Hochschule für Technik und Wirtschaft Dresden

Abschlussdokumentation Sensornetze

Stefan Lorenz, Thomas Kühnel
23. Februar 2014

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1. Einführung

Diese Dokumentation wurde zum Abschluss eines Forschungsseminars zum Thema Sensornetze verfasst. Die Dokumentation ist anhand der Aufgabenstellungen gegliedert. In den einzelnen Kapiteln werden die jeweiligen Aufgabenstellungen an die Autoren vorgestellt und abgearbeitet. Sie soll den nachfolgenden Teilnehmern des Forschungsseminars als mögliche Basis für die Fortführung dieser oder anderer Aufgaben dienen.

Das Grundgerüst für unser Sensornetz sind die Sensorknoten, bestehend aus einer AVR ATmega128RFA1 Mikrocontroller/RF Transceiver Kombination (siehe [Atmel1] für genauere Details), welche auf deRFbreakout Boards aufgesetzt sind.

Abbildung 1.0.1.: deRFbreakout Board

Abbildung 1.0.2.: deRFmega128 Board

Daran können diverse Sensoren, Aktoren und andere Geräte angeschlossen werden (beispielsweise Temperatur- / Feuchtigkeitssensoren, Buttons, LEDs, USB für externe Strom-

versorgung). Programmiert werden die Mikrocontroller mit dem Atmel AVR Dragon Board.

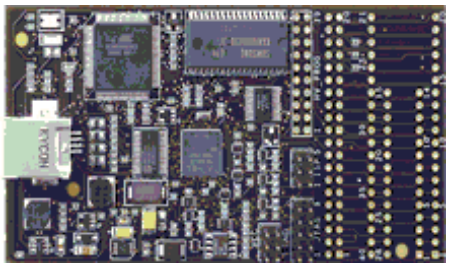

Abbildung 1.0.3.: Atmel AVR Dragon Board (In System Programmer)

Auf den Mikrocontrollern ist das Contiki-Betriebssystem aufgespielt. Es wurde von Adam Dunkels entworfen und wird von ihm und weiteren Personen in einer Open-Source-Community weiterentwickelt. Dieses Betriebssystem ist für 8-Bit Computer und integrierte Mikrocontroller gedacht. Es ist gut für das Internet der Dinge geeignet, da es IPv6 unterstützt. Dafür hat es u.a. einen TCP/IP-Netzwerkstack, den uIP-Stack (ehemals eigenständige Entwicklung von Adam Dunkels), integriert.

2. ContikiMAC

Im letzten Semester wurde untersucht, welche MAC-Implementierungen in Contiki vorhanden sind, und welche Stromsparmöglichkeiten diese bieten. Dabei stellte sich heraus, dass es mit ContikiMAC zwar ein sehr stromsparendes Verfahren gibt, dieses allerdings ein von den Contiki-Entwicklern entworfenes Protokoll ist. Damit ist keine Interopeerabilität mit anderen Systemen gegeben, da diese üblicherweise nicht das ContikiMAC Protokoll implementieren oder speziell darauf angepasst werden müssten. Alternativ dazu gibt es bereits im 802.15.4 Standard eine Methode, wie stromsparende Knoten betrieben werden sollen. Deshalb wäre es wünschenswert, diesen Modus auch in Contiki umzusetzen.

2.1. Standardkonformer Stromsparmodus

Um einen standardkonformen Stromsparmodus in Contiki zu ermöglichen bieten sich zwei verschiedene Möglichkeiten an:

1. Es wird eine fertige Implementierung des Protokolls verwendet und in Contiki integriert. Dafür wurden wiederum zwei Implementierungen betrachtet, welche allerdings beide nicht geeignet sind.

    a) Ist der MAC-Stack von Atmel, der für die Im Forschungsseminar verwendeten Mikrocontroller entwickelt wurde. Dieser Implementiert zwar die nötigen Features, allerdings wird er zwar kostenlos zur Verfügung gestellt, aber ist unter keiner freien Lizenz veröffentlicht. Damit könnte er zwar für den eigenen Bedarf in den Contiki-Code integriert werden, allerdings wäre es nicht möglich, das fertige Ergebnis in das offizielle Contiki-Repository zu integrieren und damit allen Nutzern zur Verfügung zu stellen.

    b) Alternativ wurde die 802.15.4 Implementierung für Linux betrachtet. In dieser fehlt allerdings auch noch der benötigte Stromsparmodus, weshalb sie auch nicht verwendbar ist. Zudem existiert auch hier ein Lizenzproblem, da die von Linux und von Contiki verwendeten Open Source Lizenzen nicht miteinander kompatibel sind, kann auch kein Linux Code in Contiki integriert werden.

2. Die Alternative dazu war eine eigene Erweiterung des Contiki-Codes, um ihn standardkonform zu machen. Diese Variante wurde nicht durchgeführt, da der nicht mehr genügend Zeit vorhanden war, um dies umzusetzen.

Deshalb wurde für den weiteren Verlauf den Forschungsseminars entschieden ContikiMAC zu verwenden und dieses dann gegebenenfalls später zu ersetzen. Die Entwicklung einen standardkonformen Modus wird in einer Masterthesis bearbeitet und kann dann in das System integriert werden.

2.2. Betrachtung des Stromverbrauchs

Im ersten Semester wurde der Stromverbrauch der Sensorknoten für die verschiedenen RDC-Verfahren in Contiki untersucht. Dabei wurde für ContikiMAC eine Batterielebensdauer von einem halben Jahr ermittelt. allerdings wurde auch beobachtet, dass die Messung des Stromverbrauchs im Deep Sleep Modus des Mikrocontrollers möglicherweise sehr ungenau ist, da er extrem gering und damit kaum messbar ist. Um bessere Ergebnisse zu erzielen wurde die Messung wiederholt, aber dieses Mal anders ausgewertet. Aus dem gemessenen Stromverbrauch wird ermittelt, wie lange das System aktiv ist und wie lange es schläft. Damit kann dann mit den im Datenblatt angegebenen Werten für den Stromverbrauch im Schlaf- und Wachzustand ermittelt werden, wie hoch der Stromverbrauch im insgesamt im Betrieb mit ContikiMAC ist.

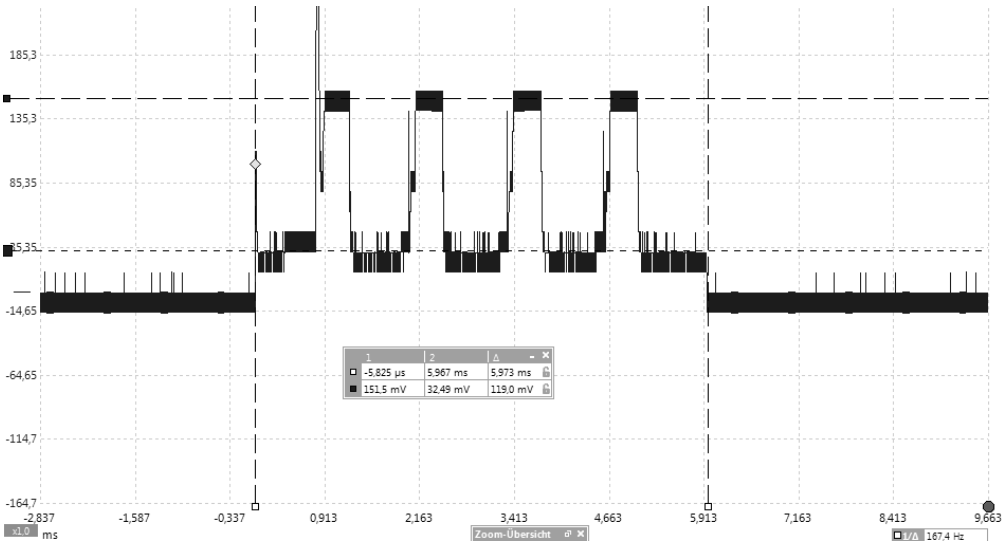

Abbildung 2.2.1.: Messung Stromverbrauch von ContikiMAC

Bei den Messungen wurde ContikiMAC mit einer Frequenz von 8 Hertz betrieben, das heißt, der Kanal wird 8 Mal pro Sekunde auf potentielle Sender überprüft. Dabei wurde beobachtet, dass der Mikrocontroller immer 3 Mal für 12 ms aktiv wird und in der 4. Periode dann für 18 ms. Damit ist das Board für $(6 * 12ms + 2 * 18ms)/1s = 3\%$ der Laufzeit nicht im Deep Sleep Modus. Während jeder Periode wird das Funkmodul mehrmals an und aus geschaltet, wie in Abbildung 2.2.1 zu sehen. Daraus ergibt sich, dass nur zu 2/6 der Zeit das Funkmodul im Listen-Modus ist und es für 4/6 der Zeit

deaktiviert ist. Damit lässt sich der Stromverbrauch während der ContikiMAC-Perioden wie folgt ermitteln:

$$2/6 * 16.6mA + 4/6 * 4.7mA = 8.666mA \tag{2.2.1}$$

Für das Senden eines Sensormesswertes über CoAP wurde eine Dauer von 9ms ermittelt, in dieser Zeit beträgt der Energieverbrauch 18.6mA. Im folgenden wird angenommen, dass jede Minute ein Messwert übertragen wird. Es ist also $9ms/60s = 0.015\%$ der Zeit der Sendemodus aktiv. Die restliche Zeit befindet sich der Mikrocontroller im Deep Sleep Modus mit einem Stromverbrauch von 250nA.

Mit diesen Ergebnissen lässt sich der durchschnittliche Stromverbrauch und damit die erwartete Batterielebensdauer ermitteln. Er lautet wie folgt:

$$(8.666mA)*0.03+18.6mA*1.5*10^{-4}+250nA*(1-0.03-1.5*10^{-4}) = 0.263mA \tag{2.2.2}$$

Daraus ergibt sich etwa für den Betrieb mit 2 AAA Batterien mit je 1000mAh eine Laufzeit von $2000mAh/0.263mA = 7605h$, also ungefähr 317 Tage. Dieses Ergebnis ist besser als das im letzten Semester ermittelte von etwa 0,5 Jahren und sollte auch näher am tatsächlichen Stromverbrauch liegen. Wird ContikiMAC durch den in 802.15.4 spezifizierten stromsparenden Modus ersetzt, sollte sich der Stromverbrauch weiter verringern, da damit längere Schlafphasen möglich sind.

3. Testnetz

Um die Komponenten der anderen Gruppen auch mit echter Hardware zu testen wurde ein kleines Testnetz mit Contiki-Knoten erstellt. In diesem befindet sich ein Border-Router, der das Sensornetz mit dem Server verbindet und damit überhaupt Zugang zum Sensornetz ermöglicht, einem Router, der Datenverkehr von Knoten weiterleitet, die nicht in direkter Reichweite des Border-Routers liegen und zwei Sensorknoten. Davon ist einer direkt mit dem Border-Router verbunden und einer indirekt über den Router. Der Aubau des Netzes ist in Abbildung 3.0.1 zu sehen. An dies Sensorknoten wurde jeweils ein SHT-21 Temperatur und Luftfeuchtesensor angeschlossen. Zusätzlich besitzt einer der Knoten zwei LEDs, die als Aktoren agieren.

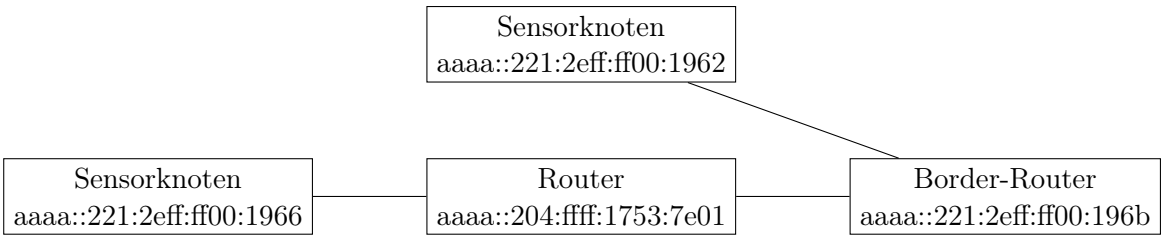

Abbildung 3.0.1.: Testnetz

3.1. CoAP-Server

Auf den beiden Sensorknoten läuft ein CoAP-Server, über den die Ressourcen erreichbar sind und ausgelesen bzw. geschaltet werden können. Die verfügbaren Ressourcen der jeweiligen Sensorknoten lassen sich über die DISCOVER-Funktion in CoAP ermitteln. Neben den Namen der Ressourcen teilt hier auch jeder Sensor mit, in welchem Format er seine Messwerte zurückliefern wird. Dafür wird die Resource-Type Eigenschaft verwendet. Diese erhält einen Wert der Form „rt=<Sensortyp>-<Einheit>“.

Derzeit sind die in Tabelle 3.1.1 enthaltenen Sensoren möglich. Jede dieser Ressourcen liefert den aktuellen Messwert zurück, wenn sie mittels GET abgefragt wird, weiterhin ist es möglich, die Sensoren mittels OBSERVE zu abonnieren, und sich damit periodisch den aktuellen Messwert zusenden zu lassen.

Ressource	Resource-Type	Beschreibung
sensors/sht21_temperature	Temperature-C	Temperatur in °C
sensors/sht21_humidity	Humidity-rel	Luftdruck in hPa
sensors/bmp085	Pressure-hPa	relative Luftfeuchtigkeit in %

Tabelle 3.1.1.: Sensorressourcen der Sensorknoten

Zusätzlich zu den Sensoren gibt es auch Aktoren, die derzeit zur Veranschaulichung LEDs am Sensorboard sind, aber auch durch andere Aktoren wie Lichtschalter oder Türöffner ersetzt werden könnten. Die folgenden Aktoren existieren derzeit:

actuators/leds?color=r|g|b Hiermit kann eine der drei vorhandenen LEDs geschaltet werden, die anzusprechende LED wird mit dem Wert des „color“-Parameter in der URL gewählt. Die Ressource lässt sich über POST oder PUT ansprechen. Als Content muss „mode=on|off“ übergeben werden. Wenn sich der Status der LED durch den Aufruf geändert hat, wird der Statuscode CHANGED zurückgeliefert.

actuators/toggle Diese Ressource lässt sich nur mit POST ansprechen und benötigt keine Parameter. ihr Aufruf sorgt dafür, dass der Status der roten LED geändert wird.

3.2. Anschluss der Peripherie

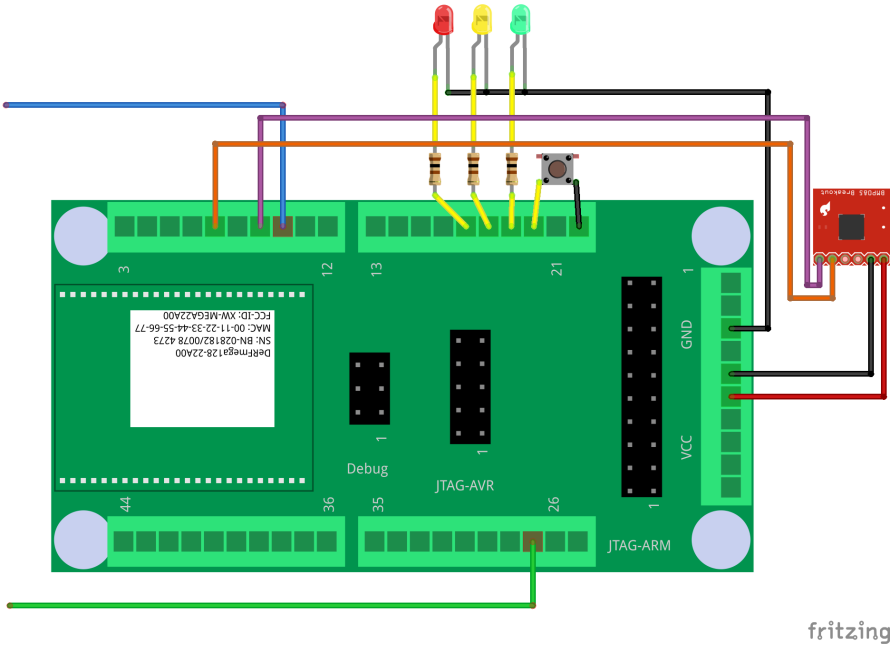

Abbildung 3.2.1.: deRFmega128 Platine mit bmp085 Sensor

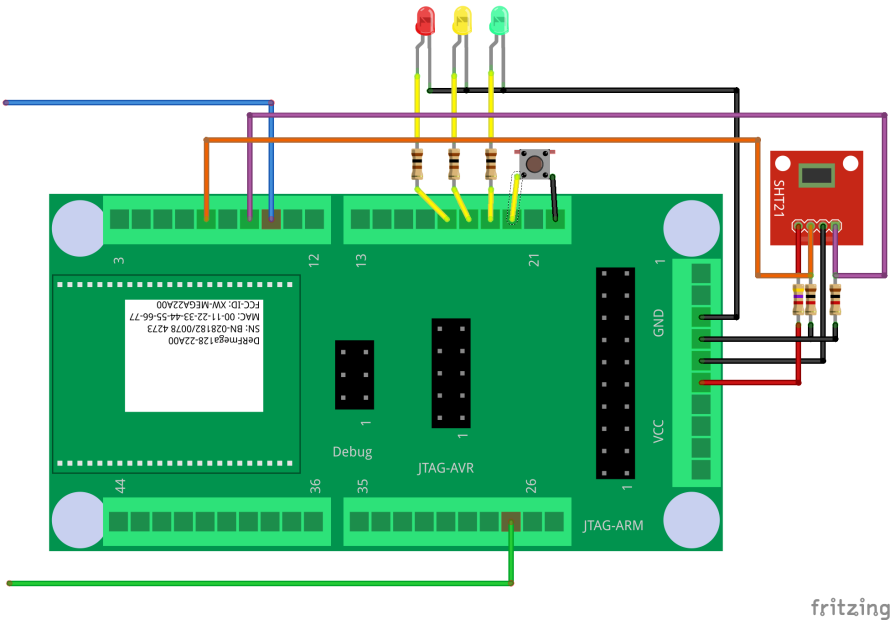

Abbildung 3.2.2.: deRFmega128 Platine mit sht-21 Sensor

Die Sensoren und Aktoren werden wie in den Abbildungen 3.2.1 und 3.2.2 angeschlossen. Dabei sind für den sht21 zusätzliche Widerstände notwendig (4,7 kΩ an VCC und 1kΩ an SDA und SCL). Für den Betrieb des bmp085-Sensors werden keine zusätzlichen Widerstände benötigt, da sich diese bereits auf dem Breakout Board befinden. Die LEDs wurde mit 100Ω Widerständen angeschlossen. Die Bedeutungen der Kabelfarben in den beiden Abbildungen wird in der folgenden Tabelle 3.2.1 erläutert.

Farbe	Bedeutung
schwarz	GND
rot	VCC
gelb	VCC LEDs, Button
blau	UART TXD
grün	UART RXD
orange	I^2C SCL
violett	I^2C SDA

Tabelle 3.2.1.: Zuordnung der Kabelfarben

3.3. ContikiMAC auf der Atmega128rfa1 Plattform

ContikiMAC [CMAC] ist das Radio Duty Cycle Verfahren, welches verwendet wird, bis der im vorherigen Kapitel erwähnte standardkonforme Stromsparmodus implementiert

ist. Dabei muss der Mikrocontroller nur in bestimmten Intervallen aktiv sein, um zu lauschen, ob ein anderer Knoten Nachrichten an ihn senden will. Der Sender schickt seine Nachrichten mehrfach, bis der Empfänger den Empfang bestätigt. Für spätere Sendeversuche kann gespeichert werden, wann der Empfänger empfangsbereit war, um beim nächsten Sendeversuch weniger Wiederholungen zu benötigen. Die restliche Zeit kann er sich im Stromsparmodus befinden. Dafür wird ein Timer verwendet, um möglichst genau diese Periode einzuhalten.

Anfangs wurde das Testnetz ohne ContikiMAC betrieben, um dies als mögliche Fehlerquelle bei der Kommunikation mit den Sensoren und der drahtlosen Datenübertragung auszuschließen, da dies im ersten Teil des Forschungsprojekts bereits einige Probleme bereitet hat. Um das Netz dann aber auch batteriebetrieben zu verwenden, sollte ContikiMAC wieder aktiviert werden.

Das Aktivieren von ContikiMAC sorgte allerdings dafür, dass teilweise beim Auslesen der Sensoren Fehler aufgetreten sind. Dies hat entweder dafür gesorgt, dass keine oder falsche Messergebnisse zurückgeliefert wurden, oder Contiki abgestürzt ist und der Sensorknoten bis zu einem manuellen Neustart nicht mehr ansprechbar war.

Der erste Lösungsversuch, war, ContikiMAC vor Auslesen der Sensoren zu deaktivieren und danach wieder zu starten. Dies löste das Problem mit den Fehlern beim Auslesen der Sensoren, aber sorgte dafür für Störungen im Netzwerkbetrieb. Grund dafür ist unter anderem, dass die Wachperioden damit nach jedem Neustart von ContikiMAC verschoben sind und die Synchronisation nicht mehr funktioniert. Allerdings konnte damit festgestellt werden, dass das problem beim Zusammenspiel von ContikiMAC und dem Zugriff auf die Sensoren lag.

Zur Lösung des Problem führte schließlich eine Diskussion auf der Contiki Mailingliste [ContikiML]. Dort veröffentlichte der Nutzer Johan Westö seine Lösung für dieses Problem. Die vorgeschlagene Änderung bewirkt, dass der Schlafzustand nur noch aktiviert werden kann, wenn alle Prozesse abgearbeitet sind. Damit ist es nicht mehr möglich, dass eine I^2C Übertragung gestört wird.

4. Pairing

Der Begriff Pairing stammt aus dem Bereich des Datenübertragungsstandards Bluetooth. Er bezeichnet die erstmalige Verbindungsaufnahme zweier Geräte, bei der Informationen ausgetauscht werden, um zukünftige Kommunikation zu ermöglichen bzw. zu vereinfachen.

Dabei können verschiedene Kommunikationswege und Abläufe angewandt werden. Zur Unterscheidung kann man feststellen welche Kommunikationswege zum Einsatz kommen oder welche Nutzeraktionen bzw. Phasen nötig sind. Findet die erstmalige Verbindungsaufnahme über einen anderen Weg statt, als die übliche Kommunikation, nennt man dies OOB-Kanal(Out-of-Bound/Out-of-Band). Ausschlaggebend für ein Pairingverfahren sind die Aktionen, die der Nutzer durchführen muss. Man kann diese in 3 Phasen unterteilen, wobei diese je nach Verfahren auch teilweise entfallen können. Der Nutzer ist entweder für die Einleitung bzw. den Abschluss des Verfahrens oder für den Informationsaustausch nötig. Ein gute Übersicht bietet die Tabelle im Anhang A.0.1. Diese Verfahren sind hauptsächlich für mobile Geräte gedacht. Trotzdem können einige der genannten Verfahren auch für stationäre Geräte verwendet werden.

4.1. Aufgabe

Im Forschungsseminar Sensornetze sollte ein Konzept sowie die letztendliche Implementierung für einen Pairingvorgang bereitgestellt werden, welches es ermöglicht einen neuen Sensorknoten mittels eines Pairingvorgangs in ein bestehendes Sensornetz aufzunehmen. Dafür sollte das bestehende Sensornetz der HTW Dresden verwendet werden, welches bereits in vorherigen Forschungsseminaren aufgebaut wurde. Im ersten Schritt galt es, sich mit der bestehenden Hardware und Software vertraut zu machen. Nachdem die Materie ausreichend bekannt war, mussten Mittel und Wege gefunden werden, wie ein Pairingvorgang umgesetzt werden kann.

Der IEEE 802.15.4 Standard, welcher die drahtlose Datenübertragung für Geräte mit geringer Leistungsaufnahme und niedriger Datenübertragungsrate definiert, sollte möglichst eingehalten werden. Aus diesem Grund musste ein Pairing-Ansatz gewählt werden, der keine kabelgebunde Erstkommunikation voraussetzt. Weiterhin soll der Pairingvorgang selbst sicher sein.

4.2. Ist-Analyse

Die nächsten 2 Unterpunkte beschäftigen sich mit dem Verfahren der Aufnahme eines neuen Knotens und der allgemeinen Sicherheit in Contiki-Sensornetzen. Daraus werden die Anforderungen abgeleitet, welche dann in die Konzeptionierung für einen sicheren Pairing-Vorgang einfließen.

4.2.1. Sicherheitsarchitektur des Contiki OS

Einen expliziten Zugriffsschutz, welcher ein Pairing überhaupt nötig machen würde, ist in der derzeitigen Version von Contiki nicht vorhanden. Derzeit wird jeder Sensorknoten, auf dem Contiki läuft, einfach in ein bestehendes Netz in Reichweite aufgenommen. Verantwortlich dafür ist das RPL-Protokoll, auf welches im nächsten Abschnitt eingegangen wird. Ein Pairingvorgang mit Nutzerinteraktion ist unter diesen Bedingungen nicht nötig. Für den allgemeinen Gebrauch von Sensornetzen ist es aber meist erwünscht, dass nicht jeder Knoten einfach in ein bestehendes Sensornetz aufgenommen wird.

4.2.2. Routing Protokoll - RPL

Im vorherigen Abschnitt wurde bereits gesagt, dass das RPL (Routing Protocoll for Low power and Lossy networks) für den barrierefreien Zugang eines neuen Netzknotens zu einem bestehenden Sensornetz verantwortlich ist. Dieser Abschnitt geht genauer darauf ein, wie RPL funktioniert und warum dadurch jeder Knoten ohne weiteres in ein Netzwerk aufgenommen wird.Die Netzwerke von denen in diesem Abschnitt gesprochen wird, nennt man LLNs (Low power and Lossy Networks). Das Protokoll ist distanzvektorbasiert. Es baut eine baumartige Toplogie, genauer gesagt einen zielorientierten, gerichteten, azyklischen Graphen (engl. DODAG - Destination Oriented Directed Acyclic Graph) auf, mit welchem das LLN gebildet wird. Diese Art von Netzwerk kann immer nur einen Zielknoten haben, nämlich die Wurzel des Baumes. (Vergleich [Ali])

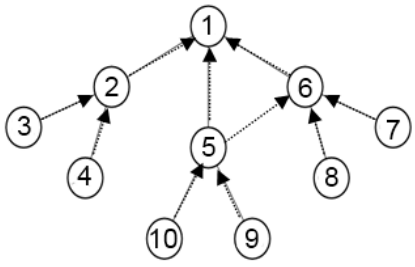

Abbildung 4.2.1.: RPL-Netzwerk-Topologie [Ali, S. 18]

Die Kommunikation findet dabei mittels IP-Broadcast (Multicast bei IPv6) oder den Link-Local Adressen der Teilnehmer statt. Die IP-Adressen der Teilnehmer setzen sich dabei aus der Link-Local Adresse (fe80::/64) und der Interface-ID zusammen.

Multicast-Adresse (speziell für RPL-Router)	ff02::1a
Link-Local + Interface ID(beliebiges Beispiel)	fe80::221:2eff:ff00:196a

Tabelle 4.2.1.: IPv6 Adresstypen für RPL

Beim Aufbau des LLN wird im wesentlichen auf 3 Nachrichtentypen zurückgegriffen. Jeder Knoten teilt seinen Status periodisch in einem DODAG Information Object mit (DIO). Hierin enthalten sind zum Beispiel Informationen wie: Knotenrang, DODAG-ID, RPL-Instanz-ID und Iteration der RPL-Instanz. Anhand der Informationen in den DIO Nachrichten, wird das LLN gebildet. Wenn die Knoten längere Zeit keine DIOs austauschen kann ein Knoten mittels eines DODAG Information Solicitation (DIS) explizit ein DIO anfordern.Nun ist jeder Knoten im Graphen in der Lage den Zielknoten zu erreichen, allerdings wissen die Elternknoten nichts von ihren untergeordneten Knoten. Damit es ermöglicht wird, dass die übergeordneten Knoten ihre untergeordneten Knoten erreichen können, schicken diese ein Destination Advertisement Object (DAO) an die übergeordneten Knoten. In diesem Nachrichtentyp wird lediglich mitgeteilt, welche Knoten über den Absender des DAO erreichbar sind. Es entsteht somit eine Art Routingtabelle. Diese wird immer weiter nach oben geleitet bzw. erweitert, bis der Zielknoten die DAOs seiner direkt untergeordneten Knoten bekommt. (Vergleich [Wiki1])

Der RPL-Standard unterstützt eigentlich 3 verschiedene Sicherheitsmodi: einen ungesicherten Modus, einen vorinstallierten Modus und einen authentisierenden Modus. Der erste stellt keine zusätzlichen Sicherheitsfeatures bereit. Die 2 anderen Modi stellen in der Theorie eigentlich genau das gewünschte Ergebnis her. Der Standard schreibt im vorinstallierten Modus einen nicht näher beschriebenen Schlüssel vor, der auf dem Netzknoten vorinstalliert ist. Mit diesem wird Zugang zum Netz gewährt und die RPL-Nachrichten können verschlüsselt werden. Der authentisierende Modus schränkt den vorher beschriebenen Modus ein, indem er neue Netzknoten nur als “Blätter” in der Topologie zulässt. Um als Border-Router arbeiten zu können muss von einer Authentifizierungsinstanz ein Schlüssel zugeteilt werden werden. Auch hier ist nicht näher spezifiziert wie diese Instanz aussehen muss. Der Schlüsselaustausch wird absichtlich aus dem Standard herausgehalten. Das eigentliche Problem ist aber, dass die RPL-Implementation von Contiki diese Sicherheitsmodi nicht beinhaltet. (Vergleich [RFC6550])

Sofern also auf dem Mikrocontroller Contiki läuft und RPL und uIP-Stack aktiviert sind, gibt es keine Zugriffsbeschränkung in das Netz und keine Vertraulichkeit der netzwerkweiten Kommunikation. Jeder Sensorknoten in Reichweite auf dem Contiki installiert ist wird in das bestehende Netz integriert, sobald er an dem Prozedere des RPL-Nachrichtenaustauschs teilnimmt.

4.3. Konzept

Der Ist-Zustand des Sensornetzes, insbesondere die Gegebenheiten des RPL, sorgen dafür, dass kein Pairing von Nöten ist. Wenn man aber dafür sorgt, dass die Netzwerkkommunikation vertraulich wird, benötigt man einen Pairingvorgang, um den Zugriff zum Netzwerk zu bekommen. Eine Verschlüsselung der Kommunikation sorgt für Vertraulichkeit der Nachrichten. Wenn man eine Verschlüsselung der Daten unterhalb der OSI-Schicht 3, auf der RPL angesiedelt ist, entwirft, können die RPL-Nachrichten nur noch von Knoten entschlüsselt werden, die bereits Zugriff zum Schlüssel haben. Dieses Kapitel geht auf die Art des Zugriffsschutzes ein. Außerdem wird erklärt wie die erstmalige Verbindungsaufnahme von statten geht und der Schlüsselaustausch realisiert wird, mit dem ein neuer Knoten Zugriff auf das Netz bekommt.

4.3.1. Zugriffsschutz

Da die Netzwerkknoten im Normalfall energiesparend und ressourcenschonend arbeiten müssen, sollte möglichst ein Verschlüsselungsverfahren verwendet werden, welches darauf Rücksicht nehmen kann. Da die an der HTW Dresden verwendeten Mikrocontroller(ATmega128rfa1) bereits eine hardwareintegrierte AES-128 Verschlüsselung ermöglichen, ist diese Lösung naheliegend. Der Vorteil liegt darin, dass nur ein Schlüssel benötigt wird(symmetrisches Verfahren) und die Daten ressourcensparend ver- und entschlüsselt werden können.

Abbildung 4.3.1.: Ver- und Entschlüsselung mit einfacher XOR-Operation [Prodromos, S. 26]

Eine Verwendung eines netzweiten AES-Schlüssels auf Ebene der Sicherungsschicht (OSI-Schicht 2) sichert den Zugriffsschutz auf das Netzwerk. (Vergleich: ContikiSec, eine Netzwerksicherungsschicht für ContikiOS, [Casado])

Dabei stehen 2 Optionen zur Auswahl. Eine einfache AES-ECB Verschlüsselung benötigt keine Veränderung der Nachrichtenpakete. Dabei werden die eigentlichen Daten einfach mit dem AES-Schlüssel XOR verknüpft. Der Nachteil ist, das die Daten sehr oft gleich/-ähnlich sind. Die verschlüsselten Daten würden ebenfalls immer ähnlich aussehen und geben dadurch Rückschlüsse auf den Klartext. Die AES-CBC Variante benötigt einen Initialisierungsvektor(IV). Dieser wird zufällig erzeugt und mit den Daten XOR-verknüpft. Dadurch sehen auch gleiche Daten nach jeder Verschlüsselung anders aus. Dieser IV muss

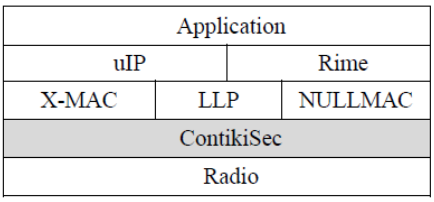

Abbildung 4.3.2.: ContikiSec bildet die Sicherungsschicht(OSI-Schicht 2) für ContikiOS [Casado, S. 13]

aber in den Paketen mit versendet werden. Umso länger der IV ist, desto sicherer ist die Verschlüsselung. Das sorgt dafür, dass die Pakete größer werden. Das heißt aber auch, dass der Energieverbrauch steigt. Der in der Forschungsarbeit (siehe [Casado]) verwendete IV ist 2 Byte lang und stellt damit einen optimalen Kompromiss zwischen Sicherheit und Ressourcenverbrauch dar (Der Beweis wird in der Forschungsarbeit erbracht).

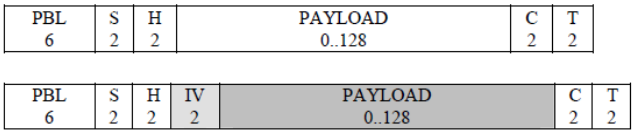

Abbildung 4.3.3.: Contiki-Paketdesign mit und ohne IV [Casado, S. 13]

4.3.2. Erstmalige Kommunikationsaufnahme

Die erstmalige Verbindungsaufnahme erfolgt genauso wie bei einem RPL-DIO, mittels eines unverschlüsseltem IPv6-Multicasts. Der Multicast soll durch eine manuelle Interaktion des Nutzers ausgelöst werden. Die Multicast-Nachricht wird vom Border-Router empfangen, sofern er in Reichweite ist. Nun muss der Border-Router die Anfrage des neuen Knotens an die Verwaltungsoberfläche des FHEM-Servers schicken, mit dem das Sensornetz der HTW verwaltet wird. Wiederum muss der Nutzer die Anfrage manuell über die Verwaltungsoberfläche bestätigen. Wenn dies geschieht, bekommt der neue Knoten mit der Antwort den netzweiten Schlüssel für die Ver- und Entschlüsselung der Kommunikation zugesandt.

4.3.3. Verschlüsselung und Schlüsselaustausch

Das Problem bei der Übermittlung des netzweiten Schlüssels vom Border-Router zum neuen Knoten ist, dass die Antwort des Border-Routers abgefangen werden kann. Damit wäre der netzweite Schlüssel für das Sensornetz bekannt. Damit dies nicht geschieht brauchen wir ein Kryptoverfahren, welches nur dem vorgesehenen Empfänger der Nachricht das Lesen ermöglicht. Dafür brauchen wir ein asymmetrisches Verfahren, wie beispielsweise

RSA. Dazu müsste der neue Knoten seiner Anfrage-Nachricht lediglich seinen öffentlichen RSA-Schlüssel hinzufügen. Der Border-Router verschlüsselt mit dem öffentlichen RSA-Schlüssel den AES-Schlüssel(netzweiter Schlüssel) in seiner Antwort. Der neue Knoten kann nun als einziger die Nachricht mit seinem privaten RSA-Schlüssel entschlüsseln. Allerdings kommen wir zu einem weiteren Problem. Asymmetrische Verfahren verursachen einen großen Aufwand bei der Ver- bzw. Entschlüsselung. Glücklicherweise gibt es bereits c-Bibliotheken die das RSA-Verfahren implementiert haben. Beispielsweise openSSL oder AVR crypto-lib. Während openSSL in seiner Gesamtheit viel zu groß ist um auf einem Mikrocontroller eingesetzt zu werden, wurde AVR crypto-lib speziell für diesen Zweck entworfen. Weiterhin ist festzuhalten das auf dem Mikrocontroller lediglich einmal eine Entschlüsselung vorgenommen werden muss. Damit kann man den Aufwand in Kauf nehmen. Die vorherige Verschlüsselung kann auf dem Rechner vorgenommen werden, auf dem der FHEM-Server läuft. Dieser sollte in der Regel keinen Restriktionen unterlegen sein, was Ressourcenverwendung angeht.

4.4. Durchführung

Die Durchführung kann im wesentlichen in 3 Abschnitte unterteilt werden. Die Implementierung des AES-Verfahrens, des RSA-Verfahrens und der Kommunikation von Netzknoten, Border-Router und FHEM-Server. Die Codeabschnitte die im Text gezeigt werden sind teilweise Ausschnitte und im Format komprimiert um nur die wesentlichen Dinge zu zeigen.

4.4.1. AES

Da AES-128(16 Byte Schlüssel- bzw. Blocklänge) in unserem Fall bereits hardwareintegriert bereitgestellt ist, müssen lediglich die Funktionen zur Verwendung in Contiki implementiert werden. Als Anleitung diente hier das Datenblatt des ATmega128RFA1 (siehe [Atmel2, Seite 94ff]), welches die zu verwendenden Register und die abzuarbeitenden Schritte erläutert.

Register Name	Description
AES_STATUS	AES status register
AES_CTRL	AES control register
AES_KEY	Access to 16 Byte key buffer
AES_STATE	Access to 16 Byte data buffer

Abbildung 4.4.1.: Register/Buffer des ATmega128RFA1 für die Verwendung des AES-Moduls [Atmel2, S. 94]

Die Umsetzung erfolgte in 5 Schritten und 8 Funktionen:

- Schlüsselinitialisierung

- Schreiben der Daten
- AES-Konfiguration und Starten der Operation
- Warten bis die Operation beendet ist
- Lesen der Daten

Dazu wurde eine Header-Datei und eine C-Datei im plattformabhängigen Verzeichnis von Contiki (contiki/platform/avr-atmega128rfa1) hinterlegt. Die Schlüsselinitialisierung schreibt einen 16 Byte langen Schlüssel in das dafür vorgesehene Register. Allerdings ist der initiale Entschlüsselungsschlüssel nicht derselbe, wie der Schlüssel, welcher für die Verschlüsselung verwendet wurde, sondern der Schlüssel der letzten Verschlüsselungsrunde.

```
1 void atmega128rfa1_aes_set_key(uint8_t *key) {
2         int i;
3         for( i = 0; i < AES_BLOCK_SIZE; i++) {
4                 AES_KEY = key[i]; } }
```

Listing 4.1: AES-Schlüssel wird in den „AES_KEY“-Puffer geschrieben

Als nächstes müssen die Daten in einen Puffer geschrieben werden. Dabei ist es unerheblich, ob es sich um Klartext oder Ciphertext handelt. Da es sich um AES-128 handelt müssen die Daten in Blöcken von 128 Bit bzw. 16 Byte ver- oder entschlüsselt werden.

```
1 void atmega128rfa1_aes_write_data16(uint8_t *data) {
2         int i;
3         for( i = 0; i < AES_BLOCK_SIZE; i++ ) {
4                 AES_STATE = data[i]; } }
```

Listing 4.2: Daten werden in den „AES_STATE“-Puffer geschrieben

Im AES-Konfigurationsschritt wird festgelegt ob eine Ver- oder Entschlüsselung durchgeführt werden soll und mit welchem AES-Verfahren gearbeitet werden soll. Zur Auswahl stehen ECB oder CBC (siehe auch Abschnitt 3.3.1. Zugriffsschutz). Dafür wird jeweils nur ein Integer als Index übergeben. Die 0 steht jeweils für ECB-Modus bzw. Verschlüsselung und die 1 steht für CBC-Modus bzw. Entschlüsselung. Wobei das Datenblatt erwähnt, dass im ECB-Modus die Entschlüsselung automatisch durch den Transceiver erledigt wird. Nachdem die Konfiguration abgeschlossen ist, kann die Ver- bzw. Entschlüsselungsoperation gestartet werden.

```
1 void atmega128rfa1_aes_ctrl_conf(uint8_t direction) {
2         AES_CTRL = ( 1 << AES_REQUEST ) | ( 0 << AES_MODE ) | ( direction
              << AES_DIR ); }
```

Listing 4.3: Das „AES_CTRL“-Register wird konfiguriert und die Operation gestartet

Nun muss gewartet werden bis die Operation abgeschlossen ist, dies tun wir, indem wir das dafür vorgesehene Statusregister abfragen.

```
void atmega128rfa1_aes_wait() {
    while( ( AES_STATUS & ( 1 << AES_DONE ) ) == 0 ); }
```

Listing 4.4: Abwarten des Operationsendes

Schlussendlich müssen wir die nun ver- bzw. entschlüsselten Daten wieder in Form von 16 Byte-Blöcken aus dem Puffer auslesen.

```
void atmega128rfa1_aes_read_data16(uint8_t *data, uint8_t *output_key) {
        int i;
        for(i = 0; i < AES_BLOCK_SIZE; i++) {
                data[i] = AES_STATE; }
        for(i = 0; i < AES_BLOCK_SIZE; i++ ) {
                output_key[i] = AES_KEY; } }
```

Listing 4.5: Auslesen der Daten aus dem Puffer nach Beendigung

Die standardkonforme Verwendung von CBC macht es nötig, dass wir eine XOR-Operation verwenden. Wobei die Verwendung je nach Verschlüsselungsrichtung leicht variiert, wie die 2 folgenden Abbildungen zeigen:

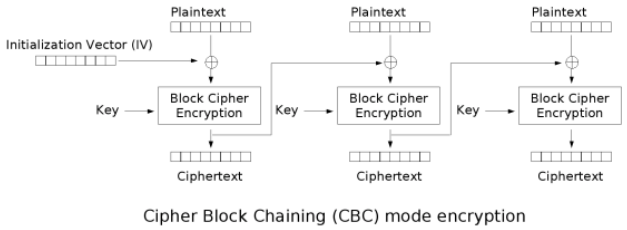

Abbildung 4.4.2.: CBC Verschlüsselung über mehrere Runden

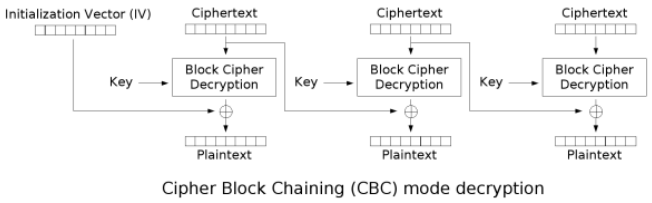

Abbildung 4.4.3.: CBC Entschlüsselung über mehrere Runden

Die XOR-Verknüpfung ist durch die folgende Funktion repräsentiert:

```
static void xor_block( uint8_t *a, const uint8_t *b) {
        int i;

        for( i = 0; i < AES_BLOCK_SIZE; i++)
```

```
5                   a[ i] ^= b[ i]; }
```

Listing 4.6: XOR-Implementation

Die letzte Funktion sorgt dafür, dass die benötigten Einzeloperationen vereinigt werden und je nach Datenlänge wiederholt werden.

```
1  void atmega128rfa1_aes_cipher(uint8_t *data, uint16_t len, uint8_t *key,
       uint8_t *iv, uint8_t direction) {
2          uint8_t i = 0;
3          uint8_t blocks = 1;
4          uint8_t temp_iv[AES_BLOCK_SIZE];
5
6          memcpy( temp_iv, iv, sizeof(temp_iv) );
7
8          atmega128rfa1_aes_set_key(key);
9
10         if( len > 16 ) {
11                 blocks += ( ( len - 1 ) / 16 ); }
12
13         if(direction == 0) {
14                 if( len < (uint16_t)( blocks * 16 ) ) {
15                         memset( data + len, 0, ( blocks * 16 ) - len ); } }
16
17         for(i = 0; i < len; i = i + AES_BLOCK_SIZE) {
18                 if(direction == 0)
19                         xor_block( temp_iv, data+i );
20                 else
21                         memcpy( temp_iv, data+i, sizeof(temp_iv) );
22
23                 atmega128rfa1_aes_write_data16(data+i);
24                 atmega128rfa1_aes_ctrl_conf(direction);
25             atmega128rfa1_aes_wait();
26             atmega128rfa1_aes_read_data16(data+i, key);
27
28                 if(direction == 0)
29                         memcpy( temp_iv, data+i, sizeof(temp_iv) );
30                 else if(i == 0)
31                         xor_block( iv, data+i );
32                 else
33                         xor_block( temp_iv, data+i ); } }
```

Listing 4.7: Zusammenfassung der Einzelschritte in einer Funktion

Die Implementierung des Initialisierungsvektors in die MAC-Protokolle von Contiki konnte bis zum Ende des Forschungsseminars noch nicht durchgeführt werden. Aus diesem Grund ist auch die sinngemäße Verwendung der AES-Funktionen noch nicht möglich und nötig. Die AES-Funktionen wurden allerdings zufriedenstellend getestet. Dazu wurde auf einem Sensorknoten ein String beliebiger Länge ver- und wieder entschlüsselt. Die dafür benötigten Parameter(AES-Schlüssel und IV) wurden dafür nur kryptografisch schwach bereitgestellt, also nicht randomisiert gewählt(Beispielpasswort: 0123456789abcdef), können aber mittels des „True Random Generator“ auch kryptografisch sicher

generiert werden. Dieser ist im AES-Hardwaremodul mit enthalten. Der im Test verwendete Initialisierungsvektor hat noch eine Länge von 16 Byte. Bei einer Verkleinerung müsste auch die XOR-Operation entsprechend angepasst werden.

Die nächsten Schritte wären die Speicherung eines festgelegten AES-Schlüssels, beispielsweise im EEPROM des ATmega128RFA1. Danach müssen die Paketstrukturen der MAC-Protokolle um einen Initialisierungsvektor erweitert werden. Schlussendlich müssen die gesamten Daten mit den implementierten AES-Funktionen bearbeitet werden, wenn es zu Sende- bzw. Empfangsaktionen kommt. Dabei muss außerdem unterschieden werden können, von wem die Daten stammen, denn es können ja weiterhin unverschlüsselte Pakete empfangen werden, die nicht zum Netz gehören(Beispiel: Pairinganfrage eines neuen Knotens)

4.4.2. RSA

Wie bereits erwähnt unterliegen Mikrocontroller Einschränkungen bezüglich Speicherplatz und Rechenleistung. Also galt es eine Implementierung zu finden, welche auf unserer Hardware umgesetzt werden kann. Es bietet sich an eine etablierte Bibliothek zu verwenden, welche die RSA-Funktionalität bereits anbietet. Dadurch ist gewährleistet das die Funktionalität korrekt und vollstädnig gegeben ist. Diese wurde mit der AVR-Crypto-Lib(Quelle: [AVRCryptoLib1]) gefunden. Diese Bibliothek ist gut geeignet, weil sie nicht die Einbettung des gesamten Codes nötig macht. Stattdessen gibt es die Möglichkeit nur Teile der Bibliothek zu verwenden. Welche Teile für welche Kryptografie-Funktion nötig sind, kann anhand der funktionsspezifischen Makefiles ermittelt werden(Quelle: [AVRCryptoLib2]). Die Funktionen konnten allerdings bis zum Ende des Forschungsseminars noch nicht getestet werden.

4.4.3. Kommunikation zwischen Pairingpartnern und Dritten

Die Lösung für die Kommunikation zwischen neuem Netzknoten und Border-Router eines bestehenden Netzes ist ein UDP-IP-Multicast, der in Contiki bereits implementiert ist. Dafür gibt es die c-Bibliothek „simple-udp.h“. Ausgelöst werden soll diese Anfrage, in Form des Multicast, durch eine Nutzeraktion. Am Beispiel eines Buttons wurde dies verwirklicht. Wenn der Button gedrückt wird, sollen über einen bestimmten Zeitintervall periodisch Multicasts geschickt werden. Das mehrfache Senden soll garantieren, dass die Anfrage wirklich ankommt, für den Fall, dass eine Anfrage verloren geht. Anzumerken ist, dass derzeit noch keine Maßnahme implementiert ist, die das mehrfache Empfangen der Anfrage beim Border-Router unterdrückt. Das Abfragen des Buttons sowie die Behandlung des Sende- und Empfangsvorgangs wurden in einem Contiki-Prozess eingearbeitet.

```
1 PROCESS_THREAD(broadcast_process, ev, data) {
2     ...
3     PROCESS_BEGIN();
4     while(1) {
```

```
5           ...
6           PROCESS_WAIT_EVENT_UNTIL(CONDITION); }//Bedingungen:
                Timerbearbeitung, Buttonstatus
7           ...
8           PROCESS_END(); }
```

Listing 4.8: Contiki-Prozess(Protothread) Aufbau

Um das Senden und Empfangen von UDP-Nachrichten zu ermöglichen muss eine Struktur vom Typ „simple_udp_connection“ definiert werden. Außerdem muss eine UDP-Verbindung registriert werden. Übergeben werden ein Pointer auf die genannte Struktur, der lokale Port, die Empfängeradresse(NULL bedeutet Empfangbarkeit von allen IP-Adressen ⇒ Broadcast), Remote Port und ein Pointer auf eine Callback-Funktion für antwortende Pakete.

```
1 ...
2 static struct simple_udp_connection broadcast_connection;
3 ...
4 simple_udp_register(&broadcast_connection, UDP_PORT, NULL, UDP_PORT,
      receiver);
```

Listing 4.9: Strukturdeklarierung und Registrierung einer UDP-Verbindung

Um die UDP-Pakete zu senden brauchen wir noch eine Absenderadresse(zur Erinnerung: Wir brauchen eine Link-Local Adresse). Diese wird mit folgender Funktion gebildet:

```
1 uip_create_linklocal_allnodes_mcast(&addr);
```

Listing 4.10: Erzeugung einer Link-Local-Multicast Adresse

Schlussendlich kann ein UDP-Paket mit folgender Funktion gesendet werden:

```
1 simple_udp_sendto(&broadcast_connection, "Test", 4, &addr);
```

Listing 4.11: Sendebefehl für ein UDP-Paket

Dieser Vorgang soll nun auf Seiten des neuen Knotens den öffentlichen RSA-Schlüssel senden. Die Callback-Funktion auf der Border-Routerseite empfängt dieses und muss die Anfrage an den FHEM-Server weiterleiten. Dort muss der Nutzer manuell die Anfrage bestätigen. Wenn dies geschieht muss der FHEM-Server den öffentlichen RSA-Schlüssel benutzen um das AES-Passwort zu verschlüsseln. Diese verschlüsselte Nachricht wird an den Border-Router zurückübermittelt. Dieser muss die Nachricht an den anfragenden Knoten weiterleiten, da nur dieser die Nachricht mit seinem privaten RSA-Schlüssel entschlüsseln kann.

Der Vorgang der Übermittlung zwischen Border-Router und FHEM-Server ist zum jetzigen Zeitpunkt nicht implementiert. Eine Möglichkeit, die in Contiki bereits vorhanden ist, wäre die Verwendung von CoAP (Request/Response Interaktionsmodel zwischen zwei Applikationen). Dazu könnte ein neuer CoAP-Nachrichtentyp(„Anfrage neuer Knoten“)

entwickelt werden. Über die Verwaltungsoberfläche des FHEM-Servers würde ein Befehl den CoAP-Client dazu veranlassen ein Request an den CoAP-Server des Border-Routers zu senden. Dieser würde als Response die Anfragen neuer Knoten (also die öffentlichen RSA-Schlüssel) übermitteln. Sinnvoll wären auch weitere Informationen zur eindeutigen Identifizierung der Knoten. Auf dem gleichen Weg könnten die dann verschlüsselten AES-Schlüssel an den Border-Router gesandt werden, der diese weiterleitet.

Literaturverzeichnis

[Ali] Hazrat Ali, A Performance Evaluation of RPL in Contiki, Okt. 2012, (Seite 17ff)

[Atmel1] Atmels AVR ATmega128RFA1 Produktbeschreibung, `http://www.atmel.com/devices/atmega128rfa1.aspx`, 28.01.2014

[Atmel2] Datenblatt ATmega128RFA1 `http://www.atmel.com/Images/doc8266.pdf`, 29.01.2014

[AVRCryptoLib1] Startseite des AVR-Crypto Lib Anbieters, `http://www.das-labor.org/wiki/AVR-Crypto-Lib`, 22.02.2014

[AVRCryptoLib2] Anleitung für die funktionsspezifischen Builds, `http://avrcryptolib.das-labor.org/trac/wiki/Building`, 22.02.2014

[Casado] Lander Casado and Philippas Tsigas, ContikiSec: A Secure Network Layer for Wireless Sensor Networks under the Contiki Operating System

[CMAC] Contikimac, `https://github.com/contiki-os/contiki/wiki/Contikimac`, 20.02.2014

[ContikiML] I2C and Contikimac on the atmega128rfa1 platform, `http://thread.gmane.org/gmane.os.contiki.devel/21130`, 20.02.2014

[Prodromos] Prodromos Mekikis, Guodong Guo, Stefano Vignati, Saad Fakher, Ibrahim Kazi, Mussie Tesfaye; Final Report: Contiki OS on ATmega128RFA1, Dez 2011

[RFC6550] RPL-Standard RFC 6550, `http://tools.ietf.org/html/rfc6550` (Seite 16f), 29.01.2014

[Wiki1] RPL, `http://de.wikipedia.org/wiki/Routing_Protocol_for_Low_power_and_Lossy_Networks`, 28.01.2014

A. Anhang

	Device/Equipment Requirements		User Actions			
Pairing Method	**Sending Device**	**Receiving Device**	**Phase I: Setup**	**Phase II: Exchange**	**Phase III: Outcome**	**OOB Channels**
Resurrecting Duckling*	Hardware port (e.g., USB) on both and extra cable		Connect cable to both devices	**NONE**	**NONE**	Cable
Talking to Strangers*	IR port on both		Activate IR on both & find /align IR ports	**NONE**	**NONE**	IR
Visual Comparison: Image, Number or Phrase	Display + user-input on both		**NONE**	Compare two images, or two numbers, or two phrases	Abort or accept on both devices	Visual
Seeing is Believing (SiB)*	Display + user-input	Photo camera + user-output	Activate photo mode on receiving device	Align camera on receiving device with displayed barcode on sending device, take picture	Abort or accept on sending device based receiving device decision	Visual
Blinking Lights*	LED + user-input	User-output + Light detector or video camera	Activate light detector or set video mode on receiving device	Initiate transmittal of OOB data by sending device	Abort or accept on sending device based receiving device decision	Visual
Loud & Clear ▪Display-Speaker ▪Speaker-Speaker	User-input on both + ▪display on one & speaker on other, or ▪speaker on both		**NONE**	Compare: two vocalizations, or display with vocalization	Abort or accept on both devices	▪Audio, or ▪audio + visual
Button-Enabled (BEDA) ▪Vibrate-Button* ▪LED-Button* ▪**Beep-Button***	User input + ▪vibration , or ▪LED, or ▪beeper	User output + One button +	Touch or hold both devices	For each signal (display, sound or vibration) by sending device, press a button on receiving device	Abort or accept on sending device based receiving device decision	▪Tactile, or ▪Visual + tactile, or ▪Audio + tactile
Button-Enabled (BEDA) ▪Button-Button*	One button on both + user-output on one		Touch or hold both devices	Simultaneously press buttons on both devices; wait a short time, repeat, until output signal	**NONE (unless synch. error)**	Tactile
Copy–and-Confirm*	Display + user-input	Keypad + user-output	**NONE**	Enter value displayed by sending device into receiving device	Abort or accept on sending device based on receiving device decision	Visual
Choose-and-Enter*	User input on both devices		**NONE**	Select "random" value and enter it into each device	**NONE (unless synch. Error)**	Tactile
Audio Pairing* (HAPADEP variant)	Speaker + user-input	Microphone + user-output	**NONE**	Wait for signal from receiving device.	Abort or accept on sending device	Audio
Audio/Visual Synch. ▪**Beep-Beep** ▪Blink-Blink ▪Blink-Beep	User-input on both + ▪Beeper on each , or, ▪LED on each, or ▪Beeper on one & LED on other		**NONE**	Monitor synchronized: ▪beeping, or ▪blinking, or ▪Beeping & blinking	Abort on both devices if no synchrony	▪Visual, or ▪ Audio, or ▪Audio + visual
Smart-its-Friends*, Shake-Well-Before-Use*	2-axis accelerometers on both + user-output on one		Hold both devices	Shake/twirl devices together, until output signal	**NONE (unless synch. error)**	Tactile + motion

***Resistant to a "rushing user" behavior**

Abbildung A.0.1.: Quelle: Arun Kumar, Nitesh Saxena, Gene Tsudik, Ersin Uzun; A comparative study of secure device pairing methods, 2009, (Seite 738)