tkl:0 c:1 id:64725
;rt="Acceleration-g";obs,</sensov:1 t:0 tkl:0 c:1 id:64726
rs/rssi>;title="RSSI statistic";v:1 t:0 tkl:0 c:1 id:64727
rt="RSSI";obs,</info/uptime>;titv:1 t:0 tkl:0 c:1 id:64728
le="System uptime";rt="Uptime";ov:1 t:0 tkl:0 c:1 id:64729
bs,</sensors/cpu_temp>;title="ATv:1 t:0 tkl:0 c:1 id:64730
mega128RFA1 internal temperaturev:1 t:0 tkl:0 c:1 id:64731
 sensor";rt="CPU temperature";obv:1 t:0 tkl:0 c:1 id:64732
s
```

Listing 1: CoAP-Ressource /.well-known/core

Listing 2 zeigt die Verwendung einer Subscription anhand der, bereits vorimplementierten, Ressource */test/push*. Bei einer Subscription zu dieser Ressource sendet die Node alle fünf Sekunden die Zeichenkette „VERY LONG EVENT“ mit dem Wert eines Zählers am Ende, welcher ständig inkrementiert wird. Der Parameter *-s 300* bewirkt eine Subscription zu dieser Ressource für eine Dauer von 300 Sekunden.[5]

```
root@OpenWrt:~# /root/bin/coap-client -s 300 coap://[2002:8d38:831a:aaaa:221:2eff:
    ff00:225d]/test/push
v:1 t:0 tkl:0 c:1 id:36519
VERY LONG EVENT 145671VERY LONG EVENT 145672VERY LONG EVENT
    145673VERY LONG EVENT 145674VERY LONG EVENT 145675VERY LONG
    EVENT 145676VERY LONG EVENT 145677VERY LONG EVENT 145678VERY
    LONG EVENT 145679VERY LONG EVENT 145680VERY LONG EVENT 145681
    VERY LONG EVENT 145682VERY LONG EVENT 145683VERY LONG EVENT
     145684VERY LONG EVENT 145685VERY LONG EVENT 145686VERY LONG
    EVENT 145687VERY LONG EVENT 145688
```

Listing 2: Demonstration der periodischen CoAP-Ressource /test/push

Abb. 2.1 auf der nächsten Seite fasst alle, während des Forschungsseminars implementierten, CoAP-Ressourcen zusammen.

[5] Tatsächlich hält die Subscription viel kürzer an, als angegeben. Hierbei handelt es sich möglicherweise um einen Fehler des Programms coap-client.

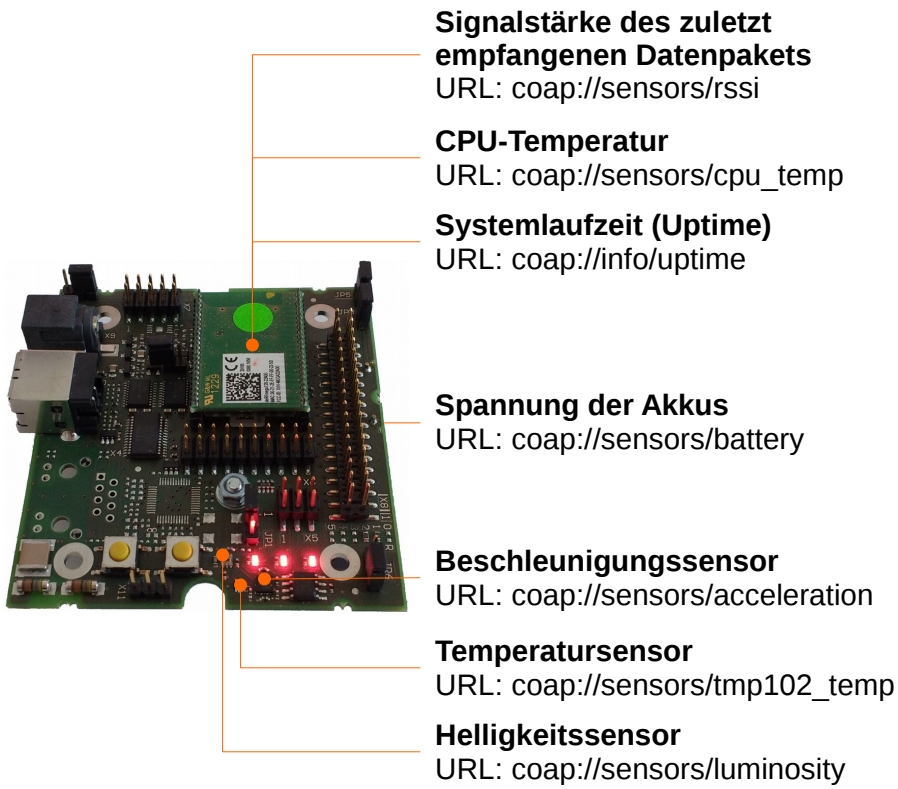

Abb. 2.1: Die CoAP-Ressourcen einer Node
Quelle: Eigene Darstellung

2.3.1 Batterie- bzw. Akku-Spannung

Die Spannung der Batterien bzw. Akkus kann über die URL */sensors/battery* abgerufen werden. Die Ressource ist nicht periodisch, d. h. Subscriptions sind nicht möglich.

Commit: 0ecc5ae

`https://github.com/HTWDD-SN/contiki/commit/0ecc5aeb0be661c23ae0727c536048940017aae9`

2.3.2 Externer Temperatursensor

Es handelt sich bei diesem Temperatursensor um einen extern, über die I^2C-Schnittstelle[6], angebundenen Chip (TMP102AIDRLT), welcher auf dem Board deRFnode verlötet ist. Der Sensor liefert die Temperatur auf einen halben Grad Celius genau (vgl. [10, S. 1]). Subscriptions sind wegen eines ungeklärten Fehler nicht möglich (siehe Kap. 4.1). Die URL lautet */sensors/tmp102_ temp*.

Commit: 0ecc5ae und 6ec14c8

`https://github.com/HTWDD-SN/contiki/commit/0ecc5aeb0be661c23ae0727c536048940017aae9`

`https://github.com/HTWDD-SN/contiki/commit/6ec14c8d1eebcf365424b628a7cc8e701caffb20`

[6] I^2C ist eine Schnittstelle, welche hauptsächlich für die Kommunikation verschiedener Teile einer Schaltung verwendet wird (siehe [14]).

2.3.3 Externer Helligkeitssensor

Es handelt sich hier um einen extern, über die I^2C-Schnittstelle, angebundenen Helligkeitssensor. Er ist in Form eines Chips (ISL29020IROZ-T7) auf dem Board deRFnode verlötet. Subscriptions sind nicht möglich (siehe Kap. 4.1). Der Helligkeits-Wert wird in Lux geliefert. Die URL lautet */sensors/luminosity*.

Commit: 0ecc5ae und 6ec14c8
`https://github.com/HTWDD-SN/contiki/commit/0ecc5aeb0be661c23ae0727c536048940017aae9`
`https://github.com/HTWDD-SN/contiki/commit/6ec14c8d1eebcf365424b628a7cc8e701caffb20`

2.3.4 Externer Beschleunigungssensor

Wie die anderen externen Sensoren auch, ist dieser Sensor über die I^2C-Schnittstelle angebundenen und sitzt in Form eines Chips (BMA150) auf dem Board deRFnode. Die Ressource erlaubt Subscriptions, jedoch gibt es auch hier wieder einen Fehler (siehe Kap. 4.1). Der Sensor gibt die Beschleunigung für drei Richtungen (X-, Y- und Z-Achse) in g (Fallbeschleunigung, $1g \approx 9,81m/2^2$) an. Der Beschleunigungssensor kann als Lagesensor verwendet werden. Liegt die Node bspw. mit dem Batterie-Fach auf einer gerade Fläche, betragen die Werte für die X- und Y-Achse 0 g und für die Z-Achse 1 g. Die URL lautet */sensors/acceleration*.

Commit: 0ecc5ae und 6ec14c8
`https://github.com/HTWDD-SN/contiki/commit/0ecc5aeb0be661c23ae0727c536048940017aae9`
`https://github.com/HTWDD-SN/contiki/commit/6ec14c8d1eebcf365424b628a7cc8e701caffb20`

2.3.5 Letzten RSSI-Wert abrufen

Der Received Signal Strength Indicator (RSSI)-Wert gibt die Signalstärke in dBm an, mit welcher das letzte Datenpaket empfangen wurde. Es sind Werte zwischen 0 (ausgezeichnet) und -90 (gerade noch ausreichend) möglich. Die Ressource ist periodisch. Bei einer Subscription wird alle zehn Sekunden ein Wert geliefert. Die URL der Ressource lautet */sensors/rssi*.

Commit: 487e788 und 7ae2a34
`https://github.com/HTWDD-SN/contiki/commit/487e788a4b890e73d41477668b7d34508902c454`
`https://github.com/HTWDD-SN/contiki/commit/7ae2a34402a5b5538025b8e6b6615d8afc477b28`

2.3.6 Uptime

Die Laufzeit (Uptime) einer Node kann über die CoAP-URL */info/uptime* abgerufen werden. Die Ausgabe hat das Format „[a] d [b] h [c] m [d] s ([x]/[y] s)“. Wobei x und y die Systemlaufzeit in Sekunden ist. x wird anhand des sog. *RTimers* von Contiki ermittelt. Der

RTimer ist ein spezieller Timer, welcher selbst dann läuft, wenn sich der Mikrocontroller im Schlafmodus befindet. Für y wird ein anderer Timer verwendet, welcher jedoch nicht im Schlafmodus läuft. x wird verwendet, um die Systemlaufzeit auszurechnen, wenn ein RDC-Protokoll wie ContikiMAC verwendet wird. y wird verwendet, wenn kein RDC-Protokoll genutzt wird, bzw., wenn dieses die Funkeinheit nicht deaktiviert (siehe Kap. 2.4). Listing 3 zeigt ein Beispiel für die Verwendung der Ressource. Die Uptime-Ressource ist nicht besonders genau. Eine Versuchs-Node hing der tatsächlich vergangenen Zeit nach ca. 86 Tagen um vier Stunden und drei Minuten hinterher. Die Ressource ermöglicht keine Subscriptions, ihre URL lautet */info/uptime*.

```
root@OpenWrt:~# /root/bin/coap-client coap://[2002:8d38:831a:aaaa:221:2eff:ff00:225
    e]/info/uptime
v:1 t:0 tkl:0 c:1 id:15924
1 d 0 h 34 m 30 s (88470/1854 s)
```

Listing 3: Beispiel für die CoAP-Ressource /info/uptime

Commit: 2702fc4, 7aaeb81, 29607c2, 8cae9db und 9decf53

`https://github.com/HTWDD-SN/contiki/commit/2702fc45ac6631690340afaf34fa84fe23847df7`

`https://github.com/HTWDD-SN/contiki/commit/7aaeb817548b95bd87b592fbe9ab66b35dd647db`

`https://github.com/HTWDD-SN/contiki/commit/29607c295800455296b0ce511b5cc85d19f554e9`

`https://github.com/HTWDD-SN/contiki/commit/8cae9db86deb06d13dbbb694bc68a80588d20812`

`https://github.com/HTWDD-SN/contiki/commit/9decf53f4c971bef25917270d157c2efdb26331c`

2.3.7 Interner Temperatursensor

Der Mikrocontroller ATmega128RFA1 verfügt über einen internen Temperatursensor. Er kann über die CoAP-URL `/sensors/cpu_temp` abgerufen werden und liefert die Temperatur in Grad Celsius. Die Ressource unterstützt Subscriptions. Typische Implementierungen von Ressource mit Subscription-Unterstützung prüfen den Sensor-Wert zyklisch, z. B. jede Minute und benachrichtigen die Subscriber[7], wenn sich der Sensor-Wert geändert hat, jedoch spätestens nachdem eine obere Zeitschranke abgelaufen ist. Ferner gibt es häufig noch eine untere Zeitschranke, um zu verhindern, dass die Subscriber nach jeder Änderung des Sensor-Wertes benachrichtigt werden. Diese Ressource ist derart implementiert, dass die Subscriber jede Minute einen Wert erhalten.

Commit: da480f3

`https://github.com/HTWDD-SN/contiki/commit/da480f3e00c6b5bf4a7b54f42476bf0fee3bfb50`

2.4 Button 1 verhindert das Deaktivieren der Funkeinheit

Wie schon in Kap. 2.1 beschrieben, deaktivieren RDC-Protokolle wie ContikiMAC die Funkeinheit zyklisch, um Energie zu sparen. Dieses Verhalten kann temporär unterbunden

[7]Subscriber nennen sich alle Clients, welche eine Subscription angefordert haben.

werden. Die Funkeinheit bleibt bis zum nächsten Neustart immer aktiv, wenn der Button 1 auf dem Board deRFnode (siehe Abb. 2.2) während der Inbetriebnahme gedrückt gehalten wird. So kann bspw. getestet werden, ob die Verbindungsqualität in bestimmten Situationen ohne Verwendung eines RDC-Protokolls besser wäre.

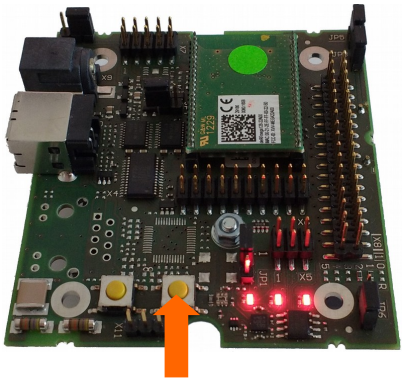

Abb. 2.2: Der Button 1 auf dem Board deRFnode
Quelle: Eigene Darstellung

Commit: da480f3

`https://github.com/HTWDD-SN/contiki/commit/da480f3e00c6b5bf4a7b54f42476bf0fee3bfb50`

3 Test- und Messergebnisse

3.1 Stromverbrauch

Der Stromverbrauch wurde bereits im Sommersemester 2016 ermittelt und die Ergebnisse im Zwischenbericht dargelegt (siehe [11, S. 23 ff.]). Durch Anpassung des Contiki-Quellcodes, vorallem durch die in Kap. 2.2 beschriebene Verbesserung, konnte der Stromverbrauch von ungefähr einem mA auf rund 0,33 mA gesenkt werden. Ebenfalls wurde der Stromverbrauch mit verschiedenen Werten für die CCR gemessen. Detaillierte Messergebnisse sind im Wiki zum Forschungsseminar zu finden (siehe [9]).

Tabelle 3.1 auf Seite 11 fasst die wichtigsten Ergebnisse zusammen. Die Messungen wurden mit verschiedenen Oszilloskop-Einstellungen und den daraus resultierenden unterschiedlichen Dauern einer Messung, durchgeführt. Die Messungen wurden über den Jumper JP5 des Boards deRFnode vorgenommen, um nur den Stromverbrauch des Funkmoduls deRFmega128-22A00 zu erfassen. Jede Messung liefert 32 Werte, da die Software des Oszilloskops 32 Speicherplätze vorhält, welche als Einzelmessungen für die numerische Berechnung exportiert werden. Die Werte der 32 Einzelmessungen sind nicht in Tabelle 3.1 zu sehen, jedoch deren Durchschnitte und Medianwerte. Die vorletzte Zeile zeigt nochmals den Durchschnitt über die Durchschnitts- und Medianwerte. Man erkennt, dass Durschnitts- und Medianwerte oft stärker voneinander abweichen. Dies liegt sehr wahrscheinlich an dem regelmäßigen Versand von Routing Protocol for Low power and Lossy Networks (RPL)-Datenpaketen (siehe Kap. 3.2).

Ferner ist anzumerken, dass der Stromverbrauch mit Contiki 3.0 leicht höher ist, als mit Contiki 2.7 (Software-Stand der HTW Dresden auf GitHub). Bei einer CCR von 8 Hz ergibt sich mit Contiki 2.7 ein durchschnittlicher Stromverbrauch von 0,24398 mA gegenüber von Contiki 3.0 mit 0,33228 mA. Auffällig ist auch, dass Contiki 2.7 ein anderes RPL-Kommunikationsverhalten zeigt (siehe Kap. 3.2).

3.1.1 Experiment mit einem Gold-Cap-Kondensator

Mit einem Gold-Cap-Kondensator sollten die in Tabelle 3.1 bestimmten Werte für den Stromverbrauch verifiziert werden. Gold-Cap-Kondensatoren sind Kondensatoren mit einer sehr hohen Kapazität aber dafür meist sehr geringen Spannungsfestigkeit. Der Kondensator sollte als Ersatz für einen Akku mit sehr geringer Kapazität dienen. Zwar wurde bereits im Zwischenbericht ein Akku mit einer Kapazität von lediglich 150 mAh verwendet (vgl. [1, S. 29]), jedoch muss man mehrere Tage warten, bis sich der Akku entladen hat und die Betriebszeit damit feststeht.
Für das Experiment wurde ein Gold-Cap Kondensator mit einer Kapazität von 1 F und einer Spannungsfestigkeit von 5,5 V gewählt. Die Kapazität lässt sich in mAh umrechnen. So ergibt sich für 1 F und einer Ladespannung von 5,5 V: $1F \cdot 5,5V = 5,5C = 5,5As \Rightarrow$

$5,5As \cdot \frac{1}{3600} = 0,001528Ah = 1,528mAh$ Dabei handelt es sich allerdings um die nutzbare Kapazität der gesamten Kondensatorladung. Während der Entladung sink die Spannung ab. Da der Supervisor des Bords deRFnode (wenn der Jumper JP4 geschlossen ist) einen Reset bei einer Betriebsspannungen unterhalb von 2,4 V auslöst (vgl. [13, S. 6]) und die Node damit ihren Betrieb einstellt, lässt sich die Kapazität des Kondensators nicht voll nutzen. Daher muss noch die nicht nutzbare Kapazität errechnet werden und von der oben errechneten Kapazität abgezogen werden: $(5,5V - 2,4V) \cdot 1F \cdot \frac{1}{3600} = 8,6111 \cdot 10^{-4}Ah = 0,86111mAh$

Die Betriebszeit wurde mit Hilfe des Skriptes in Listing 4 getestet. Das Skript prüft mit Ping, ob die Node noch antwortet. Jedoch kostet eine Ping-Anfrage Energie und verfälscht damit das Ergebnis. Ein Aufruf des Skriptes in Abständen von 20 Minuten erwies sich als günstig. Um die Node nicht unnötig zu belasten, bricht das Skript nach der ersten Ping-Antwort ab, aber spätestens nach drei unbeantworteten Ping-Anfragen.

```
1 #!/bin/bash
2
3 ip=2002:8d38:831a:aaaa:221:2eff:ff00:225d
4 max_tries=3; c=0; r=0;
5 until ping -c 1 $ip >/dev/null 2>&1 || ((++c >= max_tries)); do r=$?; done;
6
7 str_result="Node is down."
8 [ $r -eq 0 ] && str_result="Node is up."
9
10 echo `date +"%d.%m.%y %H:%M"`: $str_result
```

Listing 4: Skript zum Überprüfen der Laufzeit einer node

Geht man von einer CCR von 8 Hz und damit einem durchschnittlichen Verbrauch von 0,33228 mA aus (siehe Tabelle 3.1), ergibt sich eine theoretische Betriebsdauer von $\frac{0,86111mAh}{0,33228mA} = 2,59152$ Stunden. Die tatsächliche Betriebsdauer betrug jedoch nur 1,5 Stunden. In der Rechnung wurde allerdings nicht der Eigenverbrauch der Komponenten des Boards deRFnode berücksichtigt. Bei Messungen des Stromverbrauchs des gesamten Systems bei gleicher Konfiguration ergab sich ein durchschnittlicher Verbrauch von 0,40308 mA (vgl. [6]). Führt man die Rechnung mit diesem Wert erneut durch, ergibt sich eine Betriebsdauer von $\frac{0,86111mAh}{0,40308mA} = 2.13633$ Stunden. Auch mit dieser Rechnung betrug die tatsächliche Betriebsdauer nur etwa 70 Prozent der theoretischen Betriebsdauer. Im Umkehrschluss ergibt sich aus der tatsächlichen Laufzeit ein Verbrauch von $\frac{0,86111mAh}{1,5h} = 0,57407mA$.
Die Experimente wurden mit unterschiedlichen CCR-Werten durchgeführt und mehrmals wiederholt. Im Ergebnis betrug die tatsächliche Betriebszeit stets nur 60 bis 70 Prozent vom theoretischen Wert. Die Gründe für diese Abweichung sind ungeklärt. Möglicherweise eignen sich Kondensatoren nicht, um die Betriebszeit hinreichend genau zu bestimmen.

Channel-Check-Rate	**2 Hz**		**4 Hz**		**8 Hz**		**16 Hz**	
Messdauer	**Durch-schnitt**	**Median**	**Durch-schnitt**	**Median**	**Durch-schnitt**	**Median**	**Durch-schnitt**	**Median**
32 Sekunden (100 ms/div)	0,10238	0,10232	0,28781	0,16945	0,35414	0,307	0,83825	0,59282
64 Sekunden (200 ms/div)	0,10918	0,10214	0,21514	0,169	0,30625	0,30624	0,59853	0,58725
160 Sekunden (500 ms/div)	0,14138	0,10494	0,24216	0,16825	0,32506	0,30639	0,58708	0,58694
320 Sekunden (1 s/div)	0,29723	0,10104	0,21237	0,16883	0,34782	0,30685	0,59788	0,58938
640 Sekunden (2 s/div)	0,18452	0,10053	0,33208	0,16954	0,33511	0,30596	0,60965	0,59047
26,67 Minuten (5 s/div)	0,17911	0,10401	0,25868	0,20579	0,32877	0,33375	0,60283	0,59685
53,33 Minuten (10 s/div)	0,17096	0,15910	0,25511	0,19844	0,32882	0,31911	0,60283	0,59685
Durchschnitt über Durchschnitt/Median	0,16925	0,11058	0,25762	0,17847	0,33228	0,31218	0,63386	0,59151
Betriebsdauer mit 1900 mAh Akku	467,74 Tage	715,90 Tage	307,30 Tage	443,58 Tage	238,25 Tage	253,59 Tage	124,89 Tage	133,84 Tage

Tabelle 3.1: Ergebnisse der Messungen des Stromverbrauchs einer Node in mA vom Sommersemester 2016

3.1.2 Weitere Messergebnisse bezüglich des Stromverbrauchs

Ältere Ergebnisse

Tabelle 3.2 fasst die Messergebnisse vorangegangener Semester zusammen. Sofern nicht anders angegeben, stammen die Werte aus dem Zwischenbericht des Sommersemesters 2016 (siehe [11]) und beziehen sich auf Contiki 3.0 der HTW Dresden. Wenn nicht anders angegeben, wurden die Messungen an einem Shunt-Widerstand, welcher anstelle des Jumpers JP5 am Board deRFnode eingesetzt wurde (siehe auch [11, S. 23 ff.]). Bei den Messungen des Stromverbrauchs des gesamten Boards, wurde am Batteriefach des Boards deRFnode gemessen. Der Shunt-Widerstand wurde dabei zwischen Spannungsquelle und dem Batteriefach eingesetzt. Als Spannungsquelle wurde ein Netzteil statt Batterien verwendet, welches auf vier Volt eingestellt wurde (siehe [11, S. 27]).

	CPU-Takt	**CCR**	**Ergebnis**
Analytisch bestimmter Stromverbrauch	16 MHz	8 Hz	0,28131 mA
Analytisch in vorrangegangenen[a] Semestern bestimmter Stromverbrauch (Contiki 2.7)[b]	16 MHz	8 Hz	0,40306 mA
Analytisch in vorrangegangenen Semestern bestimmter Stromverbrauch (Contiki 2.7)[c]	8 MHz	8 Hz	0.17108 mA
Analytisch in vorrangegangenen Semestern bestimmter Stromverbrauch (Contiki 2.7)[c]	16 MHz	8 Hz	0.13979 mA
Funkmodul (Contiki 2.7)	16 MHz	8 Hz	0,24398 mA
Funkmodul	8 MHz	8 Hz	1,02062 mA
Funkmodul	16 MHz	8 Hz	0,99271 mA
gesamtes Board mit aktivierten Diagnose-LEDs	8 MHz	8 Hz	1,05693 mA
gesamtes Board mit aktivierten Diagnose-LEDs	16 MHz	8 Hz	1,02407 mA
gesamtes Board ohne Diagnose-LEDs	8 MHz	8 Hz	1,00676 mA
gesamtes Board ohne Diagnose-LEDs	16 MHz	8 Hz	1,11443 mA

[a] Gemeint sind die Semester, bevor der Autor dem Forschungsseminar beigetreten ist.
[b] Quelle: Sensornetze-Wiki (siehe [7]). Der Wiki-Artikel wurde zuletzt am 22.01.2015 bearbeitet.
[c] Quelle: Bericht von Adreas Salwasser vom 29.06.2015 (siehe [12]). Obige Werte wurden gewonnen, indem der monatliche Stromverbrauch aus [12] durch 24*30 dividiert wurde.

Tabelle 3.2: Ältere Ergebnisse bezüglich des Stromverbrauchs einer Node

Ergebnisse aus weiteren Experimenten

In Tabelle 3.2 auf der nächsten Seite sind weitere Ergebnisse mit Contiki 3.0 zusammengefasst (vgl. [9]). Dabei handelt es sich um Messergebnisse, welche bei unterschiedlichen Konfigurationen für den CPU-Takt, der CCR usw., durchgeführt wurden.

	CPU-Takt	CCR	Ergebnis
gesamtes Board mit aktivierten Diagnose-LEDs	16 MHz	8 Hz	0,4031 mA
gesamtes Board mit aktivierten Diagnose-LEDs ohne Schlafmodus[a]	8 MHz	8 Hz	11,6368 mA
gesamtes Board mit aktivierten Diagnose-LEDs ohne Schlafmodus	16 MHz	8 Hz	13,6313 mA
gesamtes Board mit aktivierten Diagnose-LEDs	8 MHz	∞[b]	22,7773 mA
gesamtes Board mit aktivierten Diagnose-LEDs	16 MHz	∞	24,1887 mA

[a] D. h. zwischen den CCA-Phasen wird lediglich die Funkeinheit deaktiviert und der Mikrocontroller wechselt niemals in den Schalfmodus (#define RDC_CONF_MCU_SLEEP 0).

[b] Die Funkeinheit war immer aktiv (Anweisung NETSTACK_MAC.off(1); in project/er-rest-example/er-example-server.c).

Tabelle 3.3: Weitere Ergebnisse bezüglich des Stromverbrauchs einer Node

3.1.3 Berechnung des Stromverbrauchs

Um den Stromverbrauch zu berechnen, genügt es, den mittleren Stromverbrauch eines Zyklus zu berechnen. Ein Zyklus besteht aus einer CCA-Phase[8], gefolgt von einer Ruhe-Phase (siehe Abb. 3.3). Die Ruhe-Phase tritt nur ein, wenn keine Daten empfangen oder gesendet werden. Es wird also der Idealfall berechnet. Weiterhin wird angenommen, dass der Mikrocontroller während der Ruhephase den Deep-Sleep-Modus nutzt, in welchem er weniger als 250 nA an Strom benötigt (vgl. [1, S. 1]).

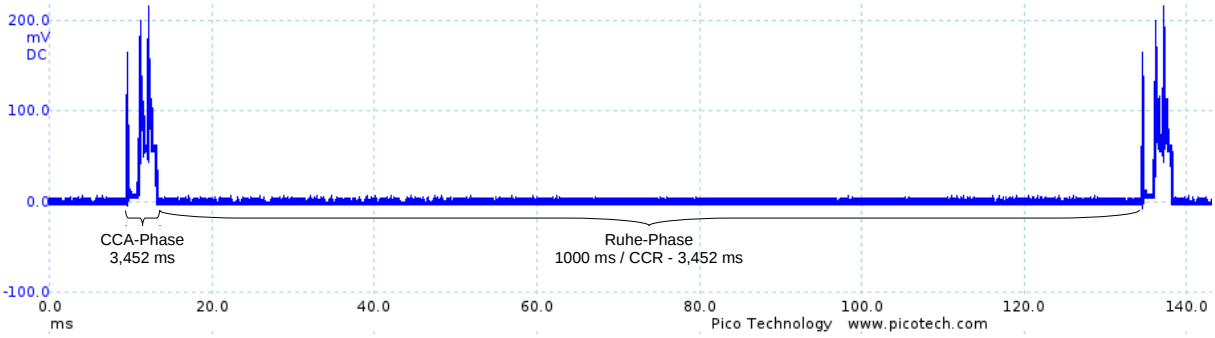

Abb. 3.3: Ein Zyklus (hier dargestellt für CCR = 8 Hz) besteht aus CCA- und Ruhe-Phase
Quelle: Eigene Darstellung

Die CCR legt fest, wie oft sich ein Zyklus in einer Sekunde wiederholt. Bei dem Standardwert von 8 Hz, wiederholt sich ein Zyklus also acht mal in der Sekunde. Ein Einzelzyklus dauert somit 125 ms. Die Dauer einer CCA-Phase beträgt, unabhängig von dem Wert der CCR, stets 3,452 ms (siehe Abb. 3.4). Die Ruhe-Phase nimt die Restliche Zeit des Zyklus ein. Bei einer CCR von 8 Hz beträgt die Dauer der Ruhe-Phase also: $\frac{1000ms}{8Hz} - 3,452ms = 121.548ms$ Der mittlere Stromverbrauch einer CCA-Phase wurde in [11] auf S. 25 bereits analytisch, mit dem Ergebnis 10,17758mA mA, berechnet. Dabei

[8] Genau genommen handelt es sich um zwei kurz aufeinanderfolgende CCA-Phasen (vgl. [2, S. 3]). Der Einfachheit halber wird hier der komplette Zeitraum, vom Aufwachen des Mikrocontrollers bis zur Ruhe-Phase, als eine CCA-Phase bezeichnet.

wurde die CCA-Phase jedoch zur Vereinfachung in wenige Abschnitte unterteilt, für welche jeweils ein konstanter Strom angenommen wurde (siehe Abb. 3.4). Daher handelt es sich bei dieser Berechnung um eine eher grobe Annäherung.

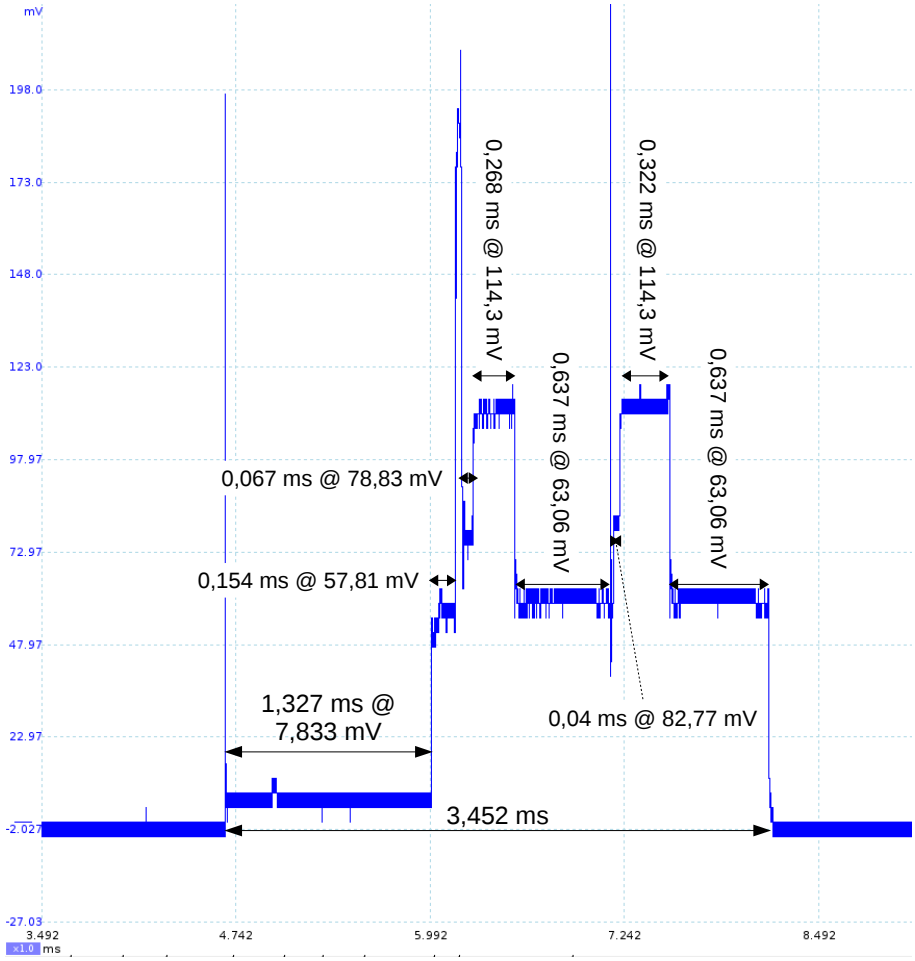

Abb. 3.4: Eine CCA-Phase mit dem Oszilloskop betrachtet
Quelle: [11, S. 26]

Exakter dürfte sich der Strom mit der numerischen Methode berechnen lassen. Dazu wurde mit dem Oszilloskop der Idealfall „eingefangen“. Schon bei einer Messung über einen Zeitraum von wenigen Sekunden, hat man mehrere Zyklen aufgezeichnet. Mit der numerischen Methode lässt sich der mittlere Stromverbrauch aller aufgezeichneten Zyklen berechnen. Tabelle 3.4 zeigt die Ergebnisse für den Idealfall bei einem CPU-Takt von 16 MHz und unterschiedlichen CCR-Werten.

	mittlerer Stromverbrauch	
CCR	**eines Zyklus**	**einer CCA-Phase**
8 Hz	0,30317 mA	10,96925 mA
4 Hz	0,16556 mA	11.9723 mA
2 Hz	0,09858 mA	14.24272 mA

Tabelle 3.4: Numerisch berechneter Stromverbrauch für den Idealfall bei unterschiedlichen CCR-Werten

Der mittlere Stromverbrauch x eines Zyklus lässt sich berechnen über:

$x = \frac{c \cdot 3{,}452ms + 0{,}00025mA \cdot (l - 3{,}452ms)}{l}$, mit c mittlerer Stromverbrauch einer CCA-Phase in mA und l Länge eines Zyklus in ms. Somit kann der Stromverbrauch c einer CCA-Phase aus den Werten x eines Zyklus in Tabelle 3.4 berechnet werden: $c = \frac{x \cdot l - 0{,}00025mA \cdot (l - 3{,}452ms)}{3{,}452ms}$
Für den mittleren Stromverbrauch einer CCA-Phase bei einer CCR von 8 Hz ergibt sich somit: $\frac{0{,}30317 \cdot 125ms - 0{,}00025mA \cdot (125 - 3{,}452ms)}{3{,}452ms} = 10,96925mA$ Die Berechnungen wurden ebenfalls für eine CCR von 4 und 2 Hz durchgeführt (siehe Tabelle 3.4). Theoretisch müssten die Werte für eine CCA-Phase für alle CCRs sehr nahe beieinander liegen. Dies ist nicht der Fall, was möglicherweise auf Messfehler zurückzuführen ist. Wenn als mittlerer Stromverbrauch für einen CCA-Zyklus der Mittelwert aus den Ergebnissen für eine CCR von 4 und 8 Hz gebildet wird (der Wert bei 2 Hz wird als Ausreißer gewertet und ignoriert), ergibt sich für eine CCA-Phase 11,47078 mA als mittlerer Stromverbrauch. Damit kann nun der theoretische Stromverbrauch für beliebige CCR-Werten für den Idealfall berechnet werden.

Bsp. für CCR=1 Hz:
$\frac{c \cdot 3{,}452ms + 0{,}00025mA \cdot (l - 3{,}452ms)}{l} = \frac{11{,}47078mA \cdot 3{,}452ms + 0{,}00025mA \cdot (1000ms - 3{,}452ms)}{1000ms} = 0.03985mA$

Bsp. für CCR=16 Hz:
$\frac{11{,}47078mA \cdot 3{,}452ms + 0{,}00025mA \cdot (62{,}5ms - 3{,}452ms)}{62{,}5ms} = 0.63379mA$

Welcher Wert für die CCR wäre nötig, damit eine Node mit den im Forschungsseminar gebräuchlichen 1900 mAh-Akkus 10 Jahre lang betrieben werden kann?

Zunächst wird berechnet, wie hoch der durchschnittliche Stromverbrauch sein darf:
$\frac{1900mAh}{24h \cdot 365 \cdot 10} = 0,02168mA$ Aus den Werten aus Tabelle 3.4 und den obigen Berechnungen kann man erkennen, dass sich der Stromverbrauch bei Halbierung der CCR ebenfalls ungefähr halbiert. In der Theorie würde es also schon ausreichen, die CCR von 1 Hz nochmals zu halbieren: $\frac{11{,}47078mA \cdot 3{,}452ms + 0{,}00025mA \cdot (2000ms - 3{,}452ms)}{2000ms} = 0.01955mA$ Daraus ergibt sich wiederum eine theoretische Betriebsdauer von $\frac{1900mAh}{0.01955mA} = 97186,70077h = 10$ Jahre + 16,64358 Tage.

Der oben berechnete Idealfall tritt leider niemals für einen längeren Zeitraum ein. In der Realität findet häufig Datenverkehr statt. Selbst, wenn eine Node nur sehr sporadisch Sensor-Werte überträgt, bzw. im Falle eines Aktors nur selten Anweisungen empfägt, bleibt noch die regelmäßige Kommunikation zur Selbstorganisation des Netzwerkes (siehe Kap. 3.2). Abb. 3.5 zeigt einen Zyklus, bei dem eine einzelne Ping-Anfrage auf eine Node abgesetzt wurde. Die rote Linie markiert einen Strom von $\frac{58{,}56mV}{5\Omega} = 11.712mA$. Gemessen wurde am Jumper JP5 des Boards deRFnode mit einem 5Ω-Shunt-Widerstand.
Ein Nachteil der Implementierung von ContikiMAC ist, dass der Mikrocontroller nur ganze Zyklen im Schlafmodus verweilen kann. So benötigte die Node bei der Messung in

Abb. 3.5 von Begin des Zyklus an bis zur Beantwortung der Ping-Anfrage nur etwa 24 ms. Der Mikrocontroller könnte danach bereits wieder in den Schlafmodus über gehen. Stattdessen muss die nächste CCA-Phase abgewartet werden. Wegen der längeren Wach-Phasen des Mikrocontrollers kann der Stromverbrauch daher in der Praxis bei niedrigeren CCR-Werten deutlich höher ausfallen, als bei nierigeren CCR-Werten.

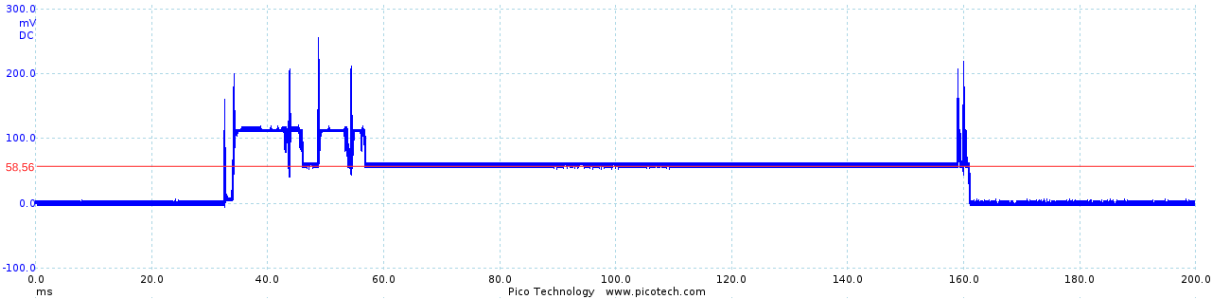

Abb. 3.5: Ein Zyklus während des Empfangens und Beantwortens einer Ping-Anfrage
Quelle: Eigene Darstellung

3.2 Analyse des Datenverkehrs mittels eines Sniffers

Während der Messungen des Stromverbrauchs wurde festgestellt, dass offenbar relativ regelmäßig Datenübertragung stattfindet. Um dies näher zu untersuchen, wurde der Datenverkehr mittels eines Sniffers analysiert. Der Sniffer basierte auf dem Sensor Terminal Board (stb231) mit dem Funkmodul RCB 231 (rdk230) von dresden elektronik und der Firmware Uracoli 0.4.2 (vgl [5]). Zur Analyse wurde das Programm Wireshark eingesetzt. Bei einem Versuchsaufbau mit drei Nodes und einem Border-Router, welche sich untereinander alle in Reichweite befanden, wurde festgestellt, dass alle Nodes in regelmäßigen Abständen Daten an alle anderen Geräte im Netzwerk und an die Multicast-Adresse ff02::1a senden (siehe Abb. 3.6). Der Border-Router sendet ausschließlich an diese Multicast-Adresse.

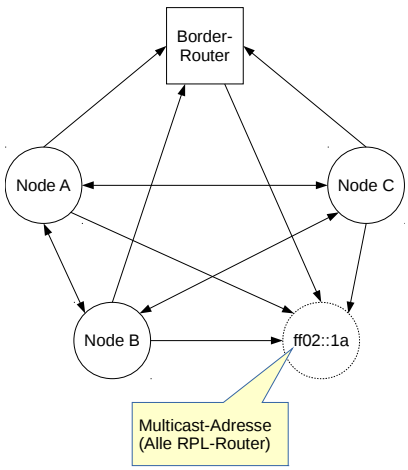

Abb. 3.6: Analyse des Datenverkehrs mittels eines Sniffers.
Quelle: Eigene Darstellung

Bei dem Datenverkehr handelte es sich um RPL- und Acknowledge-Datenpakete[9]. Aufgrund der Funktionsweise von ContikiMAC war zu beobachten, dass RPL-Pakete, welche an eine Node adressiert waren, meist mehr als einmal gesendet wurden, bis schließlich ein Acknowledge von der Ziel-Node gesendet wurde. Auf Pakete an die Multicast-Adresse ff02::1a folgten keine Acknowledges. An diese Adresse gesendete Pakete wurden stets 31 mal gesendet, scheinbar, um die Wahrscheinlichkeit zu erhöhen, dass alle Empfänger der Multicast-Adresse das Paket erhalten haben. Bei den gesendeten RPL-Paketen handelte es sich um *DODAG Information Objects*[10] und um *Destination Advertisement Objects*[11].

In welchen Abständen RPL-Pakete versendet werden, ließ sich nicht eindeutig feststellen, da die Abstände sehr unregelmäßig waren. Die Gesamtzahl der mitgeschnittenen Pakete betrug in 20 Minuten 546 Pakete. In einem weiteren 30-minütigen Sniffer-Lauf wurden 471 Pakete mitgeschnitten. Da jede Node RPL-Pakete an alle anderen Nodes (die sich in Reichweite befinden) sendet, wächst das Kommunikationsaufkommen von RPL quadratisch mit steigender Anzahl der Nodes.

Eine Node mit Contiki 2.7 sendet RPL-Pakete in ca. 20-minütigen Abständen ausschließlich an die Multicast-Adresse ff02::1a.

Ein weiteres Expermiment sollte zeigen, ob die Abstände, in welchen eine Node RPL-Pakete versendet, anders ausfallen, wenn die Funkeinheit der Node immer aktiv ist und der Mikrocontroller damit nicht in den Schlafmodus wechselt. Dies wurde mit Hilfe des in Kap. 2.4 beschriebenen Features bewerkstelligt. Es war kein Unterschied in der Frequenz der versendeten RPL-Pakete feststellbar.

3.3 Aufbau eines Testnetzes

Im Wintersemester 16/17 sollte ein stabiles Testnetzwerk aufgebaut werden, welches auch in der Lage ist, den Datenverkehr über andere Nodes zu routen, falls Sender und Empfänger außer Reichweite liegen. Bereits am Ende des Sommersemesters 2016 waren stabile Testnetzwerke möglich, mit zunächst einfacher Struktur wie in Abb. 3.7 auf der nächsten Seite gezeigt, in welcher alle Geräte in gegenseitiger Reichweite zueinander liegen.
Aufgrund der Hardware, welche nur in beschränkter Anzahl zur Verfügung stand und aus Platzgründen[12] fällt das Testnetzwerk im Wintersemester 16/17 nur wenig komplexer aus. Dem Testnetzwerk wurde eine weitere Node hinzugefügt, welche außer Reichweite des

[9] Auf jedes erfolgreich empfangene RPL-Paket folgt ein Acknowledge-Paket, es sei denn, der Empfänger ist die Multicast-Adresse ff02::1a.

[10] Jede Node teilt darüber periodisch ihren Status, mit Informationen wie ihrem Rang, die DODAG-ID usw., mit.

[11] Enthalten Informationen darüber, welche Nodes über diese Node (wenn sie als Router fungiert) erreichbar sind. Fungiert die Node nicht als Router, teilt sie lediglich ihre eigene Anwesenheit mit.

[12] Für ein größeres Testnetzwerk wird eine größere Fläche benötigt, auf welcher die Installation für eine längere Zeit unbewacht aufgebaut bleiben kann.

Border-Routers lag. Das Testnetzwerk hatte damit die Struktur wie in Abb. 3.8 gezeigt.

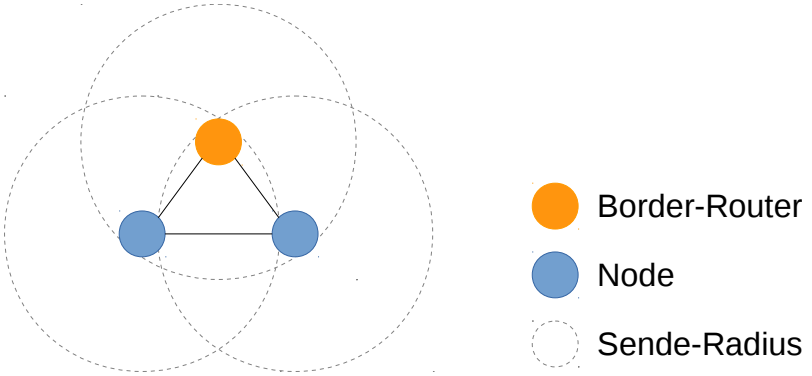

Abb. 3.7: Einfaches Testnetz aus dem Sommersemester 2016
Quelle: Eigene Darstellung

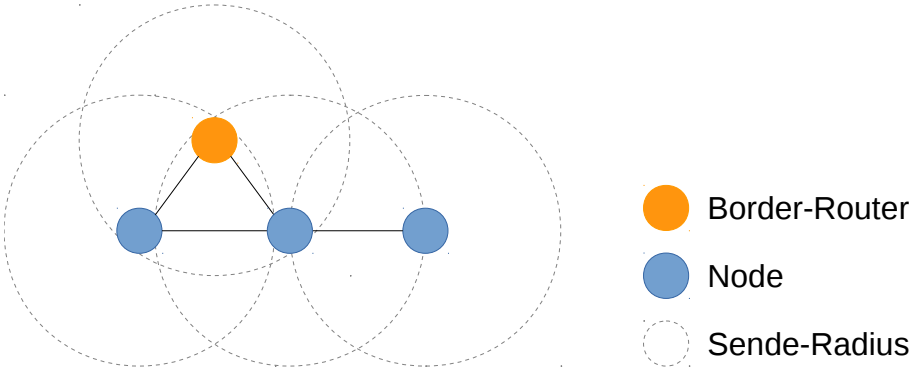

Abb. 3.8: Komplexeres Testnetz aus dem Wintersemester 16/17
Quelle: Eigene Darstellung

Zum Testen der Verbindungsqualität wurden von einem, am Border-Router angeschlossenen Computer, Ping-Tests und CoAP-Anfragen ausgeführt. Der konkrete Testaufbau ist in Abb. 3.9 auf der nächsten Seite zu sehen.
Die Nodes und Border-Router, welche durch eine schwarze oder rote Linie miteinander verbunden sind, befinden sich in gegenseitiger Reichweite zueinander. Die Ziel-Node befand sich außerhalb des Hauses in einem Briefkasten. Erschwerend kam hinzu, dass dieser aus Metall bestand. Die rote Linie beschreibt den Weg, welche die Daten nehmen, wenn Kommunikation zwischen dem Border-Router und der Ziel-Node stattfindet. Dies ist somit auch der Weg, den die Daten bei Ping- und CoAP-Anfragen zur Ziel-Node gegangen sind. Bei der Kommunikation mussten mehrere Wände überwunden werden. Border-Router und Router-Node befanden sich in einem Abstand von ca. 14 Metern (Luftlinie) zueinander. Für die Node, welche als Router fungierte, wurde eine stationäre Stromversorgung verwendet. Da Energie sparen für diese Node also keine Rolle spielte, blieb die Funkeinheit sändig aktiv (Nutzung des Features aus Kap. 2.4). Es ist auch ein praxisnahes Szenario, dass Router über eine stationäre Stromversorgung verfügen. (Die Ziel-Node wurde im energiesparenden Modus mit ContikiMAC betrieben.) Die Verbindungsqualität wurde für gut befunden (bei Ping-Anfragen blieben weitaus weniger als zehn Prozent der Ping-

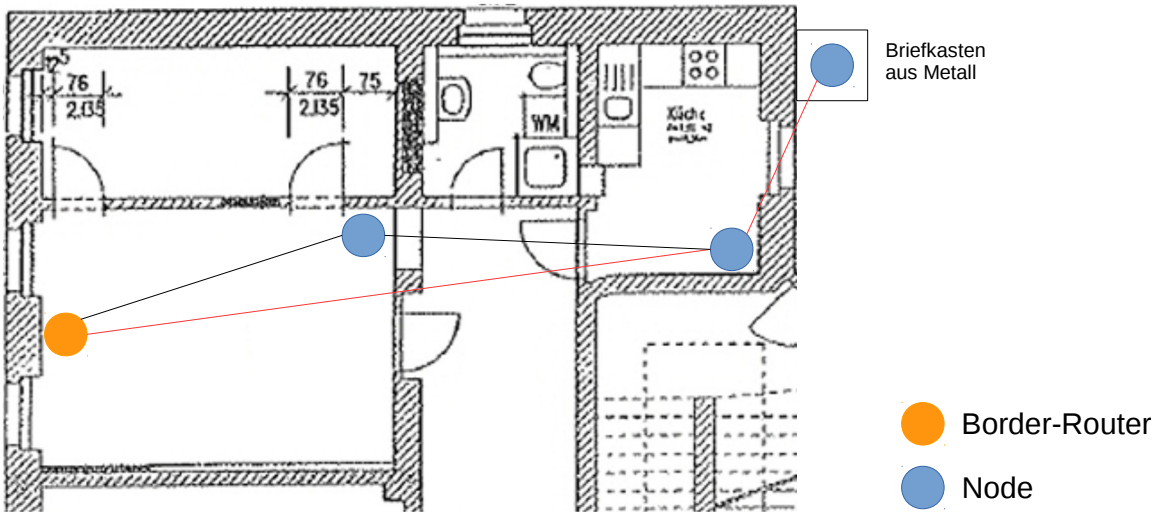

Abb. 3.9: Aufbau des Testnetzes aus dem Wintersemester 16/17
Quelle: Eigene Darstellung

Antworten aus). Die Verbindungsqualität war etwas besser, wenn die Node, welche als Router für die Ziel-Node fungierte, ihre Funkeinheit immer aktiv ließ. Die Stabilität des Testnetzwerks wurde über einen Zeitraum von einer Woche getestet, indem alle fünfzehn Minuten der Wert des externen Temperatursensors (siehe Kap. 2.3.2) via CoAP abgefragt wurde. Nur sechs Prozent der CoAP-Abfragen blieben erfolglos. Damit kann das Testnetzwerk als stabil angesehen werden.

Abschließend ist hinzuzufügen, dass Nodes ohne weitere Anpassungen von Contiki als Router fungieren. Dies kann unterbunden werden, indem die Anweisungen aus Listing 5 in die Datei *project/er-rest-example/project-conf.h* eingefügt werden.

```
#undef UIP_CONF_ROUTER
#define UIP_CONF_ROUTER 0
```

Listing 5: Zu ergänzende Anweisungen, dass Nodes nicht mehr als Router fungieren.

Dass eine Node als Router für eine andere fungiert, lässt sich über die Auswertung des internen Webservers des Border-Routers herausfinden. Listing 6 zeigt ein Beispiel, inwelchem zu sehen ist, dass die Node mit der IP-Adresse fe80::221:2eff:ff00:225d als Router für die Node mit der IP-Adresse fe80::21:2eff:ff00:3864 dient.

```
– 51 ––:––:–– ETANeighbors
fe80::221:2eff:ff00:225d
fe80::21:2eff:ff00:389a
fe80::21:2eff:ff00:3864
Routes
2002:8d38:831a:aaaa:221:2eff:ff00:225d/128 (via fe80::221:2eff:ff00:225d) 1436s
2002:8d38:831a:aaaa:21:2eff:ff00:3864/128 (via fe80::221:2eff:ff00:225d) 1121s
2002:8d38:831a:aaaa:21:2eff:ff00:389a/128 (via fe80::21:2eff:ff00:389a) 977s
```

Listing 6: Inhalt des internen Webservers eines Border-Routers

4 Bekannte Probleme

4.1 Keine CoAP-Subscription bei vielen Sensoren möglich

Subscriptions sind eine elegante Art, sich über geänderte Sensor-Werte informieren zu lassen, anstatt eine Node immer wieder aktiv anzufragen (Polling). Die Implementierung der Subscriptions wurde für einige CoAP-Ressourcen wegen eines ungeklärten Fehlers wieder rückgängig gemacht. Eine Subscription zu den betroffenen Ressourcen führte dazu, dass eine Node nach kurzer Zeit nicht mehr ansprechbar war. Es half dann nur ein Reset der Node. Betroffen sind die CoAP-Ressourcen des externen Temperatur- und Lichtsensors. Die Subscription des externen Beschleunigungssensors ist zwar noch implementiert und die Benutzung führt auch nicht zum oben beschriebenen Problem, jedoch stimmen hier die gelieferten Werte nicht. Allen dieser Sensoren ist gemein, dass sie über die I^2C-Schnittstelle angebunden sind (siehe [14]).

4.2 Externer Temperatursensor liefert falsche Werte für Temperaturen unter 0 °C

Die Werte, welcher der externer Temperatursensor (siehe Kap. 2.3.2) liefert, sind für Temperaturen unter 0 °C falsch. So wird knapp unter dem Nullpunkt eine Temperatur von ca. -128 °C zurückgegeben.

4.3 Häufiger RPL-Datenverkehr

Im Sensornetzwerk findet reger Routing Protocol for Low power and Lossy Networks (RPL)-Datenverkehr statt (siehe Kap. 3.2). Nicht nur werden RPL-Pakete in sehr kurzen Abständen gesendet, sondern von seiten der Nodes scheint der Verkehr auch redundant zu sein, da Pakete nicht nur an die Multicast-Adresse ff02::1a, sondern auch an alle, sich in Reichweite befindenden, Nodes gesendet werden. Dieses Verhalten stellt ein Problem dar, da es sich negativ auf den Stromverbrauch auswirkt (vgl. Kap. 3.1).

4.4 Reboot-Schleife bei aktivierter Brown-Out-Detection

Die Brown-Out-Detection ist ein Feature des ATmega128RFA1. Es sorgt dafür, dass der Controller in einen Reset-Zustand versetzt wird, wenn die Betriebsspannung einen gewissen Schwellwert unterschritten hat (vgl. [1, S. 182]). Der Schwellwert kann über drei spezielle Bits (BODLEVEL 0 bis 2) im Extended-Fuse-Byte festgelegt werden (siehe [1, S. 515]). Diese Funktion soll verhindern, dass der Controller bei einer zu niedrigen Betriebsspannung zwar weiter läuft, aber unzuverlässig wird.

Wenn ContikiMAC benutzt wird und die Brown-Out-Detection aktiviert ist, wird alle paar Minuten ein Reset ausgelöst. Die Nodes führten also ca. alle ein bis fünf Minuten einen Re-

boot aus. Die Ursache für diesen Fehler ist ungeklärt. Da die Brown-Out-Detection jedoch einen gewissen Eigenverbraucht hat und das Board deRFnode ohnehin über einen eigenen *Supervisor* verfügt, welcher den Reset-Zustand auslöst, sobald die Betriebsspannung unter 2,4 Volt gesunken ist (vgl. [13, S. 6]), kann die Brown-Out-Detection als Behelfslösung deaktiviert werden. Das Extended-Fuse-Byte hat damit den Wert 0xFF. Insgesamt lauten die Werte der Extended-, High- und Low-Fuse-Bytes damit: E:0xFF, H:0x91 und L:0xD7.[13]

4.5 Keine Tunslip-Verbindung mit dem Border-Router möglich

Mit älteren Hardwareversionen des Funkmoduls deRFmega128-22A00 ist es unter Verwendung von Contiki 3.0 aus den GitHub-Quellen der HTW Dresden nicht möglich, eine serielle Tunslip-Verbindung zwischen Border-Router und einem Computer aufzubauen.[14] Da der Autor über kein älteres Dunkmodul verfügte, konnte das Problem nicht nachvollzogen werden.

[13] Der Zwischenbericht des Sommersemesters 2016 nennt einen anderen Wert für das Extended-Fuse-Byte. Offenbar handelt es sich dabei um einen Fehler im Bericht.

[14] Quelle: Herr Jens Paul von der HTW Dresden

5 Ausblick

Mit den richtigen Einstellungen für die CCR erreichen wir mit drei 1900 mAh Akkus im AA-Format bereits eine Betriebszeit von über zwei Jahre (vgl. Kap. 3.1). Wünschenswert währe eine Betriebsdauer von mindestens zehn Jahren. Man könnte die Kapazität der Akkus weiter erhöhen. Jedoch benötigen Akkus mit einer größeren Kapazität mehr Platz und sind zudem teurer. Ein besserer Ansatz ist es daher, die Nodes noch sparsamer zu machen. Z. B. könnte die Verwendung von Frequenz unter zwei Hz für die CCR untersucht werden. Außerdem wurde RPL als Verursacher von relativ viel Kommunikation ausgemacht (vgl. Kap. 3.2). Es könnte lohnenswert sein, RPL und dessen Implementierung in Contiki besser zu verstehen und herauszufinden, inwieweit sich das RPL-Kommunikationsverhalten von Contiki anpassen lässt.
Eine weitere Möglichkeit wäre *Energy Harvesting*, d. h. die Gewinnung von Energie aus der Umgebung über z. B. Solarzellen. Durch Energy Harvesting gewonnene Energie könnte die Akkus unterstützen oder sogar wieder aufladen. Allerdings ist diese Möglichkeit differenziert zu betrachten, da die Einsatzumgebungen von Sonsorknoten nicht immer Energy Harvesting im ausreichenden Umfang zulassen.

Bisherige Testnetzwerke bestanden nur aus wenigen Geräten und waren nur für eine sehr beschränkte Zeit im Einsatz. Es könnte ein Testnetzwerk, z. B. in den Räumlichkeiten der HTW Dresden, mit eventuell mehr Nodes, aufgebaut werden. Dadurch können noch mehr Erkenntnisse zur Stabilität, Betriebsdauer mit einer Akku-Ladung, usw. erlangt werden.

Einige der in Kap. 4 beschriebenen Fehler könnten beseitigt werden. Insbesondere trifft dies auf die Fehler aus Kap. 4.1 zu. Denn möchte man Sensorwerte erfassen, ist die Verwendung von Subscriptions weitaus eleganter, als das ständige aktive Anfragen der Node. Außerdem kann so noch mehr Energie gespart werden, da die Kommunikation noch geringer ausfällt. Dies hat zwei Gründe: Sensor-Werte werden nur dann übertragen, wenn sie sich geändert haben[15] und es entfällt der Kommunikationsaufwand für die regelmäßigen Anfragen der Werte (da diese nicht mehr notwendig sind, wenn die Node die Daten unaufgefordert sendet).

[15] Damit der Energiespareffekt durch eine Subscription nicht zunichte gemacht wird, sollte eine sinnvolle untere Zeitschranke konfiguriert werden, welche die Frequenz der Datenübermittlung begrenzt.

Literatur

[1] *ATmega128RFA1 Datasheet*. TJA1043. 8266F-MCU Wireless-09/14. Atmel Corporation. Sep. 2014.

[2] Adam Dunkels: *The ContikiMAC Radio Duty Cycling Protocol*. Report. 2011.

[3] Sensornetze Wiki der HTW Dresden: *Coap-client installieren*. URL: https://www2.htw-dresden.de/~wiki_sn/index.php/Coap-client_installieren (besucht am 24. 02. 2017).

[4] Sensornetze Wiki der HTW Dresden: *CoAP-Ressourcen unter Contiki 3.0*. URL: https://www2.htw-dresden.de/~wiki_sn/index.php/CoAP-Ressourcen_unter_Contiki_3.0 (besucht am 24. 02. 2017).

[5] Sensornetze Wiki der HTW Dresden: *Sniffer mit uracoli und dem Sensor Terminal Board mit RCB128RFA1 als Funk-Modul*. URL: https://www2.htw-dresden.de/~wiki_sn/index.php/Sniffer_mit_uracoli_und_dem_Sensor_Terminal_Board_mit_RCB128RFA1_als_Funk-Modul (besucht am 26. 02. 2017).

[6] Sensornetze Wiki der HTW Dresden: *Strombedarf mit Contiki 3 mit Stand vom 12. Nov. 2016 (Commit f8fdd14)*. URL: https://www2.htw-dresden.de/~wiki_sn/index.php/Stromverbrauch_mit_Contiki_3.0#Strombedarf_mit_Contiki_3_mit_Stand_vom_12._Nov._2016_.28Commit_f8fdd14.29_.28gesamtes_Board.29 (besucht am 28. 02. 2017).

[7] Sensornetze Wiki der HTW Dresden: *Strommessungen des Funkmoduls*. URL: https://www2.htw-dresden.de/~wiki_sn/index.php/Strommessungen_des_Funkmoduls (besucht am 16. 03. 2017).

[8] Sensornetze Wiki der HTW Dresden: *Stromverbrauch mit Contiki 2.7*. URL: https://www2.htw-dresden.de/~wiki_sn/index.php/Stromverbrauch_mit_Contiki_3.0#Stromverbrauch_mit_Contiki_2.7 (besucht am 23. 02. 2017).

[9] Sensornetze Wiki der HTW Dresden: *Stromverbrauch mit Contiki 3.0*. URL: https://www2.htw-dresden.de/~wiki_sn/index.php/Stromverbrauch_mit_Contiki_3.0 (besucht am 25. 02. 2017).

[10] Texas Instruments: *TMP102 Low-Power Digital Temperature Sensor With SMBus and Two-Wire Serial*. URL: http://www.ti.com.cn/cn/lit/ds/symlink/tmp102.pdf (besucht am 28. 02. 2017).

[11] Richter Kai und Gräb Alexander: *Zwischenbericht für das Forschungsseminar Sensornetze*. Report. 2016.

[12] Andreas Salwasser: *Forschungsbericht für das Fach Sensornetze*. URL: https://www2.htw-dresden.de/~wiki_sn/images/c/c0/SS15_Bericht_Andreas_Salwasser.pdf (besucht am 16. 03. 2017).

[13] *User Manuel deRFnode / deRFgateway*. V1.3. dresden elektronik. Apr. 2014.

[14] Wikipedia: *I^2C*. URL: https://de.wikipedia.org/wiki/I%C2%B2C (besucht am 25. 02. 2017).

[15] Wikipedia: *Representational state transfer*. URL: https://en.wikipedia.org/wiki/Representational_state_transfer (besucht am 24. 02. 2017).